Hypermodelling

Next Level Software Engineering with Data Warehouses

Dissertation

Tim Frey

Free to read Community Version

If you like this book, please support the work by purchasing an (e) Book

Http://book.hypermodelling.com

Hypermodelling

Next Level Software Engineering with Data Warehouses

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M. Sc. Tim Frey
geb. am 07.08.1980 in Heilbronn

Gutachter:

Prof. Dr. Gunter Saake, Universität Magdeburg
Prof. Dr. Klaus Turowski, Universität Magdeburg
Prof. Dr. Colin Atkinson, Universität Mannheim

Promotionskolloquium: Magdeburg, den 26.06.2013

Frey, Tim:
Hypermodelling: Next Level Software Engineering with Data Warehouses
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2013.

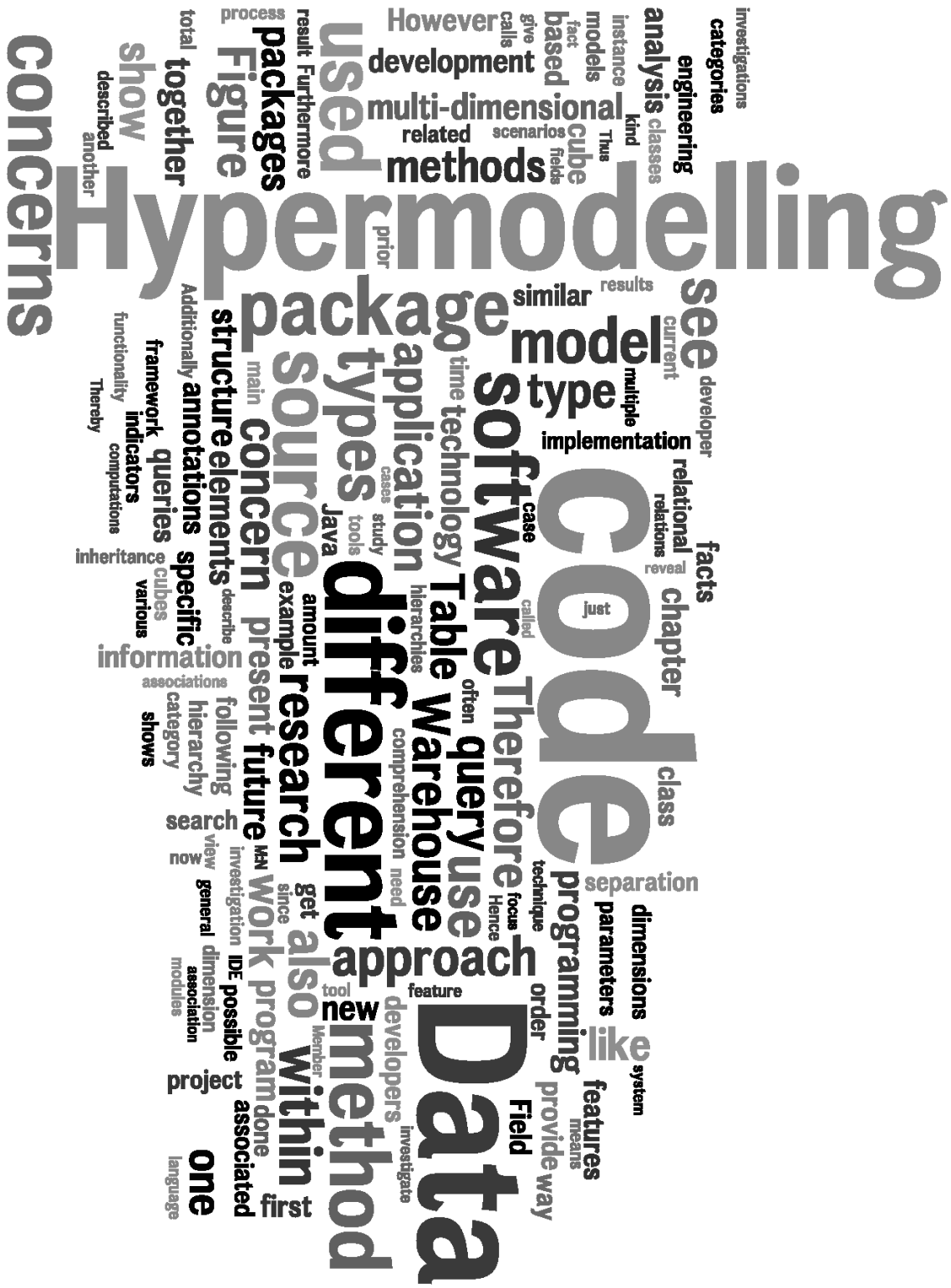
Table of Contents

List of Figures

List of Tables

List of Examples

Tag Cloud of This Work



Zusammenfassung

Quelltext ist Teil des intellektuellen Unternehmenswertes. Quelltexte in verschiedensten Projekten umfassen heute schon viele Millionen Quelltextzeilen. Dies macht es nahezu unmöglich, spezifische Details eines Projektes zu betrachten oder einen Überblick zu erhalten. Selbst das Betriebssystem eines heutigen Smartphones besitzt 12 Millionen Quelltextzeilen. Jeden Tag kommen mehr mikrochipgesteuerte Geräte auf den Markt und der weltweit erzeugte Quelltext wächst täglich enorm. Derzeit können wir kein Ende dieser Entwicklung prognostizieren. Daher ist es von starkem Interesse, Softwareentwicklungszeiten zu verkürzen, die Wiederverwendbarkeit von Software zu steigern und die Qualitätskontrolle von Software zu verbessern.

Derzeitige Forschungsarbeiten zeigen neue Softwareentwicklungsmethoden und proprietäre, in Forschungsprojekten entwickelte Werkzeuge zur Quelltextanalyse. Um diese proprietären Werkzeuge zur Quelltextanalyse in Unternehmen nutzen zu können, wären diese Unternehmen daher gezwungen, akademische Werkzeuge ohne professionellen Herstellersupport selbst einzuführen. Aus diesem Grund wird ein neuer Ansatz zur Quelltextanalyse benötigt, der sich in die bereits bestehende Systemlandschaft in Unternehmen integriert.

Diese Arbeit führt den Hypermodelling Ansatz ein, der es ermöglicht, Data Warehouse Systeme, die in vielen Unternehmen schon vorhanden sind, in der Softwareentwicklung zu verwenden. Zusätzlich wird die Anwendung des Verfahrens gezeigt und es wird auf technische Details einer Implementierung von Hypermodelling für Java eingegangen. In verschiedenen Anwendungsszenarien verdeutlichen sich die Vorteile im Vergleich zu vorherigen Ansätzen. Unter anderem können nun Abhängigkeiten und andere Beziehungen im Quelltext mit geringem Aufwand untersucht werden. Zusätzlich wird demonstriert, dass dieser Ansatz dazu geeignet ist, mit vertretbarem Aufwand Management Dashboards zu erstellen. Ferner zeigen erste Tests, dass Hypermodelling dazu geeignet ist, neue Quelltextrevisionen zu planen und erweiterte Informationen aus Quelltext auszulesen. Erste Anwendungen, um Softwareentwicklerwissen zu bestimmen, Entwickler-Fehlerkorrelationen zu berechnen oder Strukturinformationen wie Kohäsion zu untersuchen, verdeutlichen dabei das erweiterte Potential von Hypermodelling. Abschließend wird Hypermodelling als Web- und Entwicklungsumgebung basierte Quelltextsuchmaschine verwendet und in einem Szenario für Echtzeiterkennung von Quelltextduplikaten und Quelltextempfehlungen eingesetzt. Dies zeigt Verbesserungen zu verwandten Arbeiten in diesen Gebieten.

Diese Arbeit ermöglicht es ab sofort, Data Warehouse Technologie für Forschung und Praxis in der Softwaretechnik zu verwenden. Unter anderem können die Ergebnisse dieser Arbeit in den folgenden Bereichen verwendet werden: Quelltextsuche, multi-dimensionale und statistische Quelltextanalyse, überprüfende Quelltextstrukturanalysen mit automatisierter Berichtserzeugung, Echtzeiterkennung von Quelltextduplikaten und Empfehlungen, Datenintegration und Quelltextüberarbeitungskontrolle.

Zukünftige Arbeiten und Implementierungen können sich auf spezialisierte Anwendungsfälle, Hypermodelling basierte Quelltextanalyse oder die Weiterentwicklung von Hypermodelling konzentrieren. Weiterhin zeigen vielfältige Synergien mit verwandten Arbeiten vielfältige Forschungsmöglichkeiten und zukünftige weitere mögliche Anwendungsszenarien.

Executive Summary

Source code is a part of enterprises intellectual value. Today, the amount of source code in projects grows into millions of lines of code. This makes gaining overview and insights about software development projects barely impossible. Even a Smartphone runs at 12 million lines of code. Additionally, more and more microchip driven devices hit the market every day and the amount of source code is continuously growing. Today, we cannot estimate if this development will ever hit a limit. Therefore, reducing development times, improving software reuse and enhancing quality control of software is of crucial interest.

Current research shows new approaches for software development and provides custom source code analysis infrastructures. In order to apply such code analysis infrastructures, new and custom academic tools without professional vendor support would need to be introduced in enterprises. Therefore, a new source code analysis approach is needed that integrates in the current system landscape of a company.

In this work, we introduce the Hypermodelling approach and leverage already in enterprises available Data Warehouse technology for software manufacturing activities. Complementary, we provide concrete technical details about an implementation of Hypermodelling for Java. Different usage scenarios show advancements to related research and practice. In special, we present insights how dependencies and other relations of code structure can be revealed with little effort. Additionally, we report about the reasonable effort to create management dashboards for code structure. Furthermore, we test Hypermodelling to plan revision updates and to compute advanced information out of code. First demonstrations to mine developer knowledge, to compute developer-style issue correlations and to report cohesion indicators underline the potential of the approach. Finally, a web and development environment based code search engine and a code clone recommendation scenario show also advancements to related work in the corresponding areas.

Now, Data Warehouse technology can be used for software engineering research and practice. Among others, our current results can be used to carry out the following business cases: Code search, multi-dimensional and statistic code analysis, code structure audits with automated reporting, live code clone detection and recommendation, data integration and revision update progress control.

Future research and implementations can focus on specialized scenarios, Hypermodelling based source code analysis or to the advancement of Hypermodelling itself. Furthermore, plenty of synergies with related work identify additional research trails and future application scenarios.

Abstract

We are working towards a smart information infrastructure: Smart cities, smart grids, car interactions and billions of people interacting using their Smartphones. More and more micro chip driven devices hit the market every day and the amount of software driven devices is continuously growing. In plenty of development projects, the source code is already growing into several million lines of code. A Smartphone is running on roundabout 12 million lines of source code. A print of this results in 193.560 pages of A4 copy paper. This is comparable to a stack of paper, lasting over 19 meters. Revealing information out of such a code base, in a structured way, is barely impossible.

Commonly, source code fragments are grouped together into modules to enhance the maintainability and reuse of their functionality. Thereby, software developers follow the paradigm to separate concerns and try to achieve modules that follow a primary and only responsibility. Nonetheless, the concern space of a program is multi-dimensional and it is often hard or impossible to separate all concerns with object-oriented programming mechanisms. Therefore, researchers develop multi-dimensional modularization techniques, challenge complexity with concern centric projection advancements or develop multi-dimensional navigation approaches.

However, plenty of available source code exists already. Therefore, researchers develop source code analysis tools and reveal insights about the current code repositories. Such investigations reveal which weekdays in a project lead mostly to false bug closing reports or similar things. The custom code analysis infrastructures of the researchers make it hard to transfer and redo the investigations in industry practice, because "yet another tool" has to be introduced to replay investigations. Introducing a new tool in enterprises requires maintenance and support of a tool vendor what is not available for academic code investigation infrastructures. Additionally, current analysis tools are created for research and not for in-enterprise application. This results in the lack of holistic tool chains and a reduced functionality for just one kind of research. Hence, an approach is needed that integrates big data source code analytic capabilities into the system landscape of enterprises.

We introduce Hypermodelling and leverage Data Warehouse technology to provide a suitable solution for this challenge. Data Warehouses are especially designed to handle large scale multi-dimensional data and get widely applied in enterprises. They provide consistent tool chains for flexible data exploration, navigation, analysis, visualization, reporting and management. We describe how program structure can be expressed in a Data Warehouse leverage their infrastructure for software engineering. This enables to explore and analyze the concern space of a program with Data Warehouse technology. Additionally, this advances the current state of the art in program analysis, since the concern space of programs is multi-dimensional and can now be explored suitable technology that is especially designed to handle multi-dimensional data relations.

Aside the obvious, we investigate further benefits of Hypermodelling. We test and explore the advancements, by providing an implementation and different application scenarios for the Java programming language. We show that the leveraged Data Warehouse analysis capabilities provide a suitable mean to reveal facts about insights about dependencies and other program structures, easily. We recognize that a comparable commercial custom tool to analyze dependencies in source code seems rather limited compared to a holistic Data Warehouse tool landscape that offers plenty of data exploration and visualization options. In another investigation, we show advanced analytic capabilities and compute statistics about mental categories. This underlines the advancements to use Data Warehouse technology, again.

As addition to the pure analytic investigations, we show that even simple to operate dashboards can be manufactured with reasonable effort. Those dashboards enable investigations of source code structure for specific use cases without any knowledge about Data Warehouse technology. We demonstrate advanced use cases and report indicators about framework knowledge of developers, control cohesion as quality indicator and show a cockpit to investigate the inheritance structure. This is another advancement of the start of the art, because similar dashboards are limited to project management information. They do not allow detailed drill downs into the source code structure and its internal relations. We finish the reporting scenarios by adopting indicator based planning from

Data Warehousing. A first application test in a software migration scenario reveals that indicator planning can help to compute the update progress of the migration.

We implement Hypermodelling for the well-known Eclipse development environment. Different code search based application scenarios demonstrate the usefulness in the development environment. Additionally, we reveal a model for future development environments that utilize Data Warehouse components. This way, we bridge our development environment based Hypermodelling implementation the Data Warehouse based one. We create a code search engine on top of our Data Warehouse implementation and evaluate its suitability for search based application scenarios. As result, we recognize improvements of Hypermodelling based code search engines to other comparable code search engines. Nonetheless, we demonstrate that Hypermodelling can be used for code recommendation and clone detection, on top. Related work indicates potential synergies and improvements with prior code recommendation and code detection techniques.

Now, developers can use Hypermodelling to do code search and to use development tools on top of it. Software architects can use it to investigate the code structure and to derive dashboards for the most common executed investigations with reasonable effort. Managers can use those dashboards to do standardized audits of the source code structure. Scientists have an infrastructure with ad-hoc statistic computation and visualization capabilities directly at hand. Those investigations can be carried out on the same infrastructure that is already available in enterprises. The possibility to use the same infrastructure enables a faster transfer of research results in the future.

Future research can reveal specialized application scenarios for the use of Data Warehouse technology within software engineering. A next step to advance Hypermodelling is to express the time-dependent movements of source code. Another avenue is to analyze software operated machines, such as cars or robots, and integrate physical indicators and runtime information. Such physical indicators can be hardware failures, execution time, energy consumption and data about software product lines. The integration of these data opens new investigation possibilities for which all the analytic capabilities of Data Warehouses will then be available. Finally, researchers can apply Hypermodelling functionalities that are already available and reveal statistics about code structure, advance our scenarios or get into synergies with related research.

Acknowledgment

"A day may come when the courage of men fails, when we forsake our friends and break all bonds of fellowship. But it is not this day."
Aragorn. The Lord of the Rings. The Return of the King. 2003

I remember 2008. At this time, my idea came up to use Data Warehouse technology in programming. In first estimations, I considered research about this topic as too time consuming and effortful. Hence, I put it aside. However, the seed was planted and came back. Finally, I wrote my Master thesis in the direction of the idea and followed up with my PhD research. When I look back, my estimated effort was exceeded in ways beyond my imagination. Luckily, the results also outranged my expectations. Today, you can read the outcome. I hope that you will not take as long to read as it took me to develop and to write.

If you want to give me feedback, note improvements or you find mistakes, please let me know: **tim@hypermodelling.com**

The result of my efforts would not have been possible without support of others. In all those years, when I worked and refined my ideas, there were people in my life that motivated me and believed in my ideas. In special, my thanks go to Gunter Saake and Veit Köppen that gave me support to do my PhD and helped me to advance the quality of this dissertation. Colin Atkinson believed in my ideas and motivated me to go on with this thesis by giving me inspiration, discussion and feedback. The advice of Klaus Turowski helped me to embed my thesis into information science research and to gain the best readable structure and presentation of my results. All together, their support kept me going and improved the quality of this thesis in many ways.

However, there are plenty of other people that helped me on my way. I thank all the people gave me feedback, inspired me to go on, read and improved parts of my work and cheered me up with a laugh, coffee or beer. In the following, I credit a few of these people in representation to all of you that helped me:

- My entire and extended family
- My friends and colleagues
- Christian Schmitt
- Marius Gelhausen
- Alice Jacoby
- Matthias Gräf
- Maximilian Gärber
- Ralf Schimkat
- Joachim Selke
- Tino Noack
- Andreas Stefik

Symbols and Abbreviations

In this work, we use different symbols and abbreviations.

Abbreviations

AOP	Aspect-oriented programming
API	Application Programming Interface
BI	Business Intelligence
DFM	Dimensional Fact Model
DSL	Domain Specific Language
IDE	Integrated Development Environment
JEE	Java Enterprise Edition
MDSOC	Multi-dimensional separation of concerns
OLAP	Online Analytical Processing
OOP	Object-oriented programming
OSM	Orthographic Software Modeling
SQL	Structured Query Language
UML	Unified Modeling Language

Symbols

This symbol credits a definition, a main conclusion or something we refer back to in later chapters.

This symbol indicates additional important remarks.

This symbol indicates additional tips and further research avenues that we see worthwhile mentioning.

Part I. Introduction and Motivation

1. Introduction

“What's that?”
- “It's blue light”.
“What does it do?”
- “It turns blue.”
Hamid and John Rambo. Rambo 3. 1988.

Abstract. Computer programs run on plenty of devices. Their source code is growing into several million lines of code. Even a small set of functionality already exceeds thousands of lines of code. More and more micro chip driven devices hit the market every day. The amount of worldwide applied source code is still growing and we cannot estimate an end of this process. Investigating computer programs is, and gets more and more, complex. This dissertation contributes to these challenges in software engineering. In this chapter, we present our motivation, research methodology and structure of our work.

1.1. Introduction

In order to provide our research results, we lean our method against the principles of design science in information systems research framework as described in [123]. We do this, because our research is located in the area of Data Warehouse information Systems and the context of programming languages. Therefore, we see the research methods in information science as suitable for our work. However, out of our specific interdisciplinary topic, this thesis follows the most readable structure and presents the information that corresponds to the specific research guidelines in various places. We illuminate credited the guidelines of design science research and describe in a detailed way, where the corresponding information is arranged in our work. Furthermore, we provide an overview and to excel detailed insights about this work in a summarized way.

In the following, we describe contribution of this chapter to our work. We do this in every section of this work to credit our main intention for the various chapters. Then, we present a reading guide of the chapter.

1.1.1. Contribution

Our reading goal for this chapter is to provide a basic understanding for the importance of software engineering research. Additionally, we use this chapter to provide a classification of our research. We address the key problem of the immense size of code bases, resulting in challenges within code studies. All together, this chapter gives a structured overview, motivates for our work, points out the key contributions of this thesis and gives insights about our research strategy.

1.1.2. Reading Guide

First, we narrow the area of software engineering down to the specific parts wherein our research contributes. We present our motivation how and why our work is important for today's and future software engineering. Afterwards, we illuminate that the provided results of this thesis originate a search process for information technology artifacts. Then, we describe the main principles that we use to defend, present and also legitimate our results by rigorous scientific methods.

Furthermore, we credit the specific information technology artifacts that we develop as result of this thesis. We show contributions in diverse areas that allow future researchers to build on our intellectual concepts and concrete technological implementations. By doing so, we get into our provided artifacts and describe how those are evaluated. In order to give insights about our

evaluation strategy and results, we provide an overview of the effort that was done to justify our work.

We warp up the concrete benefits of our work by presenting the overall improvements by this thesis. This way, we credit the holistic result of introducing a new technique for software engineering and to focus on the high level improvements. However, one key point in research is the communication of the results to other researchers and to gain feedback from the community. Therefore, we describe how our research was communicated. Last but not least, we provide an outline of this thesis.

1.2. Software Manufacturing

Companies spend a large amount of money on software maintenance [162]. Software creation and its maintenance are called software engineering. Hence, software engineering focuses in the creation of software systems, i.e. programs [83]. Software engineers, often called developers or programmers, encode functionality to create such a program [241]. Commonly, we refer to source code when we think about encoding functionality into a program. One key factor of software manufacturing costs is the maintenance of source code.

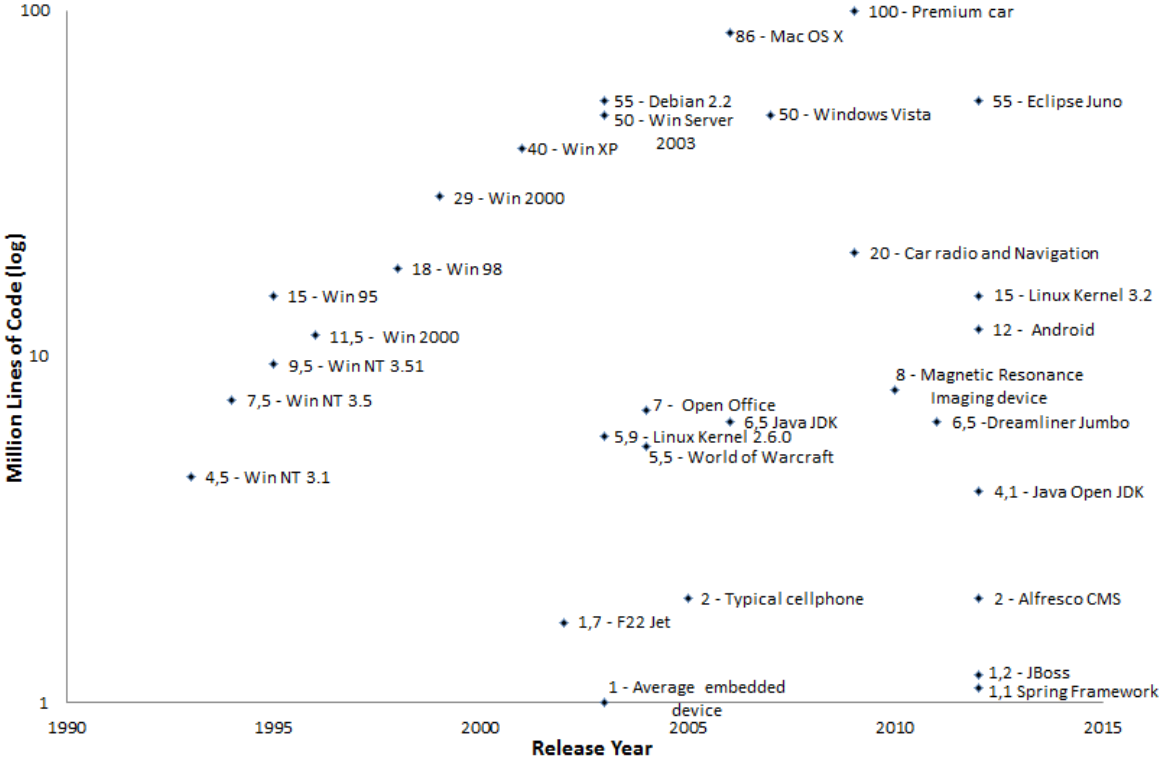
However, the creation of a program is a complex scenario that contains many obstacles to achieve the desired quality and functionality of the program [53, 32, 218]. Literature defines a software system sometimes in broad terms as the outcome of a development effort, containing software and hardware [241]. Others refer to the term of a software system as a bundle of artifacts which consist out of software elements [210]. In this work, we focus on the challenges that are faced within the development process. In detail, we concentrate on the problems, directly associated with the manufactured software system that is created by writing source code. We understand the term of "directly associated" with the software system as the artifacts that are known and used by developers. For instance, a Java developer would commonly not refer to the operating system as requirement and rather name the virtual machine version as prerequisite. Moreover, developers normally address functionality that is offered by libraries or already encoded source code when they manufacture a software system. We credit this kind of practical issues and define and define the term of a software system as follows:

A software system, short system, also called software product or program, is a bundle of software artifacts that would be named by a developer as prerequisite to execute a desired set of functionality. The software artifacts, also named software fragments, offer small pieces of functionality. Their functionality gets composed together by encoded logic of developers to offer a desired set of functionality of a system. Commonly, a program is encoded by writing source code.

1.3. Motivation and Problem Relevance

Like we defined before, all the artifacts are, created by programming functionality part of the software system. Today, more and more software is embedded into the diverse devices. The size of programs in lines of code grows into millions. Developers face the challenges to navigate and understand the complexity of this code bases. Also, quality assurance and investigation gets more and more difficult as the code repositories grow. We give an impression about the immense size of code repositories in Figure 1.1 . There, we see different popular programs and their estimated lines of code size.

Figure 1.1. Exemplary Lines of Code



This figure shows measurements and estimations of various software projects and a time association when the software was released. The origins of the data are described in the appendix (Section B.1).

In order to extend, refactor or to use already encoded functionality, developers must manage this kinds of projects. We can imagine the size of this code bases better, when we consider printing just one million lines of code. One million lines of code (MLOC) printed in font size 10 on A4 paper results in 16.130 pages. These are 32 packs of 500 pages copy paper, where each pack has the size of 5 cm. This means 1,6 meters of stabled paper for 1 MLOC. Now, imagine searching a special page within these papers. Thereby, we have to consider that source code references other source code by function calls. Furthermore, we have to realize that we use different of those programs together. For instance, the alfresco project (2 MLOC) utilizes the spring framework (1,1 MLOC) and a Java enterprise application server like JBoss (1,2 MLOC). Thus, if we just extend alfresco, we "play" with around 4 million lines of code. An overview of all the used source code is barely impossible.

For the future, we see an immense movement towards smart and "microchip driven" houses, intelligent power networks, instantaneous and ever lasting mobile network connections with smartphones, refrigerators that are communicating online and thousands of other devices that will get an IP address and get connected to the internet. Our global networked infrastructure allows cars to pass data details, such as traffic hotshots, accidents and even crash information, to each other. All of this leads to an incredible amount of software driven microchips. The amount of encoded source code in this "smart world" will challenge the skills of future generations.

Even though that the code bases will grow even more in the future, we already face huge challenges today. Just a tiny and simple program numbers in tens of thousands lines of code. We visualize this and depict printed A4 pages of diverse small projects in Table 1.1. A simple logger facility of Java already grows into an immense size of 323 pages. Searching this amount of pages for a specific pattern is very difficult. Likewise, we see primitive collections extensions, such as bags or multi sets grow to nearly 2500 pages. An xml transformer even beats this value. Finally, we show a database that is considered to be lightweight. We see, this lightweight database engine consists of over half a million lines of code. Hence, we recognize that navigation and getting overview of a program are key challenges of programmers. Altogether, we note:

One challenge in software development is the immense size of code bases.

Table 1.1. Projects printed in Pages

Project	Lines of Code	Printed A4 pages with 62 Lines per page (Font size 10)	Purpose of the Project
Simple Logging Facade for Java (SLF4J)	20.267	323	A logger facility that provides various options for logging.
Apache Commons collections	151.727	2.448	A library that offers supplemental types and operations to handle collections.
Apache Xalan	204.446	3298	XSLT processor for transforming XML documents
HyperSQL Database Engine	569.222	9.181	A tiny lightweight database that can be embedded into Java programs [a]

[a]All lines of code values have been retrieved via http://www.ohloh.net at 6.7.2012

1.4. Solution Search Process

In our work, we present the Hypermodelling approach as solution for the immense size of code repositories. However, Hypermodelling does not stand alone and is founded and inspired by the current state of the art. Hence, Hypermodelling, is the result of a search process in a solution space of prior art and today's challenges.

Our search process to create Hypermodelling was originally done by a step by step refinement of a solution for the current challenges. In order to spare the reader our time-intense and iterative steps, we present the summary of our research process in a more structured way, than it was done in reality. First, we describe the key contributions as related research and derive the demands on a new technique. Then give an overview of the needed knowledge about Data Warehousing. Lastly, we present Hypermodelling as fulfillment for those requirements and describe how and why it matches the demands.

Additionally, we present application scenarios and implementations of our approach that are the result of a search process for evaluation scenarios. Generally, the applications show the usefulness of this work in diverse areas. However, we are careful not to claim that the presented Hypermodelling technique is complete and we have covered all application areas. The main goal of the search of this work is to show the opportunity to integrate different approaches together. Hence, our search process of application scenarios is a result of a heuristic investigation and not a complete and comparison. Therefore, the in detail comparisons and further opportunities of Hypermodelling are left for future research and do not claim for a completeness of our comparison. Nevertheless, we induce plenty of other research areas and comparisons to our work. Those can be used as a staring point for an in detail and expanding search process in the future.

1.5. Research Rigor

Like we described before, Hypermodelling is derived by the motivation of current and future challenges in programming and a result of a search process to create a new and beneficial information technology artifact. Hypermodelling builds on Data Warehouses and combines their technology with research in software engineering. Therefore, the presented work has theoretical and practical foundations in both, software engineering and Data Warehousing. In order to credit both disciplines and to argument for our results with rigorous scientific methods that are common within these areas, we focus on established techniques to present our work.

We use notations for data cubes that are published within the Data Warehousing community. Additionally, we uses relational models that are founded in database theory. Together, with the

computation models for the data cubes our research can be implemented with concrete Data Warehouse technology.

Research about new methods in software engineering is founded on programming languages and graphical models of program structure. Often research in software engineering is presented with practical samples of a well known programming language. In this work we use continuously graphical models of the unified programming languages and artifacts out of the well defined and academic accepted programming language Java.

We show concrete mappings from data cubes and relational models to the Java programming language. This mapping enables the use of formalisms in Data Warehousing now with Java and other similar languages.

In summary, this work uses applied examples, accepted notion methods and as addition descriptive and visualizing figures in developing and describing the resulting artifacts and research contributions of this work.

1.6. Provided Artifacts

We use established methods to present results of our work. Those results manifest as different technical and conceptual artifacts. In general, we provide artifacts in four different categories. First, we refer to technology arrangements that describe how different technologies can be assembled together to leverage its combination. Then, we stick to conceptual models how the technologies can be utilized in detail. After this, we point out the concrete implementations that we provide in our work. And, lastly, we address specific applications and scenarios.

In the following, we describe the artifacts in the different categories in more details and reference the chapters wherein those can be found.

1.6.1. Technology Arrangements

The first category of provided artifacts are arrangements of technologies and tools. We describe those in the following:

We provide a general arrangement methodology that describes the architecture of a multi-dimensional access for source code and associated artifacts. Additionally, we provide a generic architecture arrangement of Data Warehouse technology for this access (Chapter 5).

We also define various kinds of tools which can be created on top of such arrangements. Furthermore, this includes a tool chain how the different technologies that are commonly used within Data Warehousing can be adapted (Chapter 5).

Additionally, we describe an arrangement of Data Warehouse technology within the development environment and give details how our new Hypermodelling approach integrates within it (Chapter 10).

1.6.2. Models

The second part of provided artifacts are models and their terminology. The models address different perspectives. A Data Warehouse centric perspective and a perspective of components that provide the necessary data within the development environment. Nevertheless, the two different models are additive. The Data Warehouse models describe the computation and the models for the development environment describe a process centric alternative realization. In addition to those, we also look beyond our own nose and present an appendix about program comprehension and present models about categorization in psychology.

Data Warehouse related models define concrete graphical representations and computation directives. Additionally, we also introduce the new vocabulary of dimensions, facts and their relations to the area of programming (Chapter 4) and map programming constructs to those models (Chapter 5). This way, our models also includes the addition of a new terminology that can now be used within the area of program analysis. In special, our models contain a holistic relational

and multi-dimensional model for Java source code and the associated processing and computation directives (Chapter 6). This enables other scientists to extend or specialize our provided models for specific use cases.

In order to visualize the arrangement of the data not only from a Data Warehouse centric perspective, we provide further insights about a processing cycles within the development environment (Chapter 10). We show the application and usage components within an development environment and how queries with can be implemented. Furthermore, we present a reference architecture of future development environments that contain Data Warehouse technology. With this information future research can integrate the concrete Data Warehouse models and computations within the development environment .

As excursus to the data models, we also provide several visualization models about categorization in psychology and their relation to programming mechanisms (Appendix A). Finally, we reveal a mental model that is founded on research in psychology how program studies work. This model can now be used to compare it to other program comprehension theories and explain the source code study process based on psychological categorization research.

1.6.3. Implementations

The third kind of artifacts are concrete implementations of our technique. Through implementations, we evaluate and verify if our conceptual models can be applied in real world information technology. The main implementations that we do are described in the following.

A Source code Extraction and Transformation Parser

We develop a source code parser for the development environment that enables storing data directly in our relational model. The data extract by this parser is used to test our model artifacts. However, program parsers are state of the art in software engineering research. Therefore, we do not focus on our parser in this work and focus purely on the relational models that are finally populated with data of our parser.

Model Realization

As addition to the models, we develop concrete implementations of our multi-dimensional cube models within a Data Warehouse to proof their application ability. Thereby, we realize the models with the Microsoft Analysis Services server 2008 R2 and demonstrate its actual application with queries in the different usage scenarios (Chapter 6).

Development Environment Realization

In order to apply our technique within the development environment, we present a prototype for the Eclipse development environment (Chapter 10). Thereby, we enable new possibilities within the development environment, like drag and drop queries, result filtering and drill-downs.

Videos of Realizations

We provide videos of the concrete applications to enable a visual insight about the possibilities of our research. These videos describe different analysis that can be done with our approach and proof that the realizations have been done. Furthermore, the videos provide fast overviews and explanations of the final application results. The videos can be found in the youtube channel or at the Hypermodelling homepage.[1]

1.6.4. Application Scenarios

We show diverse application scenarios of Hypermodelling that we use to evaluate its usefulness.

[1]http://hypermodelling.com http://www.youtube.com/user/hypermodelling - 7.12.2012

First, we provide source code analysis scenarios that show how our technological infrastructure is applied to analyze source code. Thereby, we illuminate how different indicators are computed and give advanced details how Hypermodelling can be used to investigate source code structure (Chapter 7).

Additionally, we present how we use our technology to create a cockpit (e. g. code structure management dashboard) for code structure. We measure the effort to create the cockpit and present supplemental reports that can be included in future cockpits (Chapter 8).

We provide first insights about a future scenario how our approach can be used to plan associations in source code. We evaluate the approach preliminary by the migration of a demo application and get first indications that this can be a future research trail (Chapter 9).

We present different evaluative query based scenarios for Hypermodellings application in the integrated development environment. We advance the query based scenarios by referring to related research about code recommendation and describe how Hypermodelling in the development environment can advance this research trail (Chapter 11).

1.7. Evaluation

This work evaluates the Hypermodelling technique by using analytical, descriptive and experimental techniques. Thereby, we use the applications and implementations that we create on top of our main concept.

In the following, we describe which applications we investigate and how we do so. We follow the sequence of the chapters, wherein the different applications are described.

1.7.1. Complexity and Trade offs

First (Chapter 6), we investigate the general trade offs of Hypermodelling and study its static complexity properties. We apply Hypermodelling for Java and a Microsoft Data Warehouse and investigate how the specific data structures look like. Additionally, we reflect the known shortcomings of the implementation. Furthermore, we compare the implementations complexity properties to custom code parser for a single use case.

1.7.2. Analysis Scenario Investigations

We investigate the suitability of Hypermodelling for code analysis and present a scenario to use Hypermodelling to analyze object-oriented inheritance and dependencies of an application (Chapter 7). We advance the scenario even further and leverage different investigations into management dashboards and reports (Chapter 8). Various business cases reflect the usefulness of our approach. Furthermore, we use a dashboard realization as experiment to quantify the effort of Hypermodelling based realizations. We find indications for small efforts, what indicates time saving benefits of our approach. All over, the analysis scenarios demonstrate the suitability of Hypermodelling for code investigations and its value for flexible report based code investigation scenarios.

1.7.3. Future Code Planning

We evaluate if Hypermodelling can be used to do planning of code evolution, similar to the economic planning scenarios in Data Warehousing (Chapter 9). Therefore, we give a first example how it can be used to plan future code structure. First results demonstrate the need for further investigations and give indications that Hypermodelling can be used for code structure planning.

1.7.4. Development Environment Application

We evaluate the application of Hypermodelling in the integrated development environment (Chapter 10). A prototype for the development environment shows that the application of Hypermodelling there is generally possible. Additionally, we present different evaluative scenarios that demonstrate the utility of our approach. Supplemental, we present the proposal to integrate Data

Warehouse technology into the development environment and verify the realization possibility by providing an implementation of a code search engine.

Lastly, we show how to use Hypermodelling for code recommendation (Chapter 11). There, we show how Hypermodelling can advance recommendation systems by a query based approach.

1.7.5. Related work

We compare Hypermodelling with related work (Chapter 12). Thereby, we summarize the features of related work and the ones of Hypermodelling and provide an overview how Hypermodelling integrates and differs from related research.

The comparison to related work shows that Hypermodelling is an integrative technique and that it affects many areas of software engineering. This indicates, again, its usefulness.

1.7.6. Code Study

Additionally, we do an excursus and present a program comprehension model based on categorization research of psychology in the appendix (Appendix A). This model shows that our mind uses flexible and not fixed categories. Hypermodelling fits to this, since it allows multiple and flexible viewpoints. Additionally, we use Hypermodelling to do a statistic investigation about categories in source code. Again, we see that Hypermodelling is suitable to do code analytics.

1.8. Contributions

We discussed how we evaluate our work. In the following, we wrap up our contributions.

Our first improvement is to provide a summary of advanced software engineering methods. We present the state of the art and underline the need for advanced approaches in software engineering. Especially, we credit that separation of concerns is hard to apply in object-oriented programming and multi-dimensional programming approaches provide benefits. All together, we summarize the different challenges, key contributions and gaps that exist within software engineering research. Secondly, we introduce the area of Data Warehousing to the software engineering community. Thereby, we give insights about the technology and its way of operation.

Third, we propose, implement, discuss and evaluate the usage of Data Warehousing in software engineering, what we name Hypermodelling. Thereby, we define and answer three main research questions:

1. Is it possible to equal

   - means to apply separation of concerns in object-oriented languages and their internal hierarchies and compositions,

   - with data cubes, containing dimensions, dimension members, hierarchies and indicators of a Data Warehouse?
2. If so, how does an abstract framework for the implementation look like?
3. And, is there further benefit of loading programs into a Data Warehouse and analyzing them with Data Warehouse technology?

This way, we contribute with a new technique to the current state of the art. Furthermore, we evaluate its potential benefit through various application scenarios. Specifically, we contribute and advance the current state of the art by the following eight high level improvements:

1. The Hypermodelling approach

   We provide a new perspective for software engineering due the proposal to use Data Warehouse technology for source code representation. This is a straightforward improvement, since the concern space of a program is multi-dimensional and Data Warehouses are designed to handle multi-dimensional data. Furthermore, current code bases grow and Data Warehouses are built for big data. Hence, our approach advances the current approaches by the proposal of technology

that is suitable to those demands. The result of this approach describes a holistic framework how Data Warehouse infrastructures can be used in software engineering.

2. Relational and multi-dimensional source code schema, containing computation directives

   We leverage the possibility to represent source code within a Data Warehouse by presenting a concrete relational schema for Java source code structure. Thereby, we provide deep insights about the computation of the relations in Data Warehouse cubes. As it is for now, other applications and source code investigations can be created on top of the relational schema or adapt our results for desired specializations.

   Additionally, the multi-dimensional representations and computations of source code relations in a Data Warehouse can serve as bridge to study Data Warehouse performance mechanisms and their application in source code query tools.

   This is a direct improvement, because before our schemata, there was no direct relation between source code query tools and Data Warehouse cubes illuminated. Traditionally, performance increasing is a huge area in Data Warehousing. Through our models, researchers can now study performance tuning in Data Warehousing and transfer research results from there to source code query tools and analysis. Additionally, the technical implementation enables future researchers to use the whole toolboxes that are shipped with Data Warehouses for source code investigations.

3. Code analysis

   We utilize Hypermodelling for source code analysis and provide an in detail investigation how it can be used to analyze software variance and dependencies. Specifically, we show that Hypermodelling can be used to explore diverse facts on variances of frameworks. We depict reports and visualizations that reveal facts on software variances, dependencies and give insights into "method call statistics". Hence, our contribution is to demonstrate that Hypermodelling can be used to do investigations on software variance an reveal several facts about a large programs architecture. Our reports give first clues what types of queries can be composed with Hypermodelling. This supports our hypothesis that Data Warehouse mechanisms is beneficial for software investigations.

   However, in detail our investigation reveals different facts of specific industrial applications. We find that class inheritance is far more used then interface inheritance. As addition, we provide first indications that methods with many parameters get called less likely and the reuse drops. All together, the detailed investigation shows that Hypermodelling is suitable to investigate programs with queries and future researchers can use Hypermodelling to advance our investigations of software variance and dependencies.

4. Code Structure Reports

   We introduce the new technique for code quality to control by describing and implementing reporting dashboards for code structure. This way, we contribute with a new kind of cockpits to the current state of the art. As advancement to prior research it is now possible to create cockpits for code structure. We evaluate the feasibility and how Hypermodelling can be used to build a cockpit for parts of the code and provide investigations that show the implementation effort is reasonable. With our cockpit it is possible to investigate the dependency on frameworks on various abstraction levels. This helps project managers to investigate code more easily with customized dashboards. As completion and lookout, we present further reporting scenarios and demonstrate advanced reporting possibilities. Those serve as inspiration for additional use cases that are now possible.

   Out of the whole measured effort to built this specific cockpit and the other reports, we can conclude that Hypermodelling allows it to build cockpits for code structure for different scenarios in a reasonable time frame. Now, specific code structure reports for other use cases can be determined and created.

5. Structural Association Planning

   We present a first adoption of a Data Warehouse similar planning mechanism to source code structure. Therefore, we credit research in Data Warehousing about strategic economic planning

scenarios and do a first step in utilizing this research within software engineering research and practice. This way, we contribute a new way of planning to software engineering research. In special we, propose a method to estimate the progress of an update process of source code. Supplemental to the method, we provide a first test to apply the method and do a primarily evaluation. Managers can use this method to control the progress of developers and plan updates.

All together, this contributes to the overall usefulness of the Hypermodelling technique by showing another application scenario. Furthermore, this planning scenario opens up a hole new research area to investigate Data Warehouse planning processes for their application within software engineering.

6. IDE implementation and Search Engine

   We advance the current state of the art in code search within the development environment by introducing Hypermodelling for the IDE. In contrast to prior research, Hypermodelling for the IDE uses well known views by developers and combines those with drag and drop composing of queries and result filtering. This allows developers to design queries for code slices and their combinations. We show evaluative use cases to underline the usefulness of the tool. This is a direct straightforward advancement for the IDE, because developers spend most time navigating code. With Hypermodelling for the IDE, they can speed up this process.

   Furthermore, we present a future IDE architecture, where Data Warehouse technology is integrated to allow advanced applications. Future research can now built advanced tools that utilize Data Warehouse technology within the IDE. In addition to the IDE architecture, we present a search engine based on Data Warehouse technology and proof the suitability of Data Warehouse based Hypermodelling implantation for this task. This code search engine is also an advancement to the current state of the art, because it respects quality indicators by code search. This is currently not supported by other code search engines.

7. Advancing Code Recommendation and Clone detection flexibility

   We contribute the area of code recommendation and clone detection by proposing to use Hypermodelling there. We show how Hypermodelling can be utilized for clone recommendation and live clone detection. Thereby, we advance the state of the art in code recommendation systems and clone detection by using a query based approach that can be adjusted easily. Furthermore, a Hypermodelling based approach can limit code recommendations to quality indicators what is currently not possible with comparable approaches.

   In addition the contributions in this area, this application also contributes to the Hypermodelling approach itself and underlines, again, its usefulness.

8. A program comprehension model based on psychological research

   Lastly, we contribute with a new program comprehension model, based on research of psychology to software engineering in the appendix. We apply Hypermodelling and do preliminary investigation of source code to find first supporting indicators for our theory. The investigation shows first support that mental categories play a role in programming. Besides this, the usage of Hypermodelling for a first verification shows, again, the benefit of Hypermodelling for code analysis.

   We induce a new and additional argument for Hypermodelling: Mental categories are not arranged with sharp boundaries and they are recognized depending on the current context. Additionally objects are members in multiple categories at the same time. Therefore, the multi-dimensional viewpoint of Hypermodelling is needed if one wants to respect the way mental categories are structured. This supports the need for Hypermodelling form another perspective.

Implementations and studies for all the improvements complement our work. We show that all of this contributions can be achieved and implemented with the Hypermodelling approach. Additionally, we compare our solutions to related work. At the same time, we do not argue against competitive approaches. We rather credit related approaches, because without prior work, we would lacked inspiration for application scenarios and to create Hypermodelling itself. Therefore, we see

our key contribution as integration of different approaches from the diverse areas. This makes it possible to address the different areas at once and enables further applications. Therefore, we wrap up our key contribution as follows:

Hypermodelling.
We propose to use multi-dimensional Data Warehouse technology as integrative solution for the different areas in software engineering research. This is what we call Hypermodelling. This contribution enables efficient software analysis and shows synergies and contributions to the current state of the art in diverse areas. We claim that our technique advances the current state of the art in software engineering and shows benefits compared to current approaches.

Hence, Hypermodelling is a contribution for upcoming solutions that addresses today's and future challenges. Furthermore, it utilizes Data Warehouse technology that is already applied within enterprises. The lines of code are growing. The integration capabilities Hypermodelling make it possible to reuse technology that is already there. Therefore, we consider Hypermodelling as advancement of software engineering research and practice.

1.9. Research Communication

This work provides clear information how to use Data Warehouses in software engineering. Thereby, we illuminate the technical perspective how object-oriented source code can be mapped to data cubes and present scenarios of the utilization of the technology. In order to communicate the research, different efforts have been done to notify it different audiences about our improvements. In addition to those, a website has been constructed and videos have been uploaded in the internet that show the technology in action. Thereby, the videos show different applications and give an expression about advanced and further use cases of our technique.

In another channel, we reached out to different communities to gain feedback for our work. In the following, we give a list of the associated publications that share material with this thesis:

1. T. Frey. Vorschlag Hypermodelling: Data Warehousing für Quelltext. In Proceedings of the 23rd GI Workshop on Foundations of Databases. Obergurgel, Austria. CEUR-WS. 2011.
2. T. Frey, V. Köppen, G. Saake. Hypermodelling - Introducing Multi-dimensional Concern Reverse Engineering. In Proceedings of the 2nd International ACM/GI Workshop on Digital Engineering (IWDE), Magdeburg, Germany, 2011.
3. T. Frey, M. Gelhausen, G. Saake. Categorization of Concerns – A Categorical Program Comprehension Model. In Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at the ACM Onward! and SPLASH Conferences. ACM. 2011.
4. T. Frey, M. Gelhausen. Strawberries are nuts. In Proceedings of CHASE '11. 4th international workshop on Cooperative and human aspects of software engineering. ACM. 2011.
5. T. Frey, M. Gelhausen, H. Sorgatz, V. Köppen. On the Role of Human Thought – Towards A Categorial Concern Comprehension. In Proceedings of the Workshop on Free Composition (FREECO) at the ACM Onward! and SPLASH Conferences. USA. ACM. 2011.
6. T. Frey, Hypermodelling for Drag and Drop Concern Queries. In Proceedings of Software Engineering 2010 (SE2012). Gesellschaft für Informatik (GI).Berlin, Germany. Springer. 2012.
7. T. Frey, V. Köppen. Exploring Software Variance with Hypermodelling - An exemplary approach. In 5. Arbeitstagung Programmiersprachen (ATPS'12). im Rahmen der Software Engineering 2012 (SE2012). Gesellschaft für Informatik (GI). Berlin, Germany. 2012.
8. T. Frey, V. Köppen. Hypermodelling Live - OLAP for Code Clone Recommendation. In Proceedings of Baltic DB & IS 2012. Tenth International Baltic Conference on Databases and Information Systems. CEUR-WS. 2012

9. T. Frey, Hypermodelling - A Data Warehouse Approach for Software Analysis. Magdeburger Informatik Tage (MIT). University of Magdeburg. 2012.
10. T. Frey, M. Gräf. Data-Warehouse-Infrastruktur zur Codeanalyse. In JavaSPEKTRUM 6/2012. SIGS DATACOM GmbH. 2012
11. T. Frey, M. Gräf. Hypermodelling Reporting: Towards Cockpits for Code Structure. In Proceedings of Theory and Practice of Computer Science - 39th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM). Springer. 2013.

1.10. Outline

In the following, we give a brief overview about the different parts and chapters of our work.

Introduction and Motivation. In Chapter 2, we introduce the general concepts and research that motivates, supports and represents the foundation of our work. We focus on the paradigm of separation of concerns and give examples where object-oriented programming does not meet the demands of the principle.

In Chapter 3, we get into the advancements of multi-dimensional separation of concerns and a view based concern separation technique. Complementing, we present the area of domain specific languages and show a modeling based approach that offers multi-dimensional navigation of a software system. Lastly, we introduce the area of source code analysis.

Chapter 4 leaves the area of software engineering and introduces Data Warehousing. We explain the different used data structures and give application examples how Data Warehouses handle multi-dimensional data.

Hypermodelling and Implementation. In Chapter 5, we tie the essence of the prior chapters together and derive Hypermodelling. This is the key contribution of our work. First, we give a brief insight how we utilize Data Warehouse technology to express concern associations of software. Based on this, we present a generic framework, how a Data Warehouse infrastructure can be used within software engineering. Now, researchers have a guideline howto apply Data Warehouse technology for source code.

Succeeding, we present a concrete technology application in Chapter 6. We propose and discuss specific data cubes, present source code samples in data cubes, reveal corresponding cube computations and visualize a holistic schema for source code. These manifold technical insights enable researchers to create solutions based on our approach and utilize our ideas in the future. Additionally, we get critical and illuminate and discuss the static technical properties that we revealed.

Evaluation and Applications. The following two chapters (Chapter 7 and Chapter 8) serve as a holistic step by step scenario how Hypermodelling can be applied for source code analysis. First, we show how various facts about inheritance and dependencies can be revealed with manual queries in Chapter 7. Thereby, we present various data and charts about the different kinds of inheritance and dependencies within a program. This investigation serves as motivation to determine if such scenarios can be glued into ready-to-use investigation dashboards. Therefore, we present Chapter 8. There, we present details about the effort to create an investigation dashboard for this case. Additionally, we show further advanced reporting scenarios.

In order to credit not "read-only" reporting scenarios, we present Chapter 9. In that, we provide the new scenario to do indicator based planning and progress measurement for source code. We evaluate our approach by an exemplary migration of a demo application. This indicates that the approach is feasible and underlines the need for further research in this area.

In Chapter 10, we get into Hypermodellings application in the development environment. We present drag and drop query tool for Eclipse and reveal different evaluative application scenarios. They indicate a usefulness of the tool. Furthermore, we provide a reference architecture for future development environments with active Data Warehouse technology. We evaluate the application of Data Warehouse components suitability within the IDE, by providing a realization of a code search

engine based on Hypermodelling. All in all, future development environments can now be enriched with Data Warehouse functionality.

We present an additional application scenario for the development environment in Chapter 11. Our scenario demonstrates how Hypermodelling can be applied for code clone detection and code recommendation. Additionally, we describe a concrete example to evaluate the feasibility of the approach.

Related Work and Conclusion. Finally, we leave the application scenarios of Hypermodelling and present related work in Chapter 12. Thereby, we present clear advancements of Hypermodelling in comparison to the motivating work. Additionally, we enlist all work that has similar goals like Hypermodelling and reveal plenty of potential synergies in future research. Complementing, we discuss downsides of Hypermodelling.

Lastly, we provide a conclusion and outlook (Chapter 13). We wrap up the contributions of this thesis and point to different future research areas. Finally, we give a concluding vision, inspired by our achievements.

Appendix. As addition to our work, we present an appendix. We use it to provide complementary information. Furthermore, we use the appendix to look beyond our own nose. Since we argument for the usefulness of Hypermodelling in multiple chapters and showed benefit in there, we ask ourselves how our approach relates to the human mind. Our question is, how the mind structures information and if there is research in psychology available that goes in the same direction like our approach. Therefore, we give an excursus about program comprehension out of research in psychology in Appendix A. We derive a program comprehension model based on this research. The mental category models let us conclude that a flexible approach, like Hypermodelling, is useful. Additionally, we use Hypermodelling to query for source code structure and to reveal first indications that categorization plays a role in programming.

2. Foundations

"I've come back for you, to remind you of something. Something you once knew: This world is not real."
Dominic Cobb. Inception. 2012

Abstract. Today developers have to meet competitive deadlines. At the same time, they face the challenge to achieve the best possible software quality. Commonly, developers encode functionality into modules to create a better maintainability and reuse of the artifacts of a program. Development and usage of modules is following the general paradigm of separation of concerns. Additionally, ready-to-use modules of so called frameworks are used to speed up the development time. Developers use commonly object-oriented development languages to tie framework functionality with own logic together to gain the desired functionality. In this chapter, we describe foundations about creating a software system and address subsequently shortcomings in object-oriented programming languages. Furthermore, we give an example how framework functionality can be weaved together with custom object-oriented modules. At the end, we present a summary of the key statements that we reflect in this chapter. Now, the following chapters can built on this statements and get into details about advancements.

2.1. Introduction

In the first chapter we explained that software systems often consist out of thousands of lines of code. In this chapter we dig into commonly used techniques and paradigms by developers. One of this chapters goals is to present main concepts and limitations in current object-oriented programming and to lie down the foundations for later chapters. However, we do not intend to give an historical overview and neither a completeness of related techniques. Our main goal is to give the reader an impression of our motivation.

2.1.1. Contribution

The contribution of this chapter is to focus this thesis on the paradigm of separation of concerns, object-oriented programming and frameworks and the associated challenges of developers.

2.1.2. Reading Guide

We start by describing a general viewpoint of how programs are created and of which artifacts they are composed of. Then, we focus on the paradigm of separation of concerns. Then, we give examples where object oriented programming does not meet the demands of the principle. Afterwards, we depict the role of frameworks within software development and formulate the demand that those need to be considered in software investigation scenarios. Lastly, we give a summary and draw conclusions.

2.2. Software Creation

Software creation is done by modeling parts of the real world. The term modeling has origins in logic [229, S.56]. In this work, we understand the term model as a notation that has a well-defined syntax and semantics [260,S.13]. In special, we refer in software engineering to models often as graphical representations. Those specify a part of a software system with a well-defined syntax and semantics. Especially the Unified Modelling language (UML) is widespread and offers different graphical representations of a software system [40, 8]. One main reason to use models is to gain a better overview over complex programs. Additionally, models are used to offer programming directly in a graphical form [207]. Different models offer different perspectives of a program and enable programmers to address the various viewpoints of a system directly. Nowadays, there also is a big movement in software engineering to ease the creation of graphical or textual models for specific

application domains [242, 56]. This kind of model is often called domain specific language and follows also the goal to ease creation of a desired set of software functionality within the domain. On an abstract level, domain specific models follow the goal to achieve a better comprehensibility and abstraction. They provide suitable and fitting representations for their corresponding development tasks. Hence, we credit the discipline of software engineering in special and define the term model as follows:

A software model, short model, is viewpoint of a part of a software system. Such models may be graphical or textual. The goal of a model is to provide a perspective that is suitable and fitting to a specific development task to reduce complexity.

However, within this work we use the area of modeling as inspiration and do not dig into advanced details. But still, we see it absolutely necessary to mention modeling explicitly, because one of a programmers key activities is to understand and model in the real world into a software system. Furthermore, we present related work that grounding on modeling techniques. Nonetheless, like, our definition says, one reason to use models is to reduce the complexity of modern software system development. At the level of source code, traditionally and nowadays, the complexity in programming is also faced with modules. Those modules are programming languages constructs that allow to encapsulating a certain set of functionality together. Modules origin the purpose to achieve a better maintainability and reuse of a program [205].

Literature refers to a module as a collection of algorithms and data structures to do a closed operation within itself. Using a module does not require knowledge about its way of operation. Furthermore, the correctness test can be achieved without knowledge about a certain programming system [216, S.569]. Others refer to concrete mechanisms in a programming language, like packages or Java classes [111, 180, 89]. We focus mainly on the Java language as proxy for other object-oriented programming languages. Therefore, we restrict and define term module as follows:

A module is a fixed set of functionality like a Java class or method. Such a module encapsulates a certain set of functionality that can be summoned or composed with other modules by programming language means. The purpose of a module is to enable reuse, comprehension, interchange and testing.

Hence, a module follows the purpose of encapsulation of a set of functionality. This enables reuse of this functionality and composition with other modules to derive more complex functionality. The ability to compose different modules together eases the interchange of modules. Furthermore, modules can be tested separately to ensure that their desired functionality works in the module itself.

2.3. Separation of Concerns

As mentioned before, software creation is done encapsulating functionality into modules. One main reason modules are used nowadays is to structure a program. In order to split a programs desired functionality modules the principle of separation of concerns serves as a guideline. It is used to create a software system with a better maintainability and to challenge complexity. The basic thought of this principle goes back to Edsger W. Dijkstra that described the approach as follows:

“Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained -- on the contrary! -- by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": It does not mean ignoring the other aspects, it is just doing justice

to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.” [80, S.2]

We see that the description is based on the abstract construct of the term concern. We interpret the term concern as follows:

“[...] we regard a concern generally to be any matter of interest in a software system.” [246]

Therefore, the term concern corresponds to any vantage point and any feature of a software system. Dijkstra refers to the generic idea to decompose a software system into its distinct viewpoints. For each of those viewpoints correctness on its own can be verified. When these vantage points are mixed up, we end up in a mess. But if we consider concerns them on their own, we gain the possibility to reduce complexity. This way, we can challenge and fight the chaos of the complete system and can at least verify the parts of it.

However, when we look into the detail of Dijkstras speech we can depict:

- [...] study in depth an aspect of one's subject matter in isolation for the sake of its own consistency[...]
- [...] which, even if not perfectly possible [...]

These excerpts show the restriction that Dijkstra argues for the need to study subjects on their own to keep them consistent. But on the contrary, he also mentions explicitly that a perfect separation of these matters may not be possible in all cases.

Programming paradigms offer mechanisms to decompose functionality physically into modules. This is often combined with the principle to separate concerns. Hence it seems logical to consider modules as mean to separate concerns. Therefore, programmers derive from Dijkstra principle that modules should not cover multiple vantage points of a system at the same time. Because of this, modules are often seen as concern encapsulation technique of a programming language [169].[1] Hence, programmers face the challenge to identify the concerns of a system, depict the right "modularization" technique and to create modules that cover a certain concern. Then, they compose all this "concerns-modules" together to gain a complete system.

However, here we recognize a difference to the general thought to study concerns on their own, because to "study concerns" addresses the view of those and this differs from their physical encapsulation into modules. Viewing and studying concerns is similar to encapsulate those, but it needs to be considered just as a similarity and not as equality. Thus, we see "module encapsulation" as physical manifestation and the study process as concern presentation for the subject that studies it. Physical encapsulation can help to create this views that are needed to study concerns, but we see it rather as two complementing approaches that advance each another. All together, we sum up the paradigm "separation of concerns" as follows:

Separation of Concerns, short SOC, is the approach to regard a software system out of separate vantage points. The desired goal is to study the different concerns of a system on their own. Often, this goal is tried to be achieved by encapsulating concerns into their own modules. A software system is created by the composition of these different modules.

We credit that the application of the separation of concerns into modules is sometimes not perfectly possible. Thus, separation of concerns is a desired goal that in some cases may never be achieved. We refer to the main thought of the principles innovator Dijkstra that concerns should be studied on their own. Hence, we distinguish studying concerns explicitly from the physical

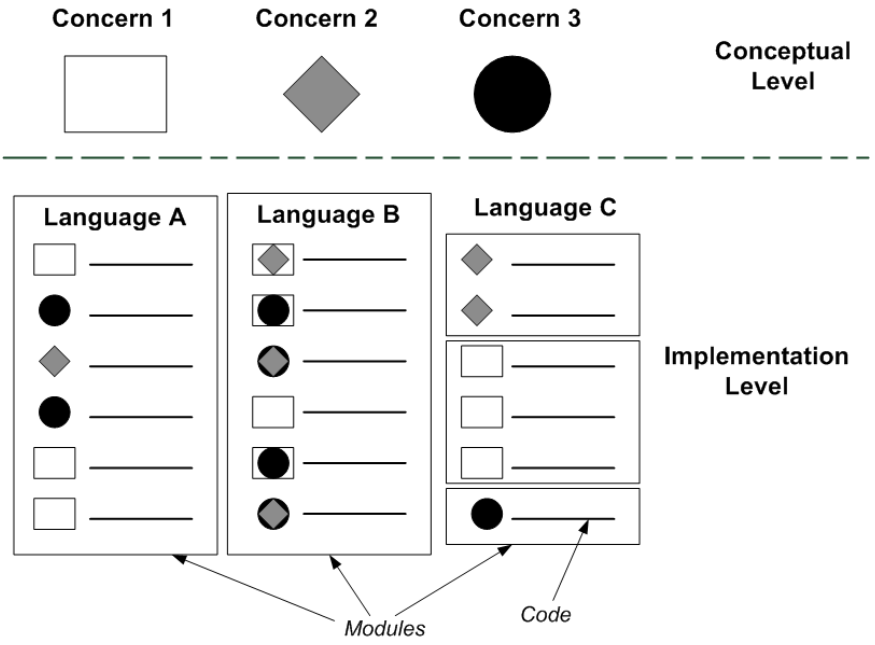
[1] See the relevant chapter about the "The Single Responsibility Principle" online at http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod - 7.11.2012

module encapsulation. Therefore, separation as concerns is referring to a presentations of a concerns. Modules of a system act as one physical mean to support to study the concerns on their own, but cannot be considered as the only way to separate concerns since the separation is originally refereed to be presentation based.

2.4. Limitations in Object-Oriented Programming

Right before, we described that separation of concerns is a desirable goal. Today, the principle is commonly applied by encapsulating concerns into modules. Therefore, programmers try to encode one concern in each module. However, the possibility to apply separation of concerns by module encapsulation it not only based on the skills of programmer and as well depending on the modularization capabilities of a programming language. Hence, if concerns got identified they need to be transformed into modules and suitable modularization techniques have to be at hand. We show a schematic overview of the possibilities to map the concern space onto the programming space in Figure 2.1. The figure shows the conceptional concern space and the lastly resulting implementation with programming languages that offer different mighty modularization capabilities. In the following, we describe the figure in more detail.

Figure 2.1. Mapping Concern Space to Program Modules



See [127, S.4] for the similar and modified picture of origin.

Language A allows no modularization at all; all concerns are assembled together in one large module. The concerns are totally intertwined and not separated. Language B offers more capabilities, but still, one module contains multiple concerns at the same time. The responsibilities of a module for multiple concerns make it difficult to reuse them in a different context, because another functionality is coming as addition, whether wanted or not. Likewise, a concern is scattered over various modules at the same time. Hence, language B also violates the principle to encode and edit concerns on their own, since the concerns are not separated perfectly. This leads to the problem that an update of the logic of this concern results in multiple changes in different modules. This kind of language often appears today and we remark two terms that express this kind of concern interweaving occurring in language B as follows:

Tangling means that a module is responsible for multiple concerns at the same time. Hence we cannot identify a single and primary responsibility of a module.

> Scattering means that a concern is distributed over various modules at the same time. Hence, a similar coding that is a part of a concern can be found in the different modules. [149]

Language C is more "mighty". Here, programmers are enabled to map their concern space directly to modules. This reduces complexity of the modules and avoids tangling and scattering [127].

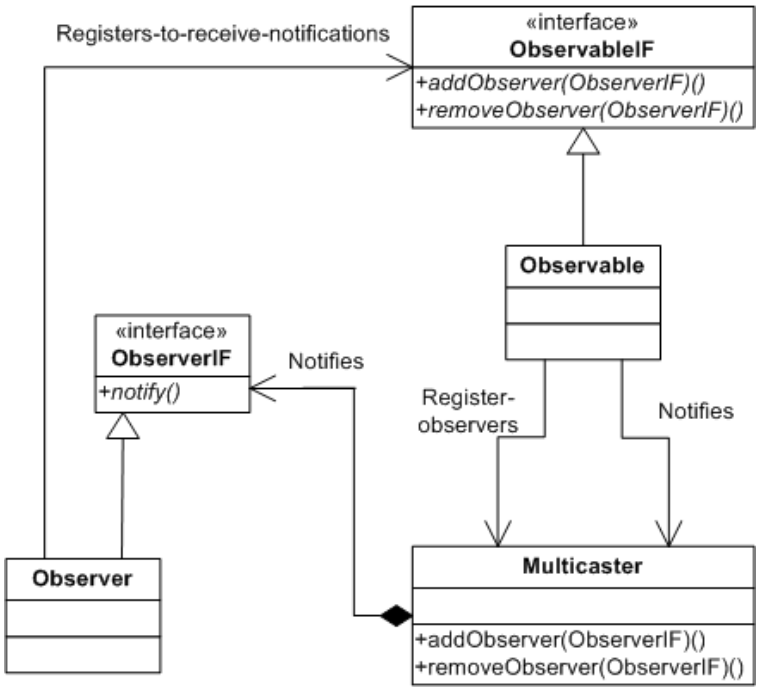
In the following we pick out examples, to demonstrate that object-oriented programming is similar to the presented Language B. We do this to make the limitations more visual then a pure reference to literature would be. In object-oriented programming, it is not possible to separate the conceptional concern space directly onto modules in the use of widespread object-oriented means. In order to provide a sound example of some limitations, we present two design patterns and describe problems of a not perfect separation concerns by using those. We chose to present design patterns out of the reason that they represent good and accepted architectures by developers and still lack to follow the paradigm to separate concerns.

Design patters are arrangement proposals for classes methods and other object-oriented fragments that were developed to enhance the reuse of parts of object-oriented software systems. Details can be found in [106].

We know that someone could come up with object-oriented workarounds to overcome the problems we describe. Therefore, we are careful not to state our samples as absolute and just refer to their main meaning of better comprehension.

2.4.1. Example: Observer

Figure 2.2. Observer Pattern



See [110, S.389] for the similar graphic of origin.

The Observer Pattern is used to distribute change notifications of traced objects.[2] An example is to notify the user interface to reload, in case of an internal data model update. We present the general

[2]For details about the pattern see [110].

approach of the observer pattern in Figure 2.2. There, every class (Observable) that is supposed to be observed for status updates implements the ObservableIF superinterface. This way, it is ensured that the class contains the means of functionality to register or de-register an Observer. The shown methods are adding this Observer to their Multicaster object.[3] In case a status update happens the Multicaster object get notified and messages the corresponding Observers.

The observed object (Observable) contains code to register observers. Every (Observable) object contains code aside its normal and primary functionality. This violates the principle of SOC though modules, because the module has multiple responsibilities. First the class contains its own functionality and additionally the functionality that makes it observable.

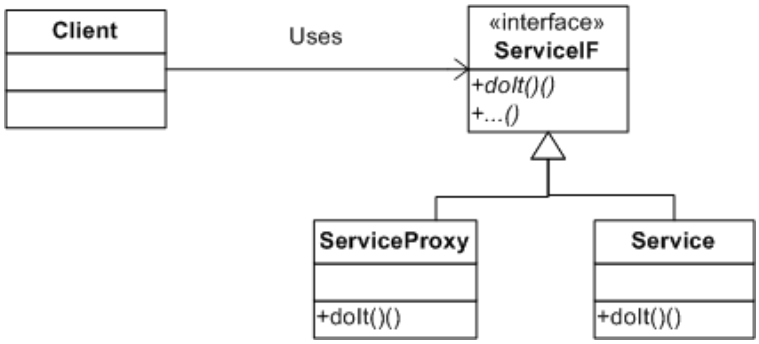
2.4.2. Example: Proxy

Another sample for a not clear applied SOC in modularization is the proxy pattern. It serves the purpose to control or alter the access to an object or a class. A proxy is installed in front of the original class or object. Accessors call the proxy instead of the original. The proxy adds logic before the method execution and then executes the original method. Typically, this pattern is used to restrict access control by checking the identity of the caller before the original object is called. We present a diagram of the pattern in Figure 2.3. There, the client Client that consumes originally the (Service) class object is programmed against an interface of the original object. The class (ServiceProxy) implements the interface as well as the original (Service). Thus, the client can work with the (ServiceProxy) or the original (Service) object.

Just imagine you want to secure an application with the proxy pattern. Thereby every class will need to be wired together with a proxy class. If the security logic differs for different classes, multiple proxy classes are needed, containing similar security logic. This replication of security logic reminds to the term of scattering that was introduced before: The same functionality is encoded in multiple places.

We note that the there are facilities in the Java Enterprise Edition (Java EE) available that allow to secure applications in a generic way. However, we do not get into it right here, because we pick out examples that reflect easy to understand challenges in classical object oriented programs. In the next chapter, we show also an advanced security example using aspect-oriented programming that works similar like using the facilities of a Java Enterprise Edition. For further information about securing enterprise applications and advanced facilities see the Java EE documentation [18].

Figure 2.3. Proxy Pattern



See [110, S.93] for the similar graphic of origin.

2.4.3. Implications

All together, our two examples gave impressions that object-oriented programming is not perfectly suitable to separate all concerns. Code replication is needed to solve the problems and the modules

[3]The reason to do so is to enable the possibility to use one observer for different at the same time.

have more than one responsibility. This means that object-oriented programming is similar to Language B that was described in the beginning of this section. This is supported by research that comes up with new solutions and indications that object-oriented programming lacks advanced modularization techniques [32, 87]. Also, researchers describe formally that a concern - module association mismatch exists [220]. Some even argue that a pure hierarchical decomposition, like it is the case in normal object-oriented programming, does not align with multi-dimensional human organization of knowledge in general [199]. Therefore, our samples provided only a supportive perspective for a whole research area that focuses on advancements of object-oriented programming.

2.5. Frameworks

Even though, object-oriented languages have some limitations, they are widely applied and plenty of pre-encoded functionality exists. Those ready-to-use code structures are used and weaved together with custom functionality by programmers to achieve the desired needs. Commonly, large object-oriented and ready to use structures that offer a set of functionality are called frameworks. Hardly related to the term Framework is also the term library and component and they are hard to distinguish and overlap sometimes.[4] Hence, literature offers different definitions and abstractions for the term of a framework [84, 70, 249, S. 425 211,S.2]. Some describe it rather abstract as usable architectures [249], while others focus on the composition of components [33]. Since this works focus is on object-oriented technology, we neglect the discussion of what is a framework, what is a component and what is a library and stick to the definition in the object-oriented context:

> "Frameworks are usually large O-O structures that can be tailored for specific applications, provide infrastructure and flexibility for deploying O-O technology, and enable reuse at a larger granularity. They do not have to be O-O structures, although they are commonly implemented this way. The design of the Framework fixes certain roles and responsibilities among the components, as well as specifying standard protocols for the communication and collaboration between them." [132, S.343]

This means, a Framework offers a (half) ready application frame that developers can use to embed their own application logic in. Commonly, such ready to use functionality is offered in Java as jar files that can be added to an application and the functionality within them can be used.[5] This usage is done through modularization techniques. For example, framework classes can be instantiated as objects, methods called or parent classes of the framework get inherited. In order to explain how developers embed their applications within a framework, we show a sample with the popular spring framework[6] in the following.

Spring is a large open source application framework that is developed to ease the development of Java Enterprise applications [130, 22]. Because of the immense size of the spring framework, we depict a sample application that uses the Model View Controller (MVC) pattern that is offered by the spring framework. Goal of the MVC Pattern is to ease application development and to advance the reuse capabilities of an application [164]. Like other components of the framework the MVC patters utilizes the inversion of control container of spring. Inversion of control is a design pattern that enables to develop applications where the components are loosely coupled and then weaved together via configuration settings. In general, the pattern specifies that all classes should be programmed against interfaces. Then developers can create concrete implementation of those interfaces. In the application configuration it is then specified which class are instantiated and how they are woven together. This way, custom code of a specific application can be easily woven together with framework code [130]. In case of the MVC pattern, developers extend and utilize the ready to use classes and interfaces and then embed the own encoded logic via configuration.

[4] See for instance the definition of the term component in [81, S.58].

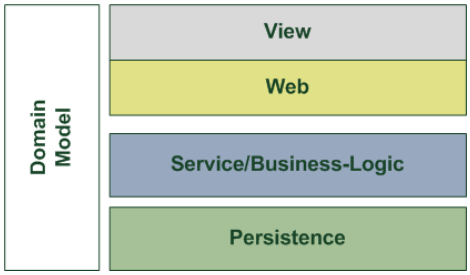
[5] Again, we see that libraries and frameworks are hard to distinguish, since jar files are also named libraries. Anyway, the most important thing is the usage of functionality. We spare the discussion here, since it has no impact for our work.

[6] We chose the spring framework through its wide acceptance within the Java Enterprise community. The homepage is available at http://www.springsource.org/ - 7.11.2012

For this work, it would be too detailed to explain the general thought how modern applications in enterprises should be developed with the spring framework and how the integration of custom code and framework code works. Therefore, we explain a use case on an abstract level and refer to literature about the principal design patterns and approaches in enterprise software development [130, 93]. Thereby, we show the generic thought that framework code is woven together with specific application code. In order to provide the possibility to look into details of the application, we refer an example to a step by step tutorial for this application is available in [217]. In the following, we first explain the spring MVC pattern and the layers that are defined by it. Then, we describe with a class diagram how code of the specific application integrates into that.

We show the diverse layers of an MVC application in Figure 2.4. The View presents the application data and is the interface for human interactions and integrates into an application by being coupled with the web layer. We discuss the other layers in the following.

Figure 2.4. Spring MVC Layer



See also a similar graphic in [147,S.22]

Web. The Web layer contains controllers that are used by the user interface. Such controllers offer functionality that is responsible to steer the application user interaction. The controllers react on the different submitted user entries and represent the actual state of the application of the server. For instance, controllers store which data was lastly submitted or submit redirection URLs to the View. The Controllers also invoke the business logic in the back end. Also, they can modify the Domain model and offer its content to the views. In order to support programmers, spring offers a ready to use controllers that can be adapted for a specific application. Their logic can be used by annotating source code or inheriting from ready controllers class modules.

We note that which controllers are offered in the spring framework and how they are wired into the own application depends on the spring version. Today, it seems that a general trend is away from inheritance towards annotations.

Service. The Service/Business-Logic layer is responsible to realize the business logic of the application. This means the internal logic how the application works and how it operates is defined within this layer. An often taken approach is to encapsulate different sets of functionality as services. In order to expose these services, the spring framework offers means like annotations or super interfaces that can be used to indicate that the logic is belonging to the service layer. Controllers access these services to tie the web layer to the internal business logic.

Persistence. In order to realize permanent data storage a Persistence layer is used. Developers encode where and which information is stored within a database. Spring eases here the access to the diverse persistence technologies. For example, templates ease the access to "hibernate", "toplink" or the "Java persistence API". This way, developers do not need to encode all of the logic that is commonly used when accessing those technologies. Furthermore, the diverse technologies are abstracted to a common ground to ease interchangeability. The service layer is programmed to work on top of these abstract constructs that it can be plugged together.

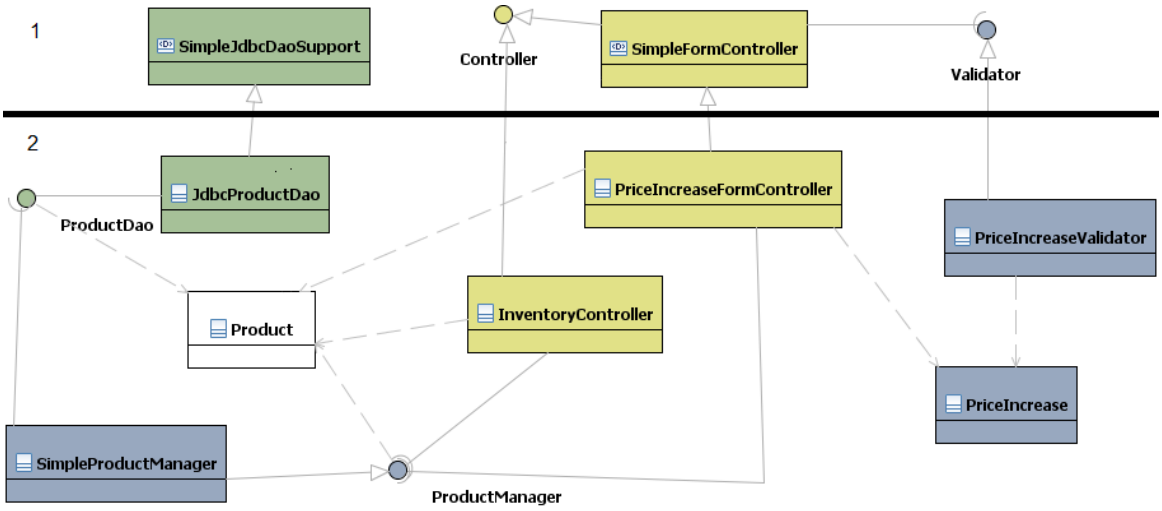
Domain Model. The Domain Model is arranged vertically. It is the "data heart" of an application and specifies the persistent model of this application domain. The domain model contains the data structure that is used throughout the application. The view presents the data to the user, the controller

ensures the validity and integrity of the data and how it may be browsed by the view, the service layer contains the business logic that is applied to the data and the persistence layer defines how and where the data is stored.

Application. Now, we describe how a demo application integrates into the diverse parts of the spring framework that help to encode the application logic within this layers and to associate it with framework logic. Our example is a simple Product management system. It can show products and raise the price of those.

We show parts of the application in Figure 2.5. In the following, we describe the diverse layers and how the integrate into the framework.

Figure 2.5. Spring MVC UML Example



We created a UML class diagram based on a projects code excerpt of the example described in [217]. On top (1), we show the modules out of the Framework and at the bottom (2) we show the implementation of the application to add its own logic. The different colors depict the association of the diverse components with the various layers. Those colors correlate with the ones in Figure 2.4.

A part of Figure 2.5 shows the domain model. The implementation of this domain model is the Product class. The ProductManager, SimpleProductManager, Validator, PriceIncreaseValidator and PriceIncrease are part of the business logic. The ProductManager offers an interface that is implemented by the SimpleProductManager. The SimpleProductManager class is programmed against an interface (ProductDao) that defines the main method of the persistence logic. This interface offers functions for saving or retrieving object from a persistent storage. Another class of the business logic is the PriceIncrease class that defines the amount of the price raise of a Product. The PriceIncreaseValidator is inheriting the generic Validator interface from the framework. The PriceIncreaseValidator contains logic that a price increase can only be done within a certain range. ProductDao offers the definition how the persistence logic can be accessed. JdbcProductDao implements this interface and inherits also from the template for database access (SimpleJdbcDaoSupport) of the spring framework. This way, the implementation of the defined interface logic is done by using pre-defined code of the framework. In order to realize the controllers and to embed the application into the framework code the controller of the application (InventoryController, PriceIncreaseFormController) inherits form framework classes and interfaces (the Controller interface and the SimpleFormController class). We can see that the controllers of the application are programmed against the business logic of the application by using the diverse interfaces of this layer.

However, the final binding into the defined application frame of the spring framework and the coupling of concrete class instances instead of interfaces of the application is done via configuration. The configuration specifies which concrete components work together. Therefore,

the spring framework offers a domain-specific language that allows this kind of configuration. All together, we can see that many types of the application are derived or stand in association with framework elements. For instance, the complete web layer is derived from framework elements. The JdbcProductDao class of the Persistence layer is inheriting form a framework class and relates this persistence layer of the framework. Furthermore, the string "Dao" in the name indicates the usage of the Data Access Object design pattern that is commonly used for database access. In case of the domain objects (e.g. Product), not all of those domain objects are associated with framework classes. Anyhow, their meaning could be inferred out of the general application structure and associations: The domain objects are accessed by all other layers, since it is arranged vertically to the other layers.

In general, the whole example shows that inferences with each layer of the framework can be done. The main meaning why we presented this example is to give impressions how an application integrates into framework logic. Programmers know how a framework operates and can derive which part of the own logic is associated with which layer and know the meaning of those layers. Therefore, frameworks offer a meta- structure and get, aside from the usage of ready-to-use functionality, used to structure the own application logic.

Implications. All together, we described that frameworks define ready to use logic and a specific application can be embedded into it. Developers use such frameworks commonly to built applications and know the classes and interfaces that are offered by a framework. An application integrates into the diverse layers of the framework and through the known framework architecture inferences to which layers application logic belongs can be built. For the further work we note the following:

Applications are commonly using frameworks. Developers know the classes and interfaces of a framework and embed their own application logic into it. They do this by encoding their own logic into modules. Then they associate this modules with framework functionality. This association is finally done by programming language modularization, composition and configuration means.

2.6. Summary and Conclusions

We presented diverse challenges and techniques in software engineering. Thereby, we described that often real world modeling is used to create a program. Also, it was pointed out that modularization is one mean to encapsulate a certain set of functionality within source code. The principle to apply separation of concerns was presented as achievable goal to handle complexity in the development process. This principle recommends study different vantage points of a software system on their own to keep the concerns on their own consistent. However, we gave examples that object- oriented programming does not allow to apply the principle with modularization in all cases and referred to related literature that reports similar problems. Lastly, we worked into an example how frameworks are used to wire predefined logic framework together with application-logic.

In the following, we will recap the essence about the diverse shown techniques in this chapter that our work references later. We do this, by formulating short claims of what we see as notable of the diverse techniques that should be respected in order to derive a new technique that helps to advance software engineering.

First, we see the paradigm to separate concerns as a key concept within modern software manufacturing. We credit this in our following work by referring to the following wrap-up statement as demand on a new enhancing concern related technique:

Different viewpoints should be supported to study concerns of a software from their perspective.

Therefore, our statement addresses the need that a new technique respects the main meaning of separation of concerns to study concerns separated. However, pure object-oriented techniques

are widely applied whereby concerns cannot be separated "perfectly" through the lack of modularization. Therefore, we credit that often concerns get not separated totally and programmers have a hard time to apply the concern modularization. Thus, in reality often concerns get intertwined within the same modules and no perfect separation exists. Therefore, we see the necessity to credit this reality in our work as we define in the following:

A new technique should respect the interweaving of concerns in source code and the current limitations in programming.

Anyhow, if separated or not, industry programmers use frameworks to speed up the development process and to advance reuse of common functionality. One mean to have ready-to-use software artifacts at hand are frameworks. Those offer plenty of functionalities that correspond to common concerns like different application layers, data access and similar. Frameworks are widely applied and the knowledge about their usage is considered to be one success factor for a developer. Hence, developers know the functionality of the frameworks and specialize in their use. Therefore, software development and code studies are based to a certain amount on the knowledge about frameworks. We consider that this reality cannot be neglected and neither be marked down. Therefore, we mark the following:

A new technique should also be capable to consider frameworks as a perspective for software.

However, our entire here considered claims address different challenges and improvements of the development process beside the immense size of code bases that was mentioned in the first chapter. Programmers face this complexity every day and have the difficult task to apply separation of concerns with limited modularization means within the object-oriented programming languages. Beside this, they have to focus on reusing framework functionality through modularization means or to create frameworks themselves.

We recognize that the different claims target different areas and features in software development process. Researchers try to split those challenges apart. One group of researchers focus on software component development and others focus on new means of modularization. Others try to investigate code bases to learn from what programmers did and if it was good or bad. Another group tries to shift complexity to the modeling space. However, all those research areas have challenges on their own and plenty of publications exist that cover deep aspects in each area. Therefore, we follow up in a next chapter on techniques that excel within the different research groups.

3. State of the Art

"You take the blue pill and the story ends. You wake in your bed and believe whatever you want to believe. You take the red pill and you stay in Wonderland and I show you how deep the rabbit-hole goes.
-- Remember -- all I am offering is the truth, nothing more."
Morpheus. Matrix. 1999

Abstract. Object-oriented programming faces several limitations and plenty of challenges. Different approaches advance or supersede the approach. Additionally, researchers dig into the analysis of programs to reveal insights about program structure. Others create holistic models and advanced environments that enable model based navigation and creation of programs. In this chapter, we present some of the approaches. We show advanced mechanisms to separate concerns, get into details about domain specific languages, present an approach to navigate with models and describe goals of program analysis. Lastly, we give a summary and point out different key contributions and gaps of the diverse approaches. Now, different areas and their contributions are assembled together at one place to serve as motivation for our own technique.

3.1. Introduction

In the prior chapters, we described the challenge of large scale code bases and presented limitations of today's widely applied object-oriented programming. Furthermore, we explained the paradigm to separate concerns. In the current chapter, we take these thoughts to use and show different areas that deal with diverse challenges in software engineering. We aim to transfer the key concepts and in current techniques to credit inspirational work that served us as motivation to develop a new technique. Hence, we neither intend to give a historical overview nor to give a complete collection of related techniques. Our main goal is to give the reader an impression of our motivation. Therefore, we reference further work to classify and associate our own work with the motivating areas of origin.

3.1.1. Contribution

The contribution of this chapter is to introduce different approaches that enhance programming. All introduced approaches are developed in more or less connected scientific communities within software engineering. We point out different key contributions and assemble them together as requirements on a new technique. Furthermore, we show that the different areas contribute different and loosely coupled artifacts that lack integration. Hence, this chapter serves as both; - as motivation for our own work and also as summary of different streams within software engineering research.

3.1.2. Reading Guide

We start by focusing on multi-dimensional separation of concerns and advanced modularization techniques. Then, we lift our perspective from the pure code and focus on view based concern separation. This following, we present the area of domain specific languages. Succeeding, we focus on a model based approach that uses models for navigation and as viewpoints of a software system. Lastly, we go back to source code and introduce the area of source code analysis. Then, we give a summary and point out the key facts of this chapter.

3.2. Multi-Dimensional Separation of Concerns

In this section, we present the advancements of multi-dimensional programming paradigms. First, we focus on subject-oriented programming that led to the so called Hyperspace approach. We focus on these approaches to credit the original thoughts about multi-dimensional separation of concerns and its origins. Then, we get into aspect-oriented programming and use it to provide examples how multi-dimensional approaches can be applied. Afterwards, we give references to further work

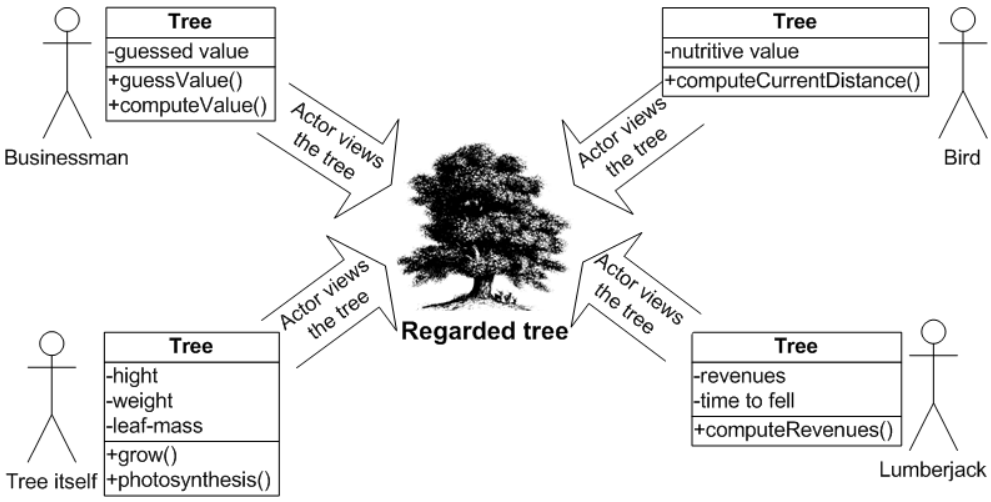
in these areas. Lastly, we draw conclusions about the key contributions of the multi-dimensional approaches.

3.2.1. Hyperspaces

An approach that advances object-oriented programming is subject-oriented programming [117 , 60]. Subject-oriented programming breaks with the single viewpoint towards the classes and objects of a software system and introduces the possibility to have different viewpoints of concerns. The main motivation for this technique is that object-oriented models do not express the reality of different perceptions. Furthermore, if a new viewpoint is added to an object-oriented system, often the whole design needs to be adapted [65]. Therefore, subject-oriented programming proposes a decentralized class definition [197].

We present the approach to have multiple vantage points of class definitions in Figure 3.1. There, each actor refers to different properties of the tree. For the bird, the tree has a nutritive value. For the lumberjack it is the effort to fell it. For the businessman it has economic value. Every actor perceives the tree differently. And even the attributes of the tree differ for each actor. Therefore, it is logical to speak of the different perception subjects. However, note that these subjects don't need to be bound to individuals and can also reflect a group of those.

Figure 3.1. Different Perceptions in Subject-Oriented Programming



Different actors regard a tree. Each subject recognizes properties of the tree differently. This is visualized through multiple class visualizations. A similar graphic can be found in [117] and [129].

Finally, subject-oriented programming approach was superseded and implemented through the Hyperspace approach [6].[1]Hyperspaces follow the goal to enable programmers to modularize all concerns and to realize arrangements and relations of classes similar to the tree picture before. Hyperspaces address the tyranny of the object dimension of object-oriented languages as main mean for decomposition and propose advanced modularization. Therefore, Hyperspaces allow to built modules for each concern and to compose this modules together [198].

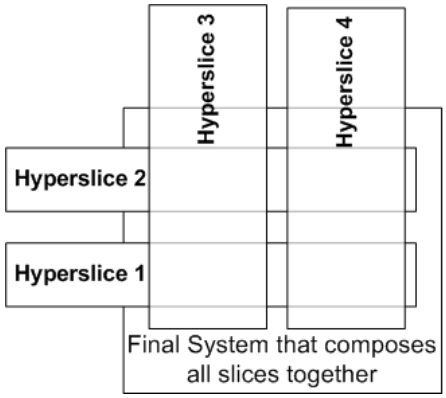
Through the advanced modularization and composition techniques, the Hyperspaces introduce a multi-dimensional view on software, consisting out of so-called Hyperslices. Hyperslices are excerpts of the source code that represent the same concern. Thus, a Hyperslice groups one concern together. The different Hyperslices interact with each other and get composed through compositions. This compositions ensure the consistency between the different viewpoints [250]. The possibility to regard every concern on its own is also called Multi-Dimensional Separation of Concerns (MDSOC).

We visualize the multi-dimensional concern space with the Hyperslices in Figure 3.2. There, a software system is consisting of different Hyperslices that overlap each other. This shows that

[1]http://www.research.ibm.com/hyperspace/ 6.11.2012

the different slices interact and get composed together to derive the final functionality of the software system. The different slices can be compared to the different viewpoints in subject-oriented programming and the compositions of the slices makes the interaction between the different viewpoints finally possible.

Figure 3.2. Sample of Hyperslices



3.2.2. Aspect-oriented Programming

Before, we describe the Hyperspaces that introduced the idea to slice a program into concerns spaces. Similar to the Hyperspaces is Aspect-oriented programming (AOP). AOP also follows the goal to enable an advanced separation of concerns into modules.

Note, that we spare the discussion which techniques exactly are covered by the term AOP and MDSOC and which not, since the focus of this work lies in the general multi-dimensional viewpoint towards software. In our work, we refer to Aspect-oriented programming as described in [142] and the AspectJ implementation for Java [140, 165, 72, 15, 16, 182].

Therefore, like the Hyperspaces, AOP was developed to overcome the current limitations of tangling and scattering through advanced modularization techniques. AOP introduces the terms of Pointcuts, Join points and Advices to realize this kind of modularization. AOP regards a program as a set of Joinpoints. Examples for such Joinpoints are the functions, variable or class access, class creation or statements within a method. Just anything that can be defined by the structure of a program can be taken as Joinpoint. Pointcuts represent and select a specific set of Joinpoints by using a specification language. Such a Pointcut specification language can be regular expressions or similar means that specify which structures within a program correspond to a Pointcut. Just imagine to select all classes of a package org.sample that contain the string 'test' within their name. Every access of these test methods belongs then to the specified Pointcut.

Note, it is also possible to use specific instances of classes (objects) as Joinpoints and define the Pointcuts for those. We drop this issues and do not discuss it in detail in this work. Here stick to the main idea of multi-dimensional separation of concerns and refer to related literature, [140, 165, 16], for detailed and further explanations.

In order to weave in additional functionality to Pointcut, AOP uses the Advices. Advices contain code that is executed before and after the code that is following the Pointcut is executed [121 ,S.154]. So, putting all of this together, we call it an Aspect. We simplify the definition of an Aspect as follows:

An Aspect is a collection of multiple Pointcuts and Advices. Pointcuts specify a set of structures of a program that correspond to a defined pattern. Advices are associated with Pointcuts to be executed when the program invokes the defined structure to which the

Pointcut corresponds. Therefore, if a code structure that is specified through a Pointcut gets executed, the code of the Advice is executed before or after it. All together, the structure of Pointcuts and Advices is grouped together into a class similar construct called Aspect.

We note that it can be distinguished between different Advice types, like "before", "after" and "after returning", what refers to program execution logic. However, we do not get into this kind of specific details here.

Challenge Logging

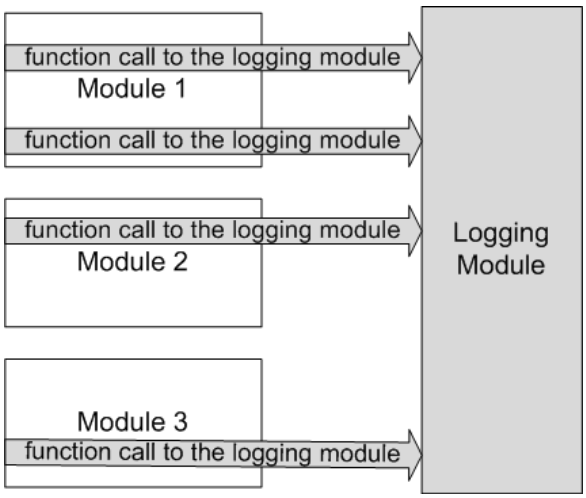
However, the simplest way to explain the benefit of AOP is by a logger example. We imagine the following Java source code:

Example 3.1. Simple Logging Code

```
public class CustomerDAO{
...
    public void saveCustomer(Customer cust){
        -...
        if(logenabled==true)
        logger.log("Database Acces", "SaveCustomerCalled")-;
        this.session.save(cust)-;
... -} -}
```

Example 3.1 shows that every module, wanting to log something needs to call logger functions. When the API of the logger is altered the invoking class, CustomerDAO, needs to be altered as well. Considerations about the primary responsibility of the module lead to the functionality to store a Customer object in the database. Therefore we see that the functionality of logging is tangled into this module. We abstract this problem to multiple modules in Figure 3.3. There we can see that different modules access the logging functionality. All this modules have to use the code and are dependent on the module.

Figure 3.3. Usage of a Logger

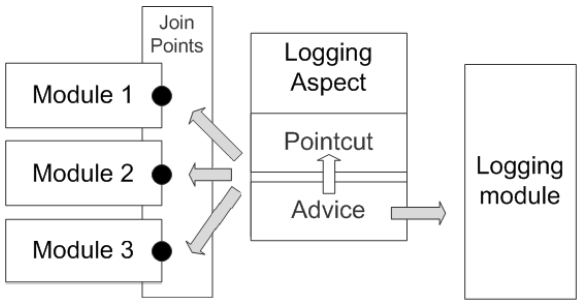


Different modules log their current state by calling the logger module. The concern of logging is therefore distributed in different modules at the same time. A similar visualization can be found in [165 , S.12]

Solution Aspect

We present the solution of the previous described logging in Figure 3.4. There, we see that the modules do not have their own logging code any more. They are "adviced" by an logging Aspect module. We show the Pointcuts as the black dots where the Advice gets finally applied.

Figure 3.4. Logging with Aspects



The different modules get additional logging functionality through an Aspect. The Aspects Advice contains the logic to access the logging module. This logic is weaved in by a Pointcut specification that defines where the Advices functionality is applied. A similar graphic can be found in [165 , S.13]

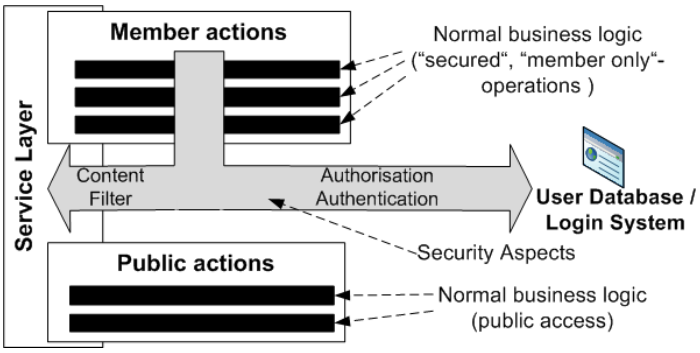
The Aspect community calls these concerns, like logging, that can be realized through Aspects, Cross-cutting concerns [148, 82]. Cross-cutting concerns express that concerns are arranged vertically to the well known concerns that are encoded via traditional object-oriented means. If Cross-cutting concerns manifest by pure object-oriented means, they excel through tangling and scattering, like it is the case for logging. Therefore, Aspects are mentioned as beneficial because they enable the decomposition of scattered and tangled concerns into their own modules [165,S.10, S.21].

Therefore, Pointcuts and Advices weave additional functionality into a program. In the following, we show an additional and often named example of an Aspects application to give a better impression about other applications scenarios and the general ideas behind AOP.

Security by Aspect

Another typical example, besides logging, for a hard to modularize concern, is security. Security is often weaved into many classes and methods and represents an additional concern to the primary logic of a module. Every method determines itself by if access is allowed, denied or which data may be returned to the caller. However, securing applications can also be done by Aspect, what we show in Figure 3.5 and describe in the following.

Figure 3.5. Security by Aspect



Different methods of an application are accessible from outside. Therefore, security audits have to be done and control which data gets transferred to the outside and which methods are allowed to be invoked. Aspects are applied to ensure that the corresponding security policies are respected.

In Figure 3.5, different methods are accessed as horizontal members of the service layer. Some are open for any access without security (public actions). Others are secured for access or the retrieved content gets filtered (member actions). This filter functionality and access restriction is

done by Aspects. Aspects Advices define what is filtered and who has access. Pointcuts are used to specify the secured methods and classes where the common logic execution has to be intercepted and security logic has to be weaved in. Out of this reason we show the Aspects vertically to the actions that are secured.

3.2.3. Further Reading

We consider advanced modularization techniques as a key inspiration for our own work. Thus, we refer to related literature to credit the work of others and also associate our own and later presented Hypermodelling technique with this area.

An overview about different programming and modularization techniques is given in [87]. An alternative approach, called composition filters gets described in [17]. Detailed information an AOP language for Java can be found in [165]. A detailed description about AOP development within the Eclipse development environment can be found in [16, 72]. However, in the previous chapter we used design patterns to explain some shortcomings of object-oriented programming. In relation to this, further information how to advance traditional design patterns with Aspect technology can be found in [115]. However, Aspects advance programming language constructs. Therefore, we also refer to a publication that gets into insights about the combination AOP with various modularization techniques that come with the Java and a widely applied application framework in [141].

3.2.4. Conclusion

All together, we showed that there are modularization techniques available that allow advanced separation of concerns into new kinds of modules. These techniques help to credit different perspectives within programming and also to avoid tangling and scattering. However, currently object-oriented programming is still the dominant and widespread paradigm. Therefore, concerns are traditionally intertwined within object-oriented modules. Therefore, we note the following conclusion:

In object-oriented programming a module often is associated with multiple concerns at the same time and it is not possible to regard every concern on its own. Concerns represent a multi-dimensional space that can be expressed by multi-dimensional modularization techniques, like Hyperspaces or Aspect-oriented programming. Those techniques offer advanced modularization techniques to encapsulate concerns in more ways than "traditional" object-oriented programs.

However, when we compare the modularization techniques and Dijkstra's original recommendation to "regard every concern on its own", we see that the advanced modules encode concerns at a physical level. This way, the techniques target the direct encapsulation and leave the challenges to study the complex concern space for other researchers. Nevertheless, left to others and physical or not, the different techniques show that the concern space of a program is multi-dimensional and the perspective on pure classes cannot cover all the concerns of a program. In order to regard a program out of the perspective of concerns, different and additional perspectives, aside of objects, are needed. Such perspectives are probably multi-dimensional, since the physical module encapsulation techniques are as well.

All together, the physical encapsulation means that the programs have to be adapted to Aspects or similar techniques. However, some researchers favor only a certain level of separation into modules and complement the module separation approaches with a view based separation. We present one such approach in the next section.

3.3. Virtual Separation of Concerns

Similarly, to the module encapsulation of concerns that was discussed before, we focus now on an additive or complementary approach from another area. Virtual separation of concerns is founded

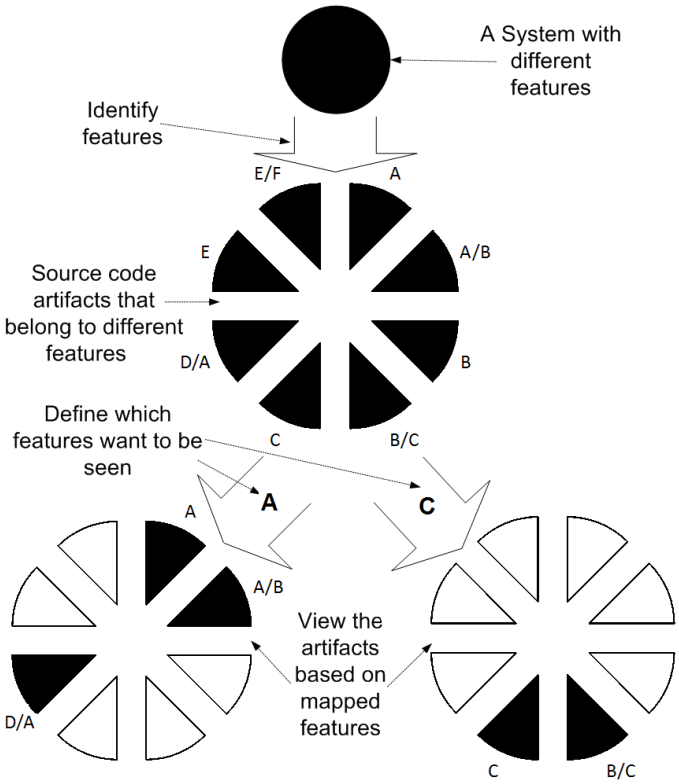
on the ideas of feature oriented software development. Feature oriented software development deals with the creation of software product families. Software product families are programs that are applied with different configurations and different features. For instance, a Linux kernel can be applied to routers, smartphones, servers notebooks and so on. The source code of the kernel is compiled for the different devices and different features get assembled into the diverse instances of the kernel. This features are similar to what we call concerns within this work.

In this work, we spare the discussion about feature oriented programming and concerns, since we do not touch feature oriented programming in detail. In this thesis we neglect the major challenges of feature oriented software engineering and focus on the contribution of one approach that is mainly targeting the visualization of features. This approach is called virtual separation of concerns and described in [135].

Therefore, feature oriented software engineering deals with the challenge to identify the different features and to ease the handling of those. Researchers identify methods and to detect features and how different features are composed together. One goal is to identify which artifact of source code belongs to which feature to ease the generation of a program with diverse feature sets [34].

Virtual separation of concerns focuses on the idea to decompose a system in the view. We show the general approach in Figure 3.6. There, we see that the different features, occurring in a system (black circle at the top), are identified and the different artifacts are mapped to features (center). We also recognize that different modules belong to different features at the same time. Based on the detected features source code artifacts that belong to certain features or a feature sets get presented at the bottom. There, two different views are created, based on different features.

Figure 3.6. Virtual Separation of Concerns



In general, the so called virtual separation of concerns approach focuses on the development environment. Within the IDE different views are adapted and offered to enable highlighting of features and to offer a better overview for developers. For instance, features can be highlighted in

code text editors through different colors. First investigations showed that this highlighting seems to help developers for some development tasks [85].

However, also other views based on an internal feature model enable developers to explore variants that are based on different feature sets directly. Thereby, developers browse and investigate code tool assisted that belongs to a specific variant of the software. Hence, the intertwining of different features within modules is done virtually by presentation whereby the features are still intertwined, physically [254,243].

Thus, we see the difference to the encapsulation methods through AOP and similar techniques to virtual separation of concerns as follows: AOP and similar techniques separate through physical modules and virtual separation of concerns emulates a physical separation in the view. We compare this to the original statement of Dijkstra that tells us about the necessity to be capable to study every concern on its own. Virtual separation of concerns addresses exactly this claim by supporting the study of different concerns. Therefore, we sum the contribution of virtual separation of concerns up as follows:

Virtual separation of concerns advances the application of separation of concerns through the possibility to study concerns on their own. In contrast to physical separation techniques, like AOP, the approach emulates modularization through the possibility of a view based separation. Developers can use this approach to study the concerns that are physically intertwined in their own, even if they are intertwined in the same modules.

All together, virtual separation of concerns origins feature oriented software development that focuses mainly on software product lines. Often, software product lines are generated with program generators. Program generators utilize often domain specific languages and make use of them to generate the different variants of a program. Hence, those domain specific languages represent a supplemental way of concern representation to virtual separation of concerns. In order to cover this supplemental area, we get now from virtual separation of concerns to domain specific languages.

3.4. Domain Specific Languages

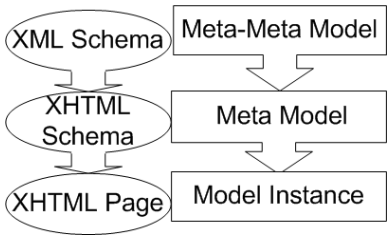
Another attempt to ease software development are Domain-Specific Languages (DSL). Domain Specific Languages are often referred to be a computer programming language of limited expressiveness focused on a particular domain. Commonly, a three tier hierarchy is used in practice in order to realize domain specific languages. First, there is a Meta-meta-model that allows the creation of Meta-models. The Meta-meta-model is capable to describe itself with its own syntax and semantics.

Popular examples for such Meta-meta-models are the MOF[9] of OMG [112], or in case of the popular Eclipse Modeling Framework [7, 56], it is Ecore. Another popular meta-meta model sample is XML Schema [10]. See the cited literature for more details and examples about Meta-meta-models.

Developers create Meta-models within the syntax of this Meta-meta-model that define a domain specific language syntax and semantics. Though, when this Meta-model exists, the domain specific language is defined and the models that are valid to the syntax and semantics of this specific language (Meta-model) can be defined.[242]. However, the easiest way to explain this approach is an example from the well known area of HTML:

XHTML is a language. Web pages written in it are defined corresponding to the syntax and semantics of this language. XHTML is defined through a an XHTML schema [11]. This XHTML schema is valid to the general XML schema. Figure 3.7 shows this relations with the model wording of domain specific languages. We show that XML is the Meta-Meta model, the XHTML language is the Meta-model and the concrete pages are the models. Therefore, XHTML is the domain specific language for web pages.

Figure 3.7. Exemplary Model Levels in case of XHTML



We remark that often DSLs, like XHTML are declarative and have additional other benefits. We do not go into details of this, since our work focuses on the general approach to encode concerns with domain specific languages. However, we refer to related work that discusses those benefits [242, 159].

From XHTML we get back to the world of programming. There, DSLs can be divided into two main subgroups. First, internal domain specific languages that are defined and part of a general purpose programming language, like Java, and external DSLs that are often used for code generation. Both kinds of DSLs follow the goal to encode domain specific concerns in their own language. In this work, we focus mainly on the internal DSLs.

The reason to focus on internal DSLs, is that our implementations are based on the Java language that offers a meta-meta model for internal DSLs.

In case of Java, such an internal DSLs are Annotations s[46]. Annotations consist out of a well defined syntax and semantics that is given by the Java language Specification.[2] Developers encode define specific Annotations once and specify which parameters they have and at which language constructs they may occur in the source code. [46] Then, instances of those Annotations can be used within code. Therefore, the Annotation specification of Java is the Meta-meta-model. The Annotation definition is the meta-model and the final usage is the model.

In order to provide a better understanding, we present two excerpts of Java Annotations definitions and their application in Example 3.2.[3]

Example 3.2. Annotations of the Java Persistence API

```
@Target(value=TYPE)
public @interface Entity
... -...
@Target(ElementType.TYPE)
public @interface Table {
    String name(-) default "";
    String catalog(-) default "";
    -...} -...
@Entity
@Table(name="Customer")
class Customer{ -...}
```

For information beyond the scope of this work, see internals of the Java persistence API[19] and the Javadoc of the specific Annotations.

The Annotations in the listing are out of the Java persistence API and get used to specify internals about the class/object to relational table mapping. We see that the target Annotation specifies the

[2] http://docs.oracle.com/javase/specs/ - 7.11.2012

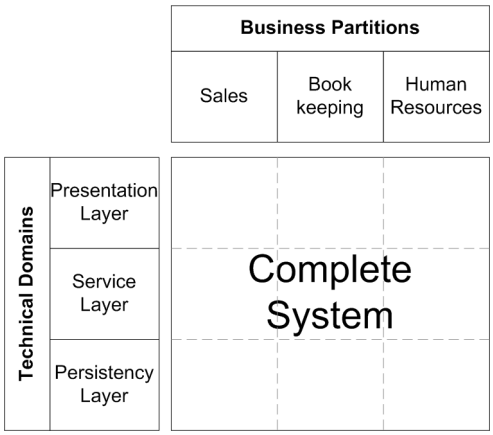
[3] Javadoc of the specific Annotations: http://docs.oracle.com/javaee/6/api/javax/persistence/Entity.html http://docs.oracle.com/javaee/6/api/javax/persistence/Table.html - 7.11.2012

allowed occurrences in the source code. In this case the occurrence target is specified to be a Java type like a class or interface. The Entity Annotation marks a type to be persistent in the database what is done with the Customer type. The Table Annotation specifies the table where the Entity type is mapped to. We see that the persistent Customer class is mapped to the Customer relational table. All together, the persistence Annotation definitions (in this case the @Interface Table and @Interface Entity) are the specification of a domain specific language.

Therefore, the Annotation declaration is the Meta-model of the language. The Java language allows the definition of such DSLs and is the Meta-Meta-Model. The concrete instances of the Annotations at the Customer class is finally one model of the domain specific persistence language.

Our example that Annotations can be used to indicate the viewpoint of persistence of classes. This viewpoint is a concern of a system. Experienced authors of DSLs recommend, similarly, to create a DSL for each concern [256,S.19]. Therefore, DSLs can be seen as additional mean to separate concerns. In addition to this, we show in Figure 3.8 an abstract view leaned on how other authors see a software assembled out of DSLs together.[4] We see that a final software system is consisting of DSLs. The vertical arranged DSLs describe technical concerns. The horizontal arranged DSLs describe business concerns. Finally, the graphic shows that concerns are composed together in a system. Hence, before the system is composed together, the concerns are separated into their own DSL. In the final software system all of them the concerns are intertwined. This way, each tiny box, that represents an artifact of the software system, gets associated with multiple DSLs / concerns at the same time.

Figure 3.8. Software System Composition with DSLs



We see a Software System that is defined through various DSLs. The shown DSLs represent different concerns which interact in the complete system. Therefore, every tiny box of the complete system is associated with multiple concerns at the same time. Thus, DSLs allow to study concerns on their own, but by composing the artifacts together, intertwining occurs.

Out of the multi-concern association of system artifacts, we get reminded on virtual separation of concerns. Virtual SOC allows regarding concerns on their own, but in the original system, those are intertwined. We point out this similarity so explicitly to credit that the problem of concern representation and composition is not limited to the traditional source code, but also appearing within other techniques. It seems one main challenge in modern programming is not only to separate concerns, but rather to decide which technique to use.

All together, we presented that DSLs are an additional mean to encapsulate concerns of a software system. Like the other means of separation before, DSLs are finally intertwined into a program and artifacts belong to multiple concerns. We also showed that DSLs can be embedded in the Java

[4]We abstracted and altered the visualization shown in [242] S.273 for a better explanation in our context.

language and follow a well defined syntax and semantics. Since the Java the Annotations DSL is embedded in the programming language, this DSL can clearly be composed with other means to separate concerns, like normal object oriented ones, but also with advanced ones like Aspects.

However, within the community of system creation with DSLs, often researchers focus on graphical models. Developers face with the graphical software system creation the problem that artifacts of the DSLs are connected within the final system. Therefore, researchers investigate and develop methods to ease the life of programmers in the context of DSL interaction can be eased. In the following section, we credit an attempt that focuses especially on the problem how developers can be supported by the interaction of different DSLs.

3.5. Orthographic Software Modeling

DSL based system creation is often associated with system creation out of UML models. In current development environments, developers use a collection of views, which are loosely related to each other. The different views present various graphical models (UML- Diagrams) that present different perspectives of a software system to a developer [40]. The approach of Orthographic Software Modeling (OSM) that we describe in the following deals with the challenges in this area.

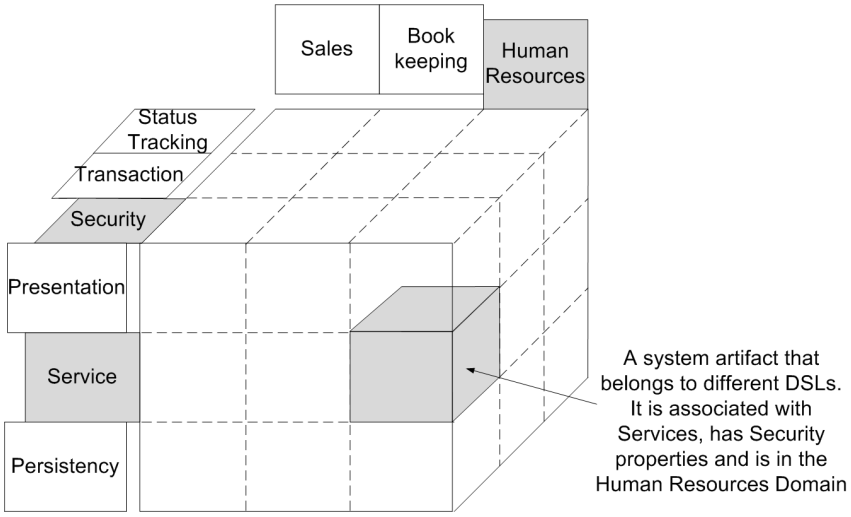
In the development of a software system with DSLs, every perspective (often just named view or model), provides different concerns of a system. Once one model is altered, this influences other models that can be shown in other views. Logically, view maintenance, overlapping model data consistency, model management and view navigation get problematic. For instance, if an entity is deleted in the persistence layer a problem arises. This can inflict that this property is not accessible any more in the sales business domain. In general, the origin of the problem is a non-existing holistic data model. When the data in one model is changed, it may not influence another view. Furthermore, properties within a DSLs that occur in one model can also appear in a different context of another model.

Nowadays, the models have limited or no interaction and their navigation is independent. Currently, the linkage of the models to the implementation and the interaction of those have to be done by the developers in their heads. Developers know which properties are represented, in different contexts, in the diverse models. However, navigation between models is not (really) supported by tools. Thus, each model has to be navigated on its own without the ability to combine the navigation and interaction for the various models. This points out the lack of consistent central contextual model, supporting a clearer design of interaction for DSLs and the to their corresponding view representation [39].

Figure 3.9 visualizes the explained interaction when a system is created. We imagine that all the horizontally, vertically and depth axes contain different DSLs. When a specific model of a DSL is altered this inflicts other DSL models. Developers navigate within the development environment and select different models of the system. By doing so, they select different cells of the cube. OSM proposes to advance this navigation and to ensure the consistency between the models. This means, when a model of one DSL is changed, also the corresponding models of other DSLs are altered accordingly. However, every cell in the shown cube represents a view of a software artifact. Through the choice of dimensions the user navigates to a specific cell. Hence, it gets clearer why the method is called orthographic, because, taken the visualization into account, the dimensions stand orthogonal upon each other.

Logically, out of the cube with multiple dimensions, OSM is trying to overcome the integration and navigation problems by introducing a multi-dimensional viewpoint towards a software system. This viewpoint is based on an analogy to the Computer Aided Manufacturing (CAD) where different projections of the same facts are presented to engineers. Engineers model each perspective on its own and through their relations a holistic model is built, internally. In case of software system creation, the diverse views of CAD are the various UML models. Therefore, the main idea is that the different projections (models) of the software system can be used to manage the complexity.

Figure 3.9. Model Landscape

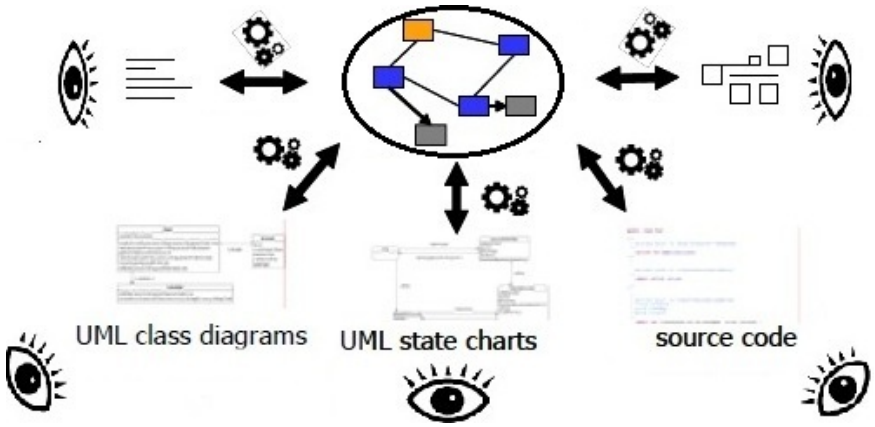


This visualization advances the viewpoint of the prior section that system artifacts belong to different DSLs at the same time. Different DSLs are arranged around the cube and system artifacts are associated with different combinations of those DSLs. [5]

In order to enable the OSM-CAD idea, a consistent model out of which these views are created is needed to ensure the consistency between the various views. To create the views out of this model different mechanisms are needed to provide the ability to generate the views. Moreover, OSM proposes additionally a coherent navigation and management mechanism to navigate between the various views [37].

The proposed navigation mechanisms of OSM credit the general idea of modeling to provide abstraction. Therefore, OSM proposes that the different dimensions of DSLs can also contain diverse levels (like the abstraction dimension; specification, realization and implementation). This is very important since different models are at different abstraction levels of a system. Just imagine, the UML models where we have class diagrams and the concrete source code that is a more fine grained. Therefore, it is important to credit this abstraction levels in the navigation and give the developer mechanisms at hand to specify the abstraction level. However, we show OSMs technical perspective of how navigation and consistency can be realized in Figure 3.10.

Figure 3.10. OSM with a Central Model



The OSM vision of multiple DSLs, interacting by using a central model. DSL models are generated on the fly and changes in the different models are transformed and stored in the central model to ensure consistency between the overlapping model elements. Picture source: [39]

[5]See also the presentation that served as inspiration for our visualization: http://d3s.mff.cuni.cz/research/seminar/download/2010-02-09-Atkinson-OrtograhicSoftwareModeling.pdf - 7.11.2012

In detail, Figure 3.10 describes the vision to enable navigation and consistency with a central model that contains. This model is then used to transform and generate the specific models for the DSLs on demand. The different DSLs store their changes within this model. Since all other models are generated out of this central model, the changes are also available in other models. Therefore, consistency is ensured.

With this consistency ensuring component the means are at hand to implement the navigation between the models. OSM introduces the selection of different dimensions and their levels. Those dimensions and levels get defined at an abstract base, like the prior mentioned implementation of code or specification of class diagrams. A developer selects a combination of different dimensions and their levels, like for instance the implementation and a process dimension, and gets a suiting model presented. For the mentioned process dimension and the implementation level can this be a sequence diagram or something similar. [38]

In order to wrap up the main contributions of the OSM approach, we note that OSM proposes to provide a better overview of a software system through the usage of models and their navigation. The OSM approach recognizes that reality is intertwined in a final software system and accepts that the same elements are appearing in multiple models at the same time. The solution approach is a consistent model and a navigation system that acts at different abstraction levels to generate a suitable and desired presentation.

We align the thought of abstraction with Dijkstra original paradigm to separate concerns and gain the new mosaic to ease life of developers in the form of the demand to provide abstraction with models. We see this as extension of separation of concerns since Dijkstra just stated that a concern should be studied from its own perspective and did not mention that to study a concern at several levels could be beneficial. The advanced mechanisms to separate concerns, like MDSOC that were described in this chapter, did not introduce the idea of abstraction, explicitly. Therefore, we note:

Orthographic software modeling introduces the idea to navigate and encode concerns by the usage of domain specific languages. For the sake of consistency a holistic model should be developed to ensure that the intertwining of the overlapping artifacts of the models is maintained. Additionally, a multi-dimensional navigation mechanism between the models is proposed. OSM contributes a new idea to the original thought to separate concerns: Different abstraction levels can be used to study the same concern.

3.6. Source Code Analysis

The prior explained approaches target mainly mechanism that can be directly used while encoding a system. However, development and advancements of software engineering do normally not start on a green meadow. Rather development activities start by reusing or advancing what is already there. Therefore, one huge challenge aside of encoding and navigating between concerns is to reveal concerns in intertwined and tangled source code. In general, the analysis of source code is not only limited to revealing concerns and rather focusing on general attempts to investigate the architecture of programs. We give an introduction to source code analysis, often named as mining software archives, in the following.

Mining software archives deals with automated extraction, collection and abstraction of data generated in the development process [107]. Mining is commonly done by data extraction out of various systems and the alignment in a format for analysis [88, 235]. Currently, researchers built their own parsers and custom infrastructures to collect the needed artifacts together. The process of assembling the facts together is so time intense that researchers even offer infrastructures, containing ready-to-use parsed and extracted source code [42]. However, those infrastructures and parsing algorithms are built for the specific research cases and do not explicitly target the integration of data that is associated with the development of software, like bugs or similar.

Thus, to alter or switch to another perspective for the analysis means to alter the custom built extraction mechanism or to customize an infrastructure. Therefore, researchers try to create

infrastructures that enable to store source code structure and extract artifacts with queries. One popular infrastructure is based on a small relational database (four tables) that can be queried with SQL [42].

Anyway, interesting results get revealed out of source code mining. For instance, in the case of investigating the eclipse IDEs source code researchers found out that bugs that are closed on Fridays are often opened up again on later dates [235]. Due the custom tools that are used to uncover such facts, this kind of results this mining is interesting for the specific project, but it is not transferable to other projects. In order to do the same investigations for other projects, time consuming extractions and investigations would need to be done to uncover the specific "bug-reopening" weekdays for other projects.

The main reason that no wide range use of code mining in practice happens maybe because of the lack of integration and a standardized investigation process. Hence, the research is not transferable to the industry, since the time and money effort is not justifiable for one potential revealed circumstance. Furthermore, the prior mentioned custom tools and extraction mechanisms lack the capability to extract a certain fact set for analysis at a specific aggregation level, since they are targeting a primary use case what could be called “the current investigation of interest”. In special, different researchers target different facts to be analyzed and focus on their specific research case. For instance, ones focus on the structure within source code [43] and others focus on bugs [235]. Currently, there seems not to be any attempt to create a common industry transferable infrastructure that gets over the custom use cases and allows a generalized extraction, integration and mining on one infrastructure.

Researchers already propose Software Intelligence that leverages means and methods that are applied within the area of Business Intelligence and transfers them to the area of software engineering. Thereby, they claim that Data Warehouse methods can be used to apply the research results of repository mining within the industry [118]. Therefore, a tool similar to a Data Warehouse for extracting facts for a perspective of interest would help to concentrate on the primary task to analyze the data. An infrastructure leaned on principles and Data Warehouse tools that are already used within businesses would enable the transfer of research methods and analysis to real world developers. Thus, a source code model is needed that can be used to integrate data from various sources at one central point. Such a model could then be used to extract desired data and apply then the analysis methods to it, without the time-consuming effort of extraction and alignment.

All together, we see the need to credit current structures of technology that are applied in enterprises to enable "broad band" code analysis in the industry. In general we see software analysis and software creation as a two coined medal where one side needs to respect the other. On side of the analysis, currently the flexibility is missing to have an adaptable analysis chain to allow transfers to practice. In order to respect this, a new method needs to come up with a industry accepted standardized investigation process. On the other side of the medal, source code analysis has to come up with a method that unleashes synergies with software creation methods.

3.7. Summary, Conclusions, Gaps

We showed different advancements in software engineering. First, we described multi-dimensional programming approaches that enhance modularization to encapsulate concerns better into modules. From this physical concern-module encapsulation, we got into the approach to separate concerns virtually within the view. Virtual separation of concerns enables presentation based concern separation even if concerns are intertwined and not separated in the source code. As another presentation based separation technique, we presented domain specific languages that allow to encode concerns in their own language, where the concerns are encoded in their own languages and get, finally, composed together in the resulting software system. On top of domain specific languages, we introduced orthographic software modeling that allows to navigate in a multi-dimensional way over software models. From all this software creation based techniques, we flipped the coin to software analysis. There, we pointed out that software analysis is a beneficial and important area that reveals interesting results about code structure. In order to advance the area,

researchers already propose to adopt means from the area of Businesses intelligence for source code analysis.

We recap the essence about the diverse shown techniques in this chapter. Again, we formulate short claims out of the key contributions that should be respected in order to derive a new technique.

3.7.1. Conclusions

Research reveals mechanisms about advanced separation of concerns. Today, programmers can use advanced modularization techniques like domain specific languages and advanced programming paradigms to encode concerns into their own units. These modules are then weaved together into a final program. All together, the multi-dimensionality of the concern space makes it hard to study it. Also, programmers face the challenge of the huge amount of source code to gain overview. Therefore, the challenge to navigate between the different concerns and to understand their arrangement still exists. Out of this, we claim the following:

The multi-dimensionality of the concern space should be respected by a new technique

However, in the last chapter we already credited that often concerns get not separated totally and programmers have a hard time to apply the concern modularization. Virtual separation of concerns proposes a view based method to present concerns separated in the presentation. Therefore, we advance the prior claim that also not separated concerns should be credited.

A new technique should respect the interweaving of concerns in source code and the current limitations in programming, where modules belong often to different concerns at the same time.

Even with intertwined concerns programmers use modularization techniques to use functionality of modules functionalities. We saw that in the example of domain specific languages, multi-dimensional separation, as well as in the chapter before, where we saw how framework logic is associated with custom code. In general, module functionality is summoned and composed together by method calls, inheritance or by advanced means. Thereby, the same functionality gets often duplicated, but is still summoned by a mean of modularization. Hence, with traditional object-oriented programming programmers face the challenge to investigate this kind of "imperfect programs" wherein the multi-dimensional concern space manifests. For example, concerns like logging or security appear multiple times as code duplicates. But even though, security or logging functionality is summoned by method call and those calls can be uncovered. Therefore, we note:

Concerns are often tangled and scattered with modularization techniques that are detectable.

Orthographic software modeling introduces the viewpoint to see models as projections of different viewpoints to a software system and depicts a multi-dimensional navigation and access problem. As solution the integrative approach of a holistic model is postulated. In this model all the various facts of models are integrated. All together, the approach sees the necessity to use models for software development to raise the abstraction level and enable multi-dimensional navigation to advance software development. We wrap this up as follows:

Abstraction and navigation is a key point in modeling, hence a technique should be capable to provide abstraction mechanisms and the multi-dimensional navigation of those.

In order to investigate programs by technical means, researchers do software repository mining. Thereby, they try to reveal concerns mechanically or to analyze the occurrence of bugs within issue tracking systems. Currently, researchers built their own parsers and custom infrastructures to collect the needed artifacts together. Hence, the research is not directly transferable to the industry, since the time and money effort is not justifiable for one potential revealed circumstance. Researchers already propose Software Intelligence that leverages means and methods that are applied within the area of Business Intelligence and transfers them to the area of software engineering. Thereby, the researchers propose that these methods can be used to apply the research results of repository mining within the industry. Hence, we define as claim on a new technique:

The technique should provide means to integrate data and analyze it from a central point and allow new facts to be “docked” together.

All together, we recovered all the different techniques and described their essence. In the following, we give an overview to of the current lack of integration.

3.7.2. Gaps

In order to give a better overview about today's shortcomings and lack of integration of the various techniques in software engineering, we show Table 3.1. There, we listed the different approaches horizontally and features of those vertically. We see that different approaches support different features to a varying degree.

Multi-dimensional approaches support a fixed set of different viewpoints of a software system. They support modularization of concern interweaving in traditional object-oriented programs through new modules. However, this approaches do still not explicitly credit that a large system, that even if properly modularized, will be hard to be investigated. The presentation of detected concerns is also not a first class citizen of this techniques and would have to be realized through additions.

Virtual separation of concerns also supports different viewpoints through its presentation based approach. Like the multi-dimensional approaches before, it does not address the immense size of today's code bases. However, the approach also uses code analysis, like the usage of annotations and compiler meta data like preprocessor directives, to detect concerns and generate the virtual separation for them. This is interesting, since virtual separation of concern is the only technique, aside the explicit source code analysis, that investigates code.

Domain specific languages are similar to the prior ones and differ mainly through the usage of models to handle the complexity of the concern space. In relation and the usage of DSLs the Orthographic software modeling approach is much more interesting. Here we can see that the internal model manages the multi-dimensional space and its the only creation technique that addresses the complexity of code bases partly through abstraction levels within the models. The other techniques lack this. Also, OSM makes the contribution to address the navigation within the multi-dimensional concern what the others do not credit in this way.

At the end, it seems nearly vertically arranged that we mention Source code analysis. The main difference in the different viewpoints is that code analysis targets to extract samples from source code to apply investigations. Therefore, the viewpoint is named dynamic. We wonder why dynamic viewpoints are not credited in creation techniques, since source code is more read then written. However, similarly strange is that the multi-dimensionality of the concern space is not mentioned explicitly in the area of code analysis. The immense size of code seems to be respected through the usage of a relational model. However, on the other hand a relational model does not help to let the views scale. When we compare a relational model with OSM and abstraction through models, we have only one side of the big data in source code analysis. Therefore, we see it necessary to use not only a database that handles many records, but rather focus on a suiting presentation technology to address the size of code bases. Lastly, source code mining is applied after the extraction and not

deep and integrated in a process chain or source code analysis infrastructures. Virtual separation of concerns, on the contrary, reveals synergies just by combining a bit of analysis logic with other tools.

Table 3.1. Lack of Integration

Support / Approach	Multi-dimensional approaches	Virtual separation of concerns	Domain specific languages	Orthographic Software Modeling	Source Code Analysis
Different Viewpoints (SOC)	yes	yes	yes, through models	yes, through models	partly, viewpoint extraction, dynamic
Concern interweaving	yes	yes, explicitly	solved through generators or model composition	solved through internal model	sometimes, concern mining
Multi-dimensional concern space	physical	partly, virtual view based separation	partly, different models and Meta-Models	multi-dimensional navigation between models	-
Explicit address of the size of code bases	-	-	-	partly, introduces different abstraction levels	implicit, uses a SQL database
Analysis/ Integration/ Mining	-	partly, uses annotations	-	-	mining algorithms are applied after data extraction
Key contribution	multi-dimensional encapsulation and composition	view based separation	abstraction, hidden composition through internal weaving	view separation, abstraction, multi-dimensional navigation	code analysis with SQL queries

The table shows, when we wrap it up, that no technique is complete. Our description above described that the diverse techniques could benefit from the contributions of others. Therefore, we see the main gap as the integration of those techniques together. Also, we assume that an analysis platform that credits the different contributions of the paradigms and approaches can be used in multiple ways and bring back benefits to the different domains. Hence, we see the need to search for a suitable technique that allows to bring the key contributions to one level together and leverages further synergies.

In this context that an integrating technique can leverage synergies, we regard the table again and see that the different techniques address distinct goals. Logically, the contributions of the techniques differ. But, in general, the different software creation based methods, multi-dimensional approaches, virtual soc, DSLs and OSM all target more or less a multi-dimensional viewpoint towards software. When we regard the software analysis part, we see that the multi-dimensional idea is not credited explicitly. Out of this, we searched for techniques from other areas that leverage multi-dimensional viewpoints of data and get used widely within the industry. We found Data Warehousing as a central technique that allows to integrate data from diverse sources in a multi-dimensional way. Therefore, we describe Data Warehousing in the following chapter.

4. Data Warehouse

"The most valuable commodity I know of is information."
Gordon Gekko. Wall Street. 1987

Abstract. Imagine the world of an executive. Dozens of plants, hundreds of salesmen, thousands of employees and millions of sold products. How do you push the company in the right direction? Who are our star salesmen? Which are the most profitable products? Where get which products sold most? These questions are asked by executives in order to steer a company. Commonly, companies use Data Warehouses to assemble all the necessary data together and to generate reports for the various hierarchy levels within a company. In order to introduce Data Warehousing to the software engineering community, we present insights about the use of the technology. We give examples of data structures and explain the general multi-dimensional approach that we use later for source code. Additionally, we dig out an example from the area of software project management that uses a Data Warehouse approach. Now, the reader understands the basic principles of Data Warehousing.

4.1. Introduction

In the prior chapters, we gave insights about techniques and principles that are used within the area of software engineering. Plenty of the techniques are multi-dimensional approaches. In the current chapter, we describe Data Warehousing that is a multi-dimensional approach from the area of business computing.

Hence, the motivation to show this chapter is to provide background information to the reader who does not have much knowledge about Data Warehousing. Furthermore, we use it to credit Data Warehousing as inspirational area for our own work and reference related literature from this area. However, we show examples and give impressions what can be done with Data Warehouse technology and how it is used and which kind of operations are supported. But, still, Data Warehousing is a huge area of research and this chapter presents only an introduction that does not go into in-depth details about the internals of a Data Warehouses and is neither a complete overview or warp up of the area. Therefore, this chapter also presents a starting point for the interested reader to dig deeper into literature.

4.1.1. Contribution

The contribution of this chapter is to introduce Data Warehousing to the software engineering community. We give insights about application scenarios, multi-dimensional data structures, a basic relational model, multi-dimensional operations and reporting. The reader is now able to understand how data is arranged in a multi-dimensional model and how it can be transformed with various multi-dimensional operations.

4.1.2. Reading Guide

First, we present an introduction to Data Warehousing by describing the areas wherein the technology is commonly applied. Succeeding, we illuminate different data structures that are used in Data Warehouses. Then, we refer to different multi-dimensional data investigation operations. We wrap up our introduction by giving references to further literature about Data Warehouses. Finally, we give a summary and draw conclusions.

4.2. Data Warehousing for Analysis

In the area of business computing Data Warehouse systems are used to integrate the data from various sources into a central repository. This central repository is then used for data explorations [128]. However, closely related to the term Data Warehousing is the multi-coined term Business intelligence (BI). The term Business intelligence is often hard to discriminate from the term Data

Warehousing. For instance, BI is often used as a collection of all tools and methods that lead to business insights by data analysis [116, S.6]. Thereby, literature also often refers to the term Online Analytical processing (OLAP), which is a multi-dimensional software processor for data exploration and computations. Another often credited and associated process with Data Warehouses is the process to extract data and prepare it for analysis a so called Extract Transform Load process (ETL) [190, 137, 109]. Out of this many terms, we neglect the discussion about what is Data Warehousing and which techniques can be associated with it, since it is not important in our context. We credit the techniques that we need to derive our own technique and refer more to application scenarios and the general data exploration with a multi-dimensional processor than to dig into specific details.

In our work, it is not important to discriminate all the terms of Data Warehouse, Business Intelligence and OLAP perfectly sharp, since we focus mainly on a few use cases. For a detailed discussion of Data Warehouse systems features, we refer to the original rules of Codd that provide a more technical perspective [71]. Furthermore, we refer to actual literature [146] that provides detailed discussions, classifications and deep insights about the different terms and technologies that we utilize.

However, different Data Warehouses are suitable for our needs. Right in this work, we use Microsoft Analysis Services 2008 R2 and version 2012 as exemplary Data Warehouse.[1] The main reason why we use this product is that there is a demo version available and it is not a patchwork solution that needs a lot of hands on to get the infrastructure running. However, all Data Warehouse solutions are similar and our results can be transferred to other solutions as well. In general, a Data Warehouse is a software system with the capability to store a large amount of data and to process and analyze it. Often those systems are described by the following demands:

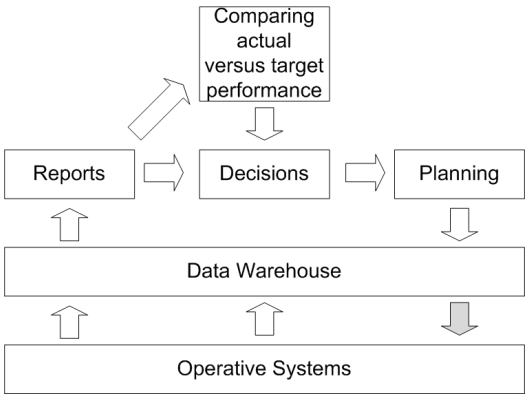
- Fast: Queries should not exceed five seconds in medium. Simple queries should stay under one seconds and only a few complex ones can go up to 20 seconds.
- Analysis: A system should be capable to handle complex analysis queries. A complex analysis with queries should take minimal effort.
- Shared: A system should be capable to handle multi-users and provide suitable means to access the system.
- Multi-dimensional: Data should be arranged in a multi-dimensional way and provide means to support hierarchies of dimensions.
- Information: The system should be capable to handle large amounts of data to have all necessary data at hand at the analysis. [2]

In the following, we give impressions how Data Warehouses are used within normal business scenarios in enterprises. Therefore, we describe the general process how a Data Warehouses can be used in the following. Figure 4.1 describes the data flow in an enterprise. The data of diverse operative systems[3] gets extracted and stored into a Data Warehouse. In there, the data gets staged and loaded into the data cubes. Those cubes represent the prior mentioned multi-dimensional structures which are queried to generate reports. Often there is also tool support to create reporting dashboards, containing graphical elements that can be used by people that do not know how to built queries manually. Out of these reports decisions are made and the business plans are adapted. In order to specific future business goals out of the reports the future business figures are persisted within the Data Warehouse. In general, there is clearly also the possibility available to write the future goal figures back into the operative systems to use them there also (grey arrow). Through the plans realization in daily business new data is generated within the operative systems (for instance the amount of cars that are sold within a region). This data is then loaded in the Warehouse, again. Like before reports may be generated and comparisons to the historical data can be done. This way, executives can determine easily the performance increase of their company within a certain time

[1] https://www.microsoft.com/sqlserver/en/us/solutions-technologies/business-intelligence/analysis.aspx - 7.11.2012
[3] Such systems can be customer relationship management or enterprise resource planning systems.

frame or the performance of specific regions. However, the newly extracted and loaded data can also be compared with the originally planned data that was persisted before. Through this comparison it is possible to discriminate if desired business goals are reached.

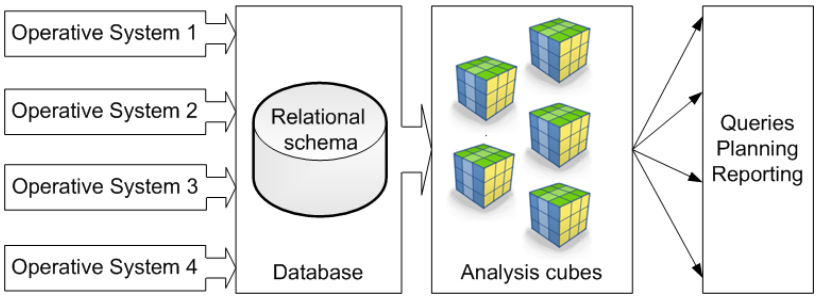
Figure 4.1. Data Warehouse Process



Data gets extracted from operative systems, such as enterprise resource planning. Within the Warehouse data is then arranged in a multidimensional way and analysis with reports is done. This reports serve as foundation to do business decisions or to compare them with prior defined business goals. The business decisions can also be used to do future planning about future business goals that get written back to the operative systems. A similar graphic can be found in [109, S. 20].

However, from this rather abstract scenario we get into the different parts and processes that are used within the area of Data Warehousing. We show that the generic approach to load and analyze data that is commonly used within the context of Data Warehousing in Figure 4.2. The picture describes the extraction of data out of operative systems into a relational model that is then used to fill multi-dimensional cubes. They serve as base to execute queries on them that are used to generate reports and to explore the data. A very common practice is to arrange these reports in dashboards that contain results of multiple queries as visualizations [185]. Users with less technical affinity, like executives, can use these dashboards to explore the data. Nonetheless, dashboards are just one kind of application of the queries on the cubes. Another advanced operation is persisting user data in the cubes to enable planning applications. Thereby, the process of data exploration gets reversed and the user enters business indicators directly into the cube.

Figure 4.2. Commonly applied data Flow in a Data Warehouse



Data of operative systems gets stored in a relational schema. This relational schema is then used to compute and fill multi-dimensional data cubes. On top of this cubes different query based applications can be applied. Data Warehouse structures come with tools that support the data flows and the building queries and reports.

Note, in this work we focus on a intermediate relational model in the database and the schematic cube representations. Data Warehouses often use and offer supplemental storage techniques and multi-dimensional storage technology. We concentrate on the relational approach and the schematic cubes, since the internal way of operation of a

Data Warehouse is not the focus of this work. For further information about the multi-dimensional storage technologies we refer to related literature [206, 253].

4.3. Data Structures

Before, we described an arbitrary overview of Data Warehouses. Right now, we dig into the specific data structures that are used as internal models within a Data Warehouse. First we cover some background information about schema development and data models. Then, we dig into a concrete relational schema instance, follow up with the multi-dimensional model and close this section with the associated computation rules.

4.3.1. Data Models

A general paradigm in Data Warehousing is to divide data into facts, cubes, dimensions, hierarchies, computation rules and indicators. Those are modeled in the relational and multi-dimensional level. We define the different terms as follows:

Indicators, also named figures or measures, specify numeric measures. Those numeric measures are derived by computations (e.g. counts) or are given in the database. Examples for indicators are revenues, loss or the amount of sold units.

Dimensions are classifications for entities like people, products, dates or geographic regions. A dimension describes one possible viewpoint towards an indicator or a relation to another dimension. Hence, it is a limited set of elements that share a semantic relationship to describe different instances (also called members, dimension elements) of the same viewpoint. Dimensions may have attribute variables that describe properties of the dimension members. Dimensions are used to structure the data space orthogonal.

Hierarchies specify generalization paths within a dimension to more generalized attributes that occur in the dimension; Or hierarchies specify a generalization path along dimension associations. A dimension and its attributes may be a member in multiple hierarchies at the same time. An example for a hierarchy is a product category that can occur as dimension member or as own dimension that groups certain members of a dimension together. Likewise the material of the product can act as a second hierarchy.

Computation rules describe how indicators are computed in relation to a set of dimensions, a single dimension, its elements, attributes and/or in relation to one or more hierarchies. They specify the algorithm and/or which arithmetic method is used. They are applied to compute aggregates, whereby, an aggregate corresponds to a collection of dimensions, their attributes, or hierarchies. An example for applied computation rules and creating an aggregate is to built the total revenues for a product or different product categories.

Facts specify the association of at least one dimension member with an indicator. This association is then used together with computation rules and hierarchies. Facts can also describe the relation of dimension elements to each other, without using a given data indicator and just count the associations. Facts are the foundation that computations and aggregations along the dimensions and hierarchies can be applied. This way, facts correspond to a specific business case. An example for facts is the association of revenues with a product and the time when it was sold. Together with the computation rules, aggregates for a certain time frame of for a product category can be computed.

A **cube schema**, also called data cube, just cube or multi-dimensional model, is a possible subset of the combination of dimension, dimension members, dimension attributes, hierarchies, indicators, computation rules and facts that are composed together to have analytic capabilities to investigate a certain set of business cases. An example for such a schema can be the analysis of sold products with certain revenues within a time frame and their aggregation to the product group.[4]

Different approaches are used to model the described elements. Researchers argue in favor for different approaches how to model the entities on a logical and relational level. Furthermore,

[4]See also the following link for a similar overview: http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/dw/dw0405/03-Multidim-Modell.pdf - 7.11.2012

different approaches are suitable for different scenarios. Since this work focuses not in advancing Data Warehousing, we do not present all different methods. We use the so called snowflake schema as relational representation since information about it is available in plenty of books that may be used for further reading, like for instance in [109,S.88]. Our multi-dimensional model is leaned against Dimensional Fact Model (DFM) cube modeling approach [108].

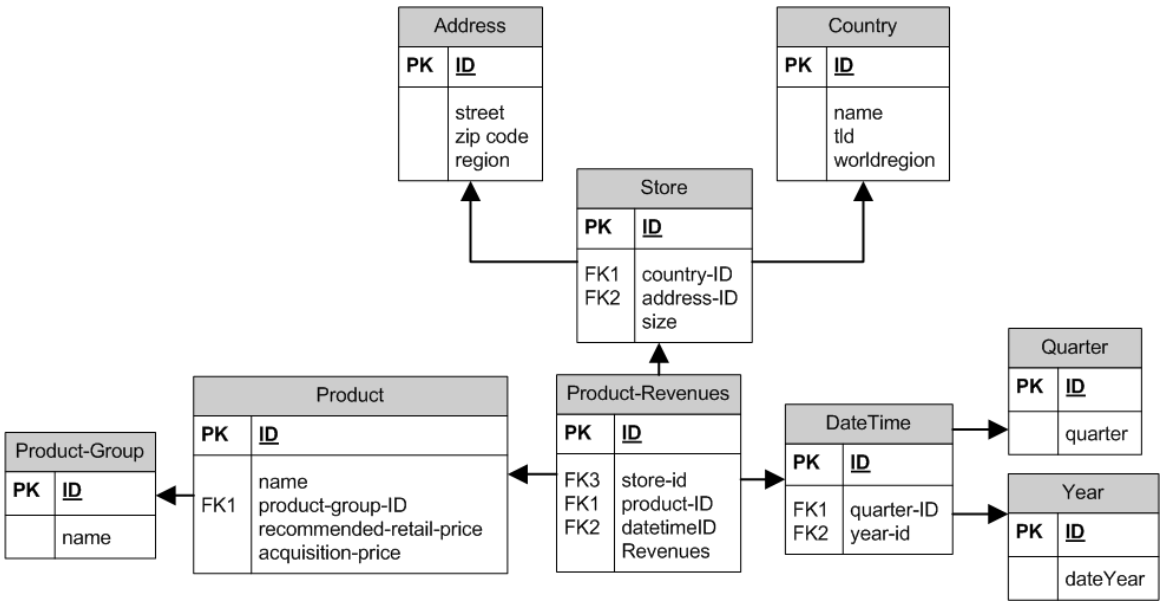
In order to present a valid relational schema, we extend and alter an example from literature Figure 4.3 to have a direct reference to complementing details about schema construction available.

In addition to the relational and the multi-dimensional DFM model, we credit Data Warehouse implementations and contribute with computation rules from relational to multi-dimensional to clarify the meaning our model in an implementation. Our computations are founded on the Microsoft SQL cube workbench, e.g. "Microsoft Business Intelligence Development Studio", that is used widely within the industry. This way, we ensure that our models and the computations can be implemented with available technology.
In the following, we first describe a relational model and then a cube on top of it.

4.3.2. Relational Snowflake Schema

We show an example of a relational snowflake schema in Figure 4.3. The fact table (Product-Revenues) associates the dimensions (DateTime, Store, Product) with each other and is located within the center of the picture. It connects the revenue indicator with a certain Date, Product and Store, what corresponds to a produce sale. We see that the fact table contains foreign keys of the dimensional tables and the revenues indicator. Note, revenues are just an example other economic indicators like profit or the amount of sold products can also be used. The tables of the diverse dimensions can be arranged into a hierarchy that is connected through foreign keys. An instance for such a hierarchic relation with a foreign key is the Product table that relates to Product group. This kind of hierarchic references look similar to a snowflake, from which the name snowflake schema origins. However, the table structure can be used to execute normal SQL queries. For instance, the schema can be queried to compute the revenues for a certain country can be queried. This schema serves then as foundation to create multi-dimensional cubes on top of it.

Figure 4.3. Snowflake-Schema Example

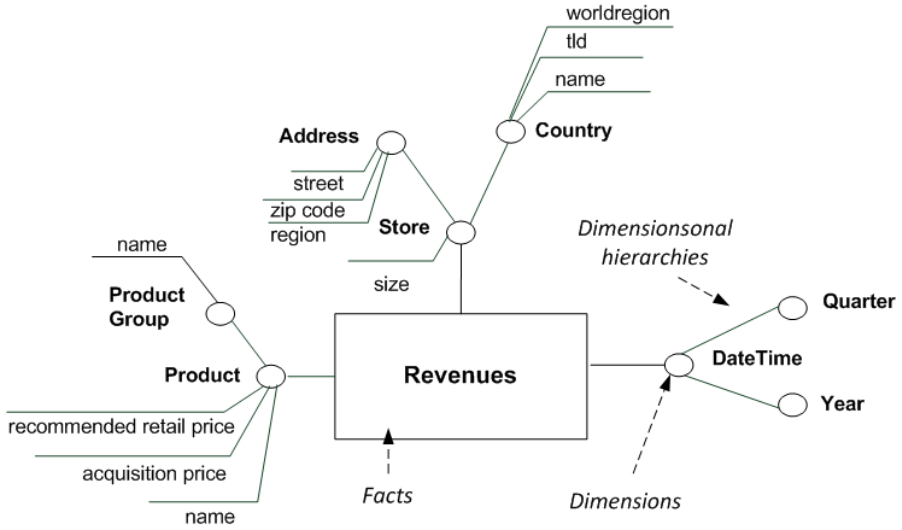


A typical snowflake schema, leaned on the original description in [109, S. 88]. A fact table connects different Dimension tables together and associates them with the revenue indicator. The Dimension tables also have some logical hierarchies that are realized with references to other tables.

4.3.3. Multi-Dimensional Schema

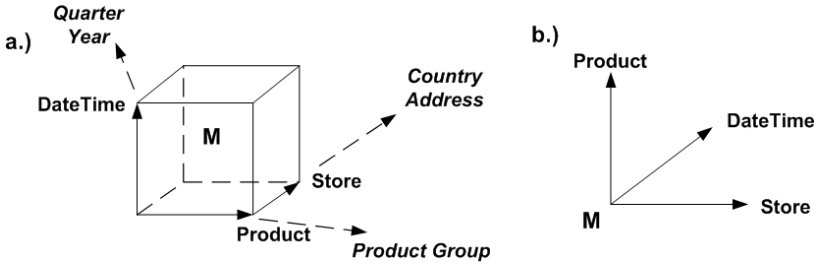
However, applying queries to such a relational structure that we described before gets complicated and hard to do, when a schema is larger. Furthermore, the needed amount of joins slows down the query performance. In order to overcome this issues and to provide an easier way to query, then with normal SQL, the multi-dimensional schemata, e.g. data cubes, are used, what we show in the following. We show the multi-dimensional schema on top of the relational schema (Figure 4.3) in Figure 4.4. We discuss them in the following.

Figure 4.4. Cube model Example



A multi-dimensional model of sales data based on the DFM notation of Figure 4.3 . We see that the facts are arranged in the center and the dimensions are hierarchically arranged in increasingly more abstract dimensions from the inside to the outside.

Figure 4.5. Cube visualization of the Multi-Dimensional Schema



Cube like presentations of parts of Figure 4.4.

Figure 4.4 shows a schematic cube arrangement. In short the circles represent dimensions and their additions represent attributes. However, we added arrows to the different exemplars of the elements to describe their meaning. We see that the different tables of the relational schema were mapped onto dimensions and the interlinked structure was arranged into hierarchies of dimensions. The hierarchies are there arranged from the fact table to the outside. Thereby most outside elements are the most abstract dimensions in the hierarchies. The additional information within the tables, like region or zip in the case of Address, are arranged as attributes. In general, every dimension or attribute may be used as element in a query to compute the revenues for the specified elements. For instance, the revenues for a certain region within a certain time for a certain product.

Anyhow, one my wonder why this multi-dimensional schema is often called cube or hypercube. Therefore, we show Figure 4.5 with an exemplary cube arrangement. We use two different representations, named a and b to show the multi-dimensionality. We see that the facts of revenues were generalized to the letter M. We use the letter of M to indicate that it may be any measure

(indicator) that is associated with the different dimensions. The different dimensions are arranged orthogonal to each other and a specific measure connects a specific member in every dimension together. We see that in the b part of the figure the point of origin of a dimension member combination is the measure to clarify our cube visualization in part a of the figure. Furthermore, we also see (the dotted arrows) clues that the dimensions that are lower in the hierarchy embed also the higher ranked dimensions.

4.3.4. Computation

Before, we described the relational as well as multi-dimensional model. One main missing point is how those models relate to each other. More precisely phrased, this is the information that a Data Warehouse uses to transform the relational data into a multi-dimensional model. Therefore, we present here exemplary computations that are stored in a Data Warehouse and get used to do aggregations. Table 4.1 provides the exemplary computations for the presented example that connect the dimensions, indicators and their hierarchies. We see that the dimensions that are directly connected to the facts have a direct reference from those. The dimensions that are higher in the hierarchy reference the attributes of the"lower ranked" dimensions in order to enable the hierarchy. The Data Warehouse uses this information as computation rules to built totals.

Table 4.1. Product Cube Computations

Dimensions	Facts: Product-Revenues
(1) Address	Reference Country (address-ID - 3)
(2) Country	Reference Store (country-ID - 3)
(3) Store	Store (store-ID)
(4) Product Group	Reference Product (product-group-ID - 5)
(5) Product	Product (product-ID)
(6) Quarter	Reference Date (quarter-ID- 8)
(7) Year	Reference Date (year-ID- 8)
(8) Date	DateTime (datetime-ID)

All together, we see that the computation table provides a mapping of the relational to the multidimensional model and is therefore the connecting element. Developers create a relational schema, a multi-dimensional one and the necessary computations how those are related. Then, data is stored in the relational model and a Data Warehouse loads it automatically with the computations in the multi-dimensional one. This schema is then accessed with queries wherefore a Data Warehouse provides the in the beginning described OLAP processor. Therefore, the next section discusses which operations can be applied on top of the multi-dimensional model.

4.4. Queries and Exploration

In the last section, we described how the data is processed within a data warehouse and how the data models look like. In this section, we present the operations that can be executed on top the data.

Often Data Warehouses support Multi-dimensional Expressions (MDX) as query language of the OLAP Processor. The MDX language is especially designed to handle and explore multi-dimensional data sets. Furthermore, the language has statistic computation capabilities for such things like correlations that are quite handy for data investigations. We provide a few query samples in later chapters. However, related literature gives insights about the MDX language and its capabilities [146, 3, 4]. Right here, we just focus on well-known OLAP operations.

In order to provide a better comprehension for the multi-dimensional query operations we provide sample data in Table 4.2. We flattened the cube in rows to provide a simple overview and have to

imagine that the data would be structured into the multi-dimensional model of the last section. We can see that multiple datasets origin the same country have the same product and so on, what results in a cube as single data the corresponding model elements.

In the following we use this sample dataset to help to explain the multi-dimensional operations that can be applied. First, we explain the standard slice and dice operations. Then, we give a brief insight about advanced operations to give an impression which kind of computations is possible with this techniques.

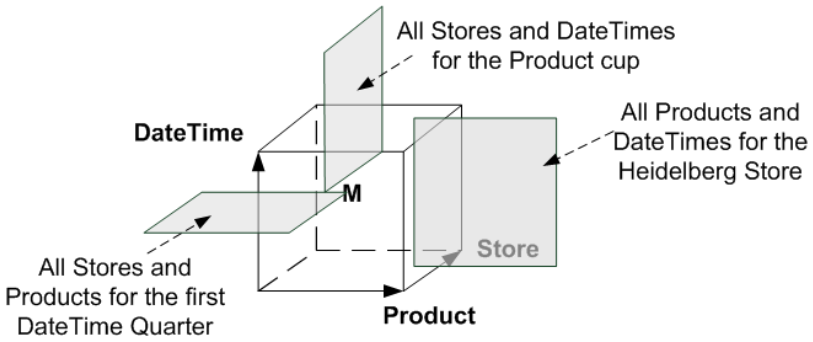
Table 4.2. Sample Data

Country	Address	Store	Product Group	Product	Quarter	Year	Date	Revenues
Germany	D-69117	Heidelberg	dishes	cup	Q1	2011	1.2011	2€
Germany	D-10115	Berlin	decoration	candle	Q2	2011	4.2011	1€
Germany	D-69117	Heidelberg	dishes	plate	Q2	2011	5.2011	3€
Germany	D-70173	Stuttgart	furniture	tablecloth	Q2	2011	5.2011	10€
Germany	D-69117	Heidelberg	dishes	cup	Q3	2011	7.2010	2€
France	F-75001	Paris	furniture	lamp	Q1	2012	2.2012	15€
Spain	ES-28001	Madrid	decoration	napkin	Q3	2012	8.2012	3€
Spain	ES-28001	Madrid	dishes	cup	Q3	2012	7.2012	2€
Germany	D-69117	Heidelberg	decoration	napkin	Q4	2012	12.2012	3€

4.4.1. Slice

Slicing is the operation to cut a specific slice out of the data structure that is commonly applied when data is analyzed. The data cube gets cut down to various dimension elements or attributes of those. We visualize different slices in Figure 4.6. There we see slices for the various dimensions of the sample data.

Figure 4.6. Slice



Slices discriminate and cut out different perspectives of the records. For instance, the store dimension can be fixed to a specific dimension member and therefore the slice for this specific store is created. However, slicing is possible for all dimensions and their attributes and the picture just shows samples how a dataset can be discriminated with slices. For a similar visualization with other sample data see [137, S.98]

We use a slice from Figure 4.6 in the following. The slice of "All Stores and Dates for the Product cup" discriminates the dataset to the fixed value of a cup. We present the result of this slice in Table 4.3. There, we see that just the stores in Madrid and Heidelberg are left that sold cups. Furthermore, we also present a common aggregation method of building totals for a slice that is commonly used within Data Warehousing. This way, decision makers have easily the total revenues for the cup product at hand. Then, they can use those totals to compare the different stores to the to the total revenues and see which store is the leading cup seller.

Table 4.3. All Stores and Dates for the Product Cup

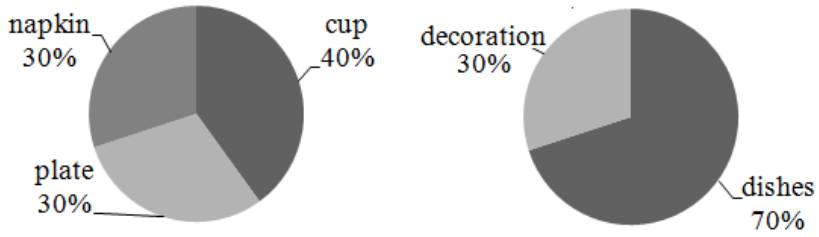
Country	Address	Store	Product Group	Product	Quarter	Year	Date	Revenues
Germany	D-69117	Heidelberg	dishes	cup	Q1	2011	1.2011	2€
Germany	D-69117	Heidelberg	dishes	cup	Q3	2011	7.2011	2€
Spain	ES-28001	Madrid	dishes	cup	Q3	2012	7.2012	2€
Total								6€

Similarly to the slice before, we present the slice of "All product and Dates for the Heidelberg Store" in Table 4.4. There we can see that the Heidelberg store sold three products for the total amount of 8€. Out of this data, charts or other graphical visualization can be populated for a specific store. Such charts may give decision makers insights if the revenues for a specific store increase over time. We show a simple visualization sample in Figure 4.7

Table 4.4. All Product and Dates for the Heidelberg Store

Country	Address	Store	Product Group	Product	Quarter	Year	Date	Revenues
Germany	D-69117	Heidelberg	dishes	cup	Q1	2011	1.2011	2€
Germany	D-69117	Heidelberg	dishes	plate	Q2	2011	5.2011	3€
Germany	D-69117	Heidelberg	dishes	cup	Q3	2011	7.2011	2€
Germany	D-69117	Heidelberg	decoration	napkin	Q4	2012	12.2012	3€
Total								10€

Figure 4.7. Slice Chart Example



Sample visualization of data from Table 4.4. The original data has been used to compute pie charts to give executives a visual overview.

Lastly, we show the slice of "All Stores and products for the first Date Quarter" in Table 4.5. We see that only Heidelberg and Paris sold products in Q1. Furthermore, the revenues compared to the total are much higher for Paris then for the Heidelberg store.

Table 4.5. All Stores and Products for the first Date Quarter

Country	Address	Store	Product Group	Product	Quarter	Year	Date	Revenues
Germany	D-69117	Heidelberg	dishes	cup	Q1	2011	1.2010	2€
France	F-75001	Paris	furniture	lamp	Q1	2012	2.2012	15€
Total								17€

However, in general such slices are the beginning of further analysis and the building of total sums and similar aggregations is a starting point to do advanced investigations. Our example contained only limited cases that are easy to overview to demonstrate the generic idea how data can be sliced. In reality, we have to imagine thousands and millions of records that hardly to overview. Though the slicing the amount can be dramatically reduced. In order to provide more detailed analysis. Furthermore, Data Warehouses allow not only to compute totals, but also subtotals and similar. For instance, subtotals may be computed for the specific stores, products and so on. Here, we focused on the generic approach to slice data, to reduce a result set and demonstrate advanced analysis

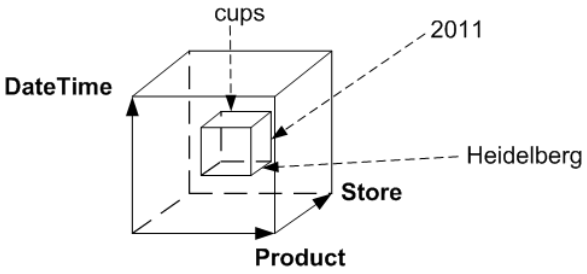
capabilities later one. For us, it is important to note, that slicing fixes a dimension on a certain member in order to reduce the original dataset.

4.4.2. Dice

Similar to the slice operation discussed is the Dice operation. The dice operation combines multiple slice operations at one time to create a subcube. We present the visualization in Figure 4.8. There we see how different dimensions are fixed to specific values and a sub cube is extracted. For our demo data we present the remaining subcube in Table 4.6. There, we see two remaining datasets are left. Furthermore, we see that slice as also dice did not change the dimensionality of the cube. All dimensions that were existing before are still remaining in the subcube.

In general, such a dice operation is often used to start an analysis for a specific entity. Imagine a local decider, responsible for the Heidelberg store, wants to do the future planning for the store in Heidelberg for 2012. He wants to determine how many cups he needs and how many he has needed in the past periods to go into negotiations with the cup vendor. In order to do so, he is interested in the history data of the year 2011 for his specific store. He is not interested in the data of the whole company and focuses on his store. So, all computations that he want to do are based on the Heidelberg- 2011- cup -subcube. If he would look the whole company data, it would be overwhelming and unimportant information for him.

Figure 4.8. Dice



Dice operations discriminate datasets to a subset of the original data. In general, dicing is done by using multiple slices together. A similar visualization with another example can be found in [137, S.98]

Table 4.6. Dice Data Result

Country	Address	Store	Product Group	Product	Quarter	Year	Date	Revenues
Germany	D-69117	Heidelberg	dishes	cup	Q1	2011	1.2010	2€
Germany	D-69117	Heidelberg	dishes	cup	Q3	2011	7.2011	2€

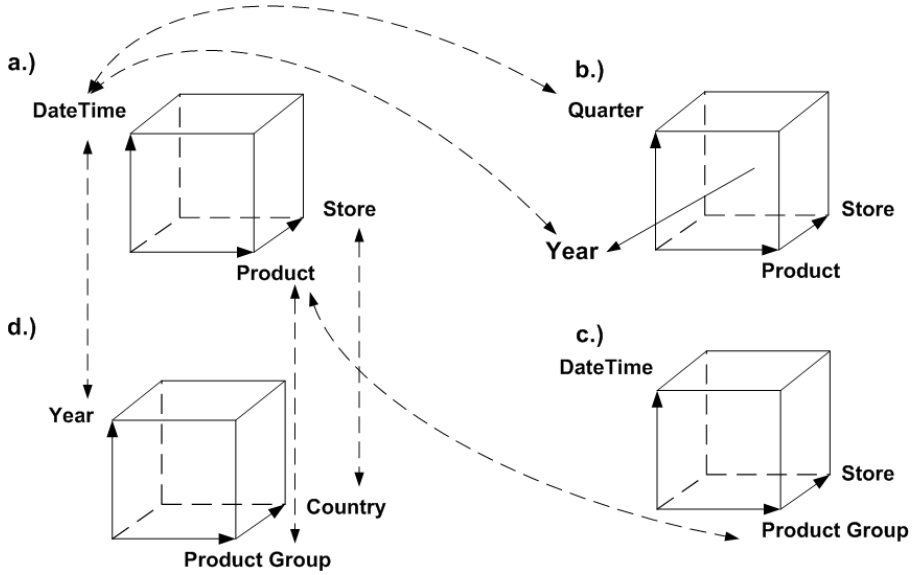
4.4.3. Advanced Operations

Before, we discussed the basic slice and dice operation. We saw that the data representation was uniform and the operations can be used to reduce the data set. As addition to those, OLAP engines offer supplemental operations from which we discuss some in this section.

We provide a visualization how our demo cube can be explored with some advanced OLAP operations in Figure 4.9. We show a subcube of the original data that can be created with a dice operation in a.). This subcube is the foundation to apply different operations and to do investigations with them. The other cubes b.), c.) and d.) show the outcome after different operations. We indicate which original dimensions and attributes result in the different outcomes by linking them with dashed lines. In the following, we discuss each resulting cube on its own. In order to provide a better understanding, we show resulting tables in the way they are commonly used by OLAP exploration tools. We also show different methods of building subtotals and totals in order to provide indications what can be done in practice. We refer not to all computations that are possible, because our main goal is to give impressions in which direction Data Warehouse tools works.

Nonetheless, since some operations have inverse operations that are named different we refer to both operations in the following. For instance, when a.) is the origin as an operation to compute cube b.) it is called split whereby b.) to a.) is called merge. We always mention first the "a.) to" operation name and then the inverse name in case it may exist. With that naming guide we first focus on the original data set in a and then follow up with the different operations. Thereby, we give always examples how the original dataset is transformed according to the operations in Figure 4.9. By presenting the different results of transformations, we also credit techniques that allow to rearrange result data sets for different perspectives.

Figure 4.9. Advanced Cube Operations



Different operations are applied to a dataset that is presented in a. From a to b a Merge is applied. In order to transform the data to c, a Roll up is applied. This roll up is applied to all dimensions in d.

a.) Dataset of Origin

We provide the point of origin, before applying OLAP operations in a.). In Table 4.7, we see the representation of the corresponding data records. In reference to our sample dataset (Table 4.2), we see that this dataset is a subset of the sample data that was created by a dice operation. Therefore, only a specific product, store, Date and revenues are contained in the records. The additional data like the country and so on is hidden in the multi-dimensional structure behind it.

We imagine, a controller looks at the data at the beginning of his investigation. He generates charts from it and applies analytic operations. Then he decides which operations he applies for further explorations. Therefore, we imagine that he can end up with the operations that we present in the following.

Table 4.7. Dice of Origin

Store	Product	Datee	Revenues
Heidelberg	cup	1.2011	2€
Berlin	candle	4.2011	1€
Heidelberg	plate	5.2011	3€
Stuttgart	tablecloth	5.2011	10€
Heidelberg	cup	7.2010	2€
Paris	lamp	2.2012	15€
Madrid	napkin	8.2012	3€
Madrid	cup	7.2012	2€
Heidelberg	napkin	12.2012	3€

b.) Split/Merge

The cube of origin (Table 4.7, Figure 4.9 - a.) is treated with the split operation generate b.). Vice versa, as inversive operation, the original cube can be created out of cube b.) by applying the merge operation. The split operation increases the dimensionality of the cube. We can see that Date is split into the year and the quarter at the same time. In contrast to the three dimensions in cube a.), cube b.) has now four. We present the result data in Table 4.8, after we applied an additional Pivot operation. In the following, we explain the Pivot operation and present the resulting data of the Split operation.

Pivote (also called Rotate). Like stated before, in order to visualize and arrange the result of the Split, Pivote was additionally used to rearrange the data. This is commonly done when analyzing data. Pivote alters the content of an axis in a spreadsheet. Therefore, it is also called rotate, because a dataset is rotated in itself. We show the outcome in Table 4.8.

We see that the quarters are arranged vertically to the years. This makes it possible to build subtotals for both; years and quarters. When we imagine a larger dataset, this gets even more handy. Currently, our dataset has no duplicate sold products in different quarters, but we see that such issues can be handled through the total aggregation that is shown vertically. We see in the resulting totals that the quarter with the most revenues in total is Q1, followed by Q2. Furthermore, we see that there was a raise in revenues from 18€ to 23€. Decision makes have this way, different possibilities to Pivote the data to analyze it form different perspectives.

Concerning the original Split operation that created the dataset, it also needs to be noted that splits are not limited to dimensions, rather they can be applied to all kind of dimensional attributes to get into specific details. Such attributes may be the store size or similar to regard every possible angle. All together, Split and Pivote can be used to increase the dimensionality and arrange the dimensions into the desired way to reveal desired details.

Table 4.8. Split and Pivote

			Quarter				
			Q1	**Q2**	**Q3**	**Q4**	**Total**
Store	**Product**	**Year**	**Revenues**				
Heidelberg	cup	2011	2€		2€		4€
Berlin	candle	2011		1€			1€
Heidelberg	plate	2011		3€			3€
Stuttgart	tablecloth	2011		10€			10€
Subtotal 2011			2€	14€	2€		18€
Paris	lamp	2012	15€				15€
Madrid	napkin	2012			3€		3€
Madrid	cup	2012			2€		2€
Heidelberg	napkin	2012				3€	3€
Subtotal 2012			15€		5€	3€	23€
Total			17€	14€	7€	3€	41€

c.) Roll-Up/Drill-Down

Roll-Up is decreasing the granularity of a dimension or a dimensional hierarchy. It is the "zoom out" operator. The opposite operator is the drill down. We use this roll up operator this with cube c.). There, we see the product dimension generalized to the product group and all the revenues are aggregated and computed together on this level. One group may contain multiple product types, but one product has only one group. This way, the product group is an reduction of different elements.

We show the Roll-Up with our sample data in Table 4.9. We see the revenues for the different product groups, dates and stores. Like before, it is possible to built subtotals like for dishes or a specific store. With such subtotals, we can depict easily that dishes have been the main sold product

in 2011 in the Heidelberg store. However, in general, Roll-Ups and Drill-Down are very effective to gain an overview or an insight into a dimensional hierarchy or attributes of a dimension.

Table 4.9. Roll-Up

Store	Product Group	Datee	Revenues
Heidelberg	dishes	1.2011	2€
		5.2011	3€
		7.2011	2€
		Subtotal	7€
	decoration	12.2012	3€
	Subtotal		10€
Madrid	decoration	8.2012	3€
	dishes	7.2012	2€
	Subtotal		5€
Berlin	decoration	4.2011	1€
Stuttgart	furniture	5.2011	10€
Paris	furniture	2.2012	15€
Total			41€

Pivot and Dimension Reduction. Like before, a result set may be investigated further, by applying additional operations. We show such a result set in Table 4.10 what applies a Pivote and exclude to Table 4.9. We pivoted the product group and exclude the Date dimension in the result set. Now, we can easily depict the totals for the different stores and the various Product Groups. This reveals that furniture is responsible for the most revenues and Paris seems to be the Store leader in revenues. Such kind of analysis is in business used to compare the performance of stores and product groups before drilling into details.

Table 4.10. Pivot and Dimension Reduction

	Product Group			
	dishes	furniture	decoration	Total
Store	**Revenues**			
Heidelberg	7€		3€	10€
Stuttgart		10€		10€
Paris		15€		15€
Madrid	2€		3€	5€
Berlin			1€	1€
Total	9€	25€	7€	41€

d.) Roll-Up all Dimensions

In the last cube visualization, we show the combination of different combined Roll-Ups at the same time. Date has been generalized to Year, Products to the Product Group and Store to its Country. The general idea is to compare the performance of countries in different years. In Germany, we have multiple stores and all of these stores are ragged together and their revenues are aggregated. Imagine, we have a large dataset, where all countries have multiple stores. Executives can have a look at the high level results and to their decisions.

We arranged the cube with a pivot of the countries in Table 4.11. There, we see the different countries in relation to years and product groups. This makes it possible to depict the top product groups and the top performing countries at once. An executive can see which product groups are not performing so well in certain countries and can to business decisions based on this data.

Table 4.11. General Roll-up

		Country			
		Germany	France	Spain	Total
Year	Product Group	Revenues			
2011	dishes	7€			7€
2011	decoration	1€			1€
2011	furniture	10€			10€
Subtotal 2011		18€			18€
2012	furniture		15€		15€
2012	decoration	3€		3€	6€
2012	dishes			2€	2€
Subtotal 2012		3€	15	5€	27€
Total		21€	15€	5€	41€

4.5. Further Reading

Before, we gave a basic introduction what Data Warehouses are used for and which areas are associated with it. Then we scratched therein occurring data structures and operations that can be applied to those. However, it is a huge research area and there is much more to know then our brief explanations. In the following, we credit literature of Data Warehousing that complements our explanations and descriptions.

For the interested reader that wants to recreate our work, we recommend different references as starting point. Free an open source Data Warehouses are available [20]. Furthermore, even the proprietary Data Warehouse that we use in this work is available as trial [5]. Complementary, an online tutorial for Analysis Services 2005 describe steps in cube design (also applies to the 2012 version)[5] and web blogs cover plenty of information about advanced cube design and query topics.[6] Furthermore, we recommend a tutorial about advanced cube design with many-to-many relations [224]. We use the contents of the tutorial for cubes that we present in later chapters of this work.

Additionally, a vendor independent introduction about Data Warehousing is available in [137]. The book describes the topic on 233 pages and is recommendable as first overview about Data Warehousing. Advanced insights about technological foundations of Data Warehouses, including the MDX query language, are given in [146]. For an economic centered introduction, we recommend [109] that contains application and installation approaches for businesses and concentrates on the specific implementation of the SAP Business Warehouse. [143] offers a general introduction and discusses different approaches to realize the relational schema of a Data Warehouse. More technical insights about improving relational performance are given in [156].

[230] offers great visualizations for the cubes and explains in detail how the cube visualizations can be mapped onto data. In this context, we note that other visualizations than the ones we used are available and described in [57].

Additional literature covers specific application topics of Data Warehouses. [185] gets into details about reporting and provides examples and explanations for this use case. Additionally, other applications of Data Warehouses are strategic enterprise planning scenarios. Literature about indicator based planing with Data Warehouses [190] covers this area.

4.6. Software Cockpits

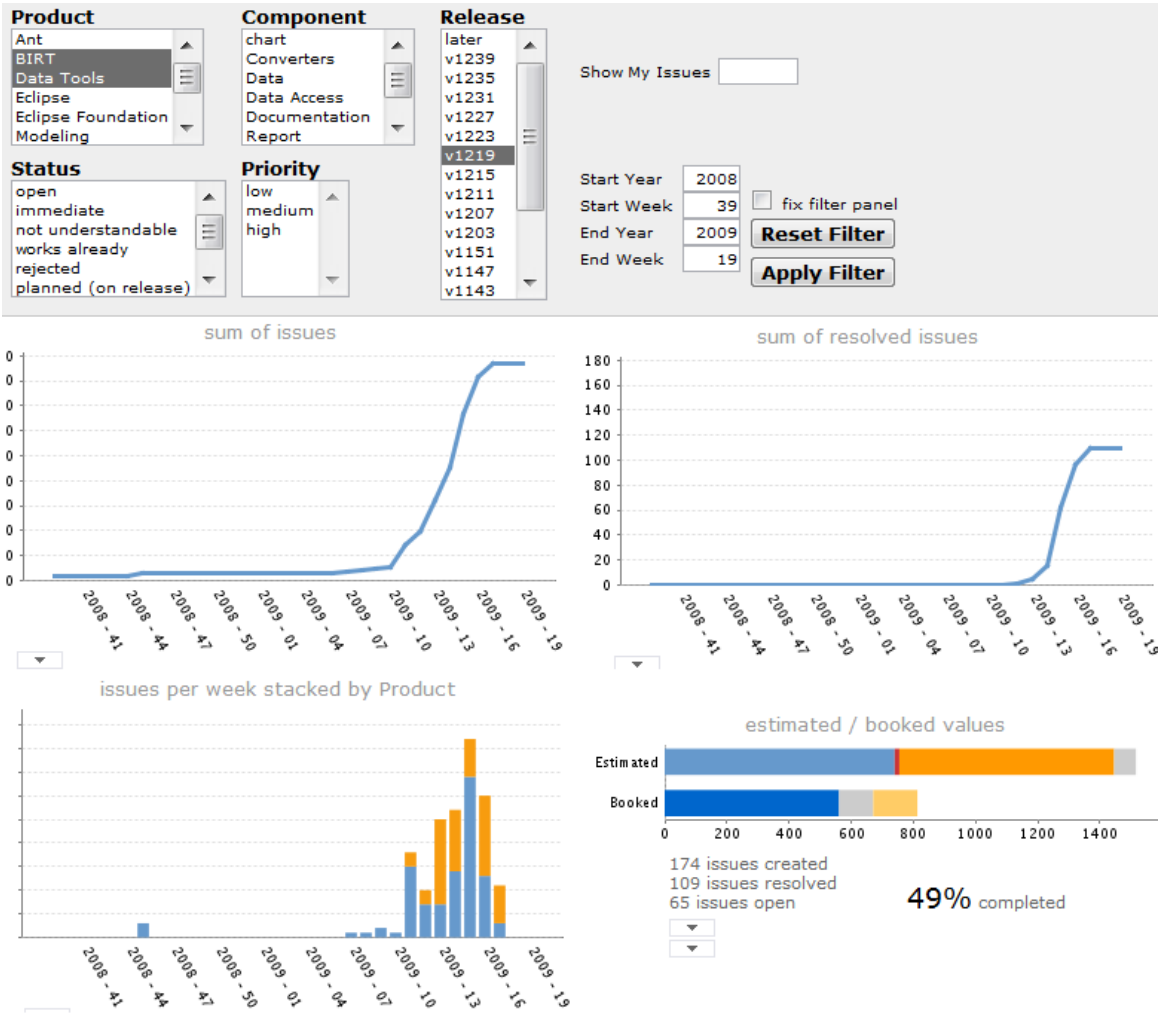
Before, we described Data Warehousing, its data models and operations to investigate data. Additionally, we gave references to further work about Data Warehouses. Right now, we describe an exemplary application of Data Warehouse technology in Software engineering.

[5] http://www.kodyaz.com/articles/how-to-create-olap-cube-in-business-intelligence-development-studio.aspx - 7.11.2012
[6] http://cwebbbi.wordpress.com/ - 7.11.2012

In software development, project managers, quality assurance and decision makers need to keep overview of a big amount of data. Currently, software development projects lack a holistic approach to provide relevant and filtered information for specific roles at a central point. In order to advance project steering, an approach is needed that allows reproducible abstractions of the relevant information to estimate the progress of an entire project. Thus, different work follows the goal to develop so and to introduce called management dashboards, i.e. software cockpits or software project control centers to enable a better overview over software projects. This dashboards show custom indicators about the status of a software development project and support controlling a projects business goals [119, 120]. The general vision of such a cockpit is similar to one in a plane that offers the relevant information for the captains and allows them to get all relevant information for decisions at one place [183].

We show an exemplary cockpit in Figure 4.10. There we can see some charts visualize different facts of issue tracking systems.[7] Furthermore, users can select certain properties, like the release version, a date range or a specific project. When such things get selected, the charts get adjusted for these specific properties.

Figure 4.10. A Software Cockpit Example



A software cockpit shows different statistics about issue tracking systems at one place. Different charts give indications about project progress. Users can drill down into different time frames and features of the statistics to gain insights.[8]

[7]For instance, Bugzilla (http://www.bugzilla.org/) or Jira (http://www.atlassian.com/software/jira/overview/) - 7.11.2012
[8]Source: Screenshot from http://softcockpit.scch.at/#up - 7.11.2012

However, like the prior shown cockpit, other researchers create similar cockpits that contain additional information. Another cockpit contains data about software tests. All this data can then be combined with drill down functionality to enable developers to investigate the package hierarchy downto the concrete member function where the specific metrics occur [154, 214]. Anyway, the main goal is to load data from the issue tracking systems and provide a dashboard for their data. An overview of those allows managers and developers to see the progress at one place and to create a more transparent development process [184, 153].

Internally, cockpits utilize Data Warehouse technology as follows. In order to provide analysis data from the issue tracking system is loaded into a relational schema and then loaded into a cube. On top of the cube queries are done an serve as data provider for the cockpit. By selecting the different features, the queries get refined and the charts reload for that refined queries. Therefore, software cockpits allow to use the toolboxes for reporting that come shipped with Data Warehouse technology.

Anyway, researchers created also more advanced cockpits then the shown one. Thereby they loaded data from additional sources into the Data Warehouse. As additional data aside the issue tracking information they loaded metrics about style issues[9]and the primary code hierarchy. The primary code hierarchy is the Java project structure from packages down to classes and their members what enables to go into the specifics of a project. Furthermore, they allowed extended configurations to specify which parts of the source code belongs to which layer. With this data it is possible to create cockpits where developers and managers can browse the code hierarchy and drill down into the details where exactly which style issues occur. Through configurations which source code parts belong to which layer, also computations about style issues in layers can be shown in a cockpit [214, 45]

Aside of the additional data, different cockpits for the specific role of developers and project managers were built and different visualizations for them were used. Developer specific cockpits showed more details and code related stuff and manager targeted cockpits showed rather abstract project information.

First evaluations showed that cockpits are useful in software development projects. One main advantage in contrast to other analysis tools was the integration of data out of various sources together in the cockpit. The centralized and role-specific views lets the researchers argue that cockpits hold benefits for the different roles. Quality problems were recognized clearer, but not necessarily earlier [45]. Furthermore, structured interviews of involved employees led to confident viewpoints about the positive effects of the introduced cockpit. Developers appreciated the benefits of the software cockpit and the prior raised concerns, before a cockpit was installed, were replaced by a positive attitude about the newly created transparency through the cockpit [155]. More detailed industrial applications and empirical evaluations indicate further benefits of a software cockpit. Thereby, a small panel with 20 people was investigated that does not offer any significance and just indications. However, in the investigated projects it was possible to detect project risks and plan deviations through a cockpit earlier. Additionally, also project risks and plan deviations that would not have found by traditional means were uncovered. In general, the cockpit was regularly used and people perceived a cockpit as positive, useful and easy to use [64].

Even though, software cockpits seem to provide benefits, they target mainly project management and the integration of data out of issue tracking systems. As code structure, only the primary hierarchy of source code was stored into the Warehouse what restricts the capabilities of current approaches. Cockpits cannot be used to investigate the relations that occur within source code and determine metrics like project couplings. However, the first investigations show that already current cockpits provide developer acceptance and hold benefits for managers. We conclude that an industry-wide adoption in maybe in lack of additional application scenarios that provide further benefits aside their current capabilities.

[9]Style issues scanners reveal quality indicators of bad coding style habits [212, 124, 225].

4.7. Summary and Conclusions

We presented and introduced Data Warehousing for the software engineering community and gave insights about its original and primary application scenarios within enterprises. We described that the technology offers means to investigate indicators, like revenues out of different perspectives and helps enterprises to control and plan their next steps. Furthermore, we described the main benefit that the Data Warehouse technology to serve as integrator for data out of various systems. We got into internals of the technology and presented the commonly used models that allow investigations with structured queries. Then, we gave examples of the multi-dimensional operations that are supported by common Data Warehouses. In order to credit related research and to provide a path to further reading about Data Warehousing, its application, performance tuning and internal way of operation, we provided references to literature. Lastly, we described research that already utilizes the technology within software engineering.

Nevertheless, aside of the general remarks of this chapter, we get now into a discussion what we remark for our work to advance software engineering with Data Warehouse technology.

When we credit our prior chapters about the multi-dimensionality of the concerns space and take into account that Data Warehousing deals with multi-dimensional data, we see that the current research about software cockpits does not credit this similarity of multi-dimensionality. Therefore, the usage of Data Warehouse technology within cockpits is currently not capable to allow detailed investigations within source code and utilize this similarity. Even though, reporting of code style issues and of issue tracking systems data to seems already to advance software development. Therefore, we state the following:

Building software cockpits that utilize the similarity of multi-dimensionality of Data Warehouse Data structures the multi-dimensional concern space would be a useful demand on a new the tool, since first indications show already the usefulness of software cockpits.

Out of this claim, we see the need to determine if the Data Warehouse data structures and technology can be used for the multi-dimensional concern space. Furthermore, we need to evaluate if Data Warehousing is suitable to solve the claims that we depicted in prior chapters to a new technique. As a solution for this, we present the next chapter, were we put all of those claims together and propose a new technique for software engineering, based on Data Warehousing.

Part II. Hypermodelling and Implementation

5. Hypermodelling

“Normal is regular. - Average, Medium.”
BMW. Mini commercial: "MINI: NOT NORMAL". 2012

“Here’s to the crazy ones.”
Apple. Tv Commercial: "Think different". 1997

This chapter shares information with:
"T. Frey, M. Gräf. Data-Warehouse-Infrastruktur zur Codeanalyse. JavaSPEKTRUM 6/2012,SIGS DATACOM GmbH. 2012"[99],

"T. Frey, Hypermodelling - A Data Warehouse Approach for Software Analysis. Magdeburger Informatik Tage (MIT). Mageburg, Germany. 2012."[100]

"T. Frey. Vorschlag Hypermodelling: Data Warehousing für Quelltext. 23rd GI Workshop on Foundations of Databases. Obergurgel, Austria. CEUR-WS. 2011."[102] and

"T. Frey, V. Köppen, G. Saake. Hypermodelling - Introducing Multi-dimensional Concern Reverse Engineering. In 2nd International ACM/GI Workshop on Digital Engineering (IWDE), Magdeburg, Germany, 2011."[101]

Abstract. Imagine the world of a software engineer and a software-project manager. Dozens of code analysis tools, 100 of metrics, 1000s of method calls, several of 1000s relations between the code elements, millions lines of code, and a whole internet of ready to use source code. How do we search for code and stick to certain quality measures. How are concerns arranged in the code? How do we explore the code? How do we extract facts for source code mining? How do we integrate different analysis tools together? Can we regard all the different viewpoints of our software?

We present the requirements on a new approach for software analysis. As solution for the prior mentioned challenges, we propose the integrative Hypermodelling approach that is based on Data Warehouse technology. In order to evaluate Hypermodelling, we define different application areas wherein it can be used. Now, software engineers, scientists and managers can use Data Warehouse infrastructures to investigate source code.

Developers can use Hypermodelling to do code search and to use development tools on top of it. Software architects can use it to clarify how, where and which parts of frameworks are used. Managers can use it for development progress control with reports. Scientists have an infrastructure with statistic computation capabilities directly at hand can do advanced ad-hoc investigations of source code. Additionally, future research can reveal specialized application scenarios and further details the use of Data Warehouse technology within software engineering. All in all, researchers and practitioners sit now at one table and can use the same infrastructure for different tasks. This kind of integration enables the rapid adoption of research results in practice.

5.1. Introduction

In the prior chapters, we described a motivation for new approaches in software engineering research. We illuminated different modularization techniques that allow multi-dimensional separation of concerns. Thereby, we also credited model driven approaches that provide abstraction and multi-dimensional navigation. We introduced the area of in soft Data Warehouses and described how there multi-dimensional data are handled. In this chapter, we combine the area of Data Warehousing with separation of concerns to derive the Hypermodelling approach. Hypermodelling is the main contribution of this thesis. We evaluate and describe it in more detail in the following chapters. Therefore, this chapter represents both; the foundation for the later shown details and also the tie-together of the previous chapters.

5.1.1. Contribution

In this chapter, we provide a new perspective for software engineering, due to the proposal to use multi-dimensional technology for source code access. We present the concrete instance to use Data Warehouse technology for that. This is a straightforward improvement, since the concern space of a program is multi-dimensional and Data Warehouses are built to handle multi-dimensional data. Furthermore, current code bases have an immense size and still grow. Data Warehouses are designed for big data and are perfectly suitable for today's large scale code bases. Hence, our approach advances the current approaches by the proposal of technology that is suitable to meet those demands. Additionally, Data Warehouses provide abstraction mechanisms, like aggregation, and multi-dimensional navigation, what can now be used within source code exploration. As addition and to provide a generic approach, we present reference architecture and a guideline, how source code can be loaded and structured within a Data Warehouse infrastructure.

Additionally, our provided reference architecture contains a information that is needed to apply Hypermodelling for an object-oriented programming language. In detail, this provides the answer to our first research question: "Is it possible to equal a. means to apply separation of concerns in object-oriented languages and their internal hierarchies and compositions, with b. data cubes, containing dimensions, dimension members, hierarchies and indicators of a Data Warehouse?" Through our approach, architecture and guidelines, now specific information is available how to express concern relations within Data Warehouses.

Furthermore, we provide first insights about our second research question:“If question former is true, how does an abstract framework for the implementation look like?” We answer this question and provide details how a whole Data Warehouse infrastructure can be used. We give a concrete guideline how and what for the different parts within a Data Warehouse can be used and provide detailed information about building cubes and schemata for source code.

We refer to the third question by presenting our framework: "If so, Is there a benefit of loading programs into a Data Warehouse and analyzing them with Data Warehouse technology?". We do so by referring to different specific application kinds that can now be created on top of Hypermodelling. Different instances of those applications are used to evaluate the benefit of the approach and get discussed in later chapters of this work in more detail.

We sum up the specific artifact contributions of this chapter as follows:

- The demands on a new technique for source code analysis
- An abstract framework for multi-dimensional source code access
- A reference architecture to use Data Warehouse technology for source code access
- A generic guideline for implementations based on Data Warehouse technology
- An overview of application scenarios

5.1.2. Reading Guide

In the following, we first derive the Hypermodelling approach. Then, we give insights about a holistic framework to realize Hypermodelling within a Data Warehouse infrastructure. Thereby, we show different application kinds that can be created on this foundation. At the end, we give a summary and draw conclusions of this chapter.

5.2. Deriving Hypermodelling

In the following, we give an overview of demands on a new technique out of prior described background. Then, we focus on Data Warehousing as solution for this problem and present the Hypermodelling approach. There, we describe what source code has in common with Data Warehouse Data Cubes. Then, We go into details and give a code centric sample of Hypermodelling.

5.2.1. Current Challenges

Ever since, Dijkstra started the thought of separation of concerns, researchers and practitioners argue about the "perfect" way to structure and modularize programs. When we recap that idea behind SOC in its very main meaning, we need to consider its original meaning. The main meaning is to "study the concerns on their own" and not about the main need for "perfect" encapsulation.[1] Clearly, separation of concerns by modules is a desirable goal for a good system architecture and it has plenty of benefits, but with modules we mainly address the encapsulation and not the original main claim of Dijkstra studying the concerns on their own. There is a difference in the physical separation in modules and to study concerns. To study concerns does not imply that concerns need to be separated into modules. Studying is a process of a developer that means that the artifacts belonging to a concern get regarded together. Encapsulation means that the elements of a concern get grouped together into a module. Therefore, to study means not to separate concerns into modules, it means just to look separately at them. Modules can help by studying, but they are no necessity to achieve this original goal. We see to study and module encapsulation as two different things, that both influence each other, but they are not the same.

In programming, we derive new functionality out of the composition of already existing functionality. Composition assembles concerns together. So, in many cases different concerns are composed together. Code fragments, like modules, are often belonging to different concerns at the same time. Hence, researchers focus mainly on the main point of encapsulation techniques. Therefore, we argue for the necessity to support to study a system from different concern perspectives and not only its physical separation.

Some researchers have already addressed the need of different views by introducing models as different projections of a software system. Similarly, other researchers propose solutions to separate concerns virtually by views. Their approaches are founded on a tool-based separation of concerns, without necessarily dividing source code into physically separated modules. Through tool support developers can study code with views of the different concerns that are originally are intertwined in the code.[2]

However, Orthographic Software Modeling and virtual separation do not reflect all parts of programs and the perspectives concerns to study code are manifold. Nevertheless, we share the opinion of this researchers and Dijkstra that one important thing in software engineering is the capability to study a software system out of perspectives of concerns. Out of this reason, we see the necessity to derive a technique that respects the multi-dimensionality of the concern space and allows to study a software system from various perspectives. Therefore, we define demands on a new technique by crediting key contributions of prior research[3] as requirements on a new technique. Thus, we demand that a new technique credits the following reality:

a. Different viewpoints should be supported to study concerns of a software from their perspective
   Stated in Section 2.3: “Separation of Concerns” (page 18)
b. A new technique should respect the interweaving of concerns in source code and the current limitations in programming, where modules belong often to different concerns at the same time.
   Stated in Section 2.4: “Limitations in Object-Oriented Programming” (page 20) and Section 3.3: “Virtual Separation of Concerns” (page 34)
c. The multi-dimensionality of the concern space should be respected.
   Stated in Section 3.2: “Multi-Dimensional Separation of Concerns” (page 29)
d. Concerns are often tangled and scattered with modularization techniques that are detectable.
   Stated in Section 3.2: “Multi-Dimensional Separation of Concerns” (page 29)
e. The technique should also be capable to consider frameworks as a perspective for software.

[1] See also Section 2.3: “Separation of Concerns” (page 18).
[2] See Section 3.3: “Virtual Separation of Concerns” (page 34) and Section 3.5: “Orthographic Software Modeling” (page 39) for further information about these techniques.
[3] See chapters: Chapter 2, Chapter 3 and Chapter 4

Stated in Section 2.5: “Frameworks” (page 23)

f. Abstraction and navigation is a key point in modeling, hence a technique should be capable to provide abstraction mechanisms and the multi-dimensional navigation of those.

   Stated in Section 3.5: “Orthographic Software Modeling” (page 39)

g. The technique should provide means to integrate data and analyze it from a central point and allow new facts to be “docked” together

   Stated in Section 3.6: “Source Code Analysis” (page 41)

h. Building software cockpits that utilize the similarity of multi-dimensionality of Data Warehouse Data structures the multi-dimensional concern space would be a useful demand on a new the tool, since first indications show already the usefulness of software cockpits.

   Stated in Section 4.6: “Software Cockpits” (page 60)

i. Considerations about investigations of modern code bases should respect their immense size.

   Stated in Section 1.3: “Motivation and Problem Relevance” (page 4)

In the following sections, we refer to the different points of this list.

5.2.2. Solution: Data Warehouse

In order to address the different demands at once, a kind of integrative technique is needed. In the following, the alphabetic letters refer to the list of demands on a new technique to match those with provided capabilities of our technique.

We describe Data Warehousing in Chapter 4. We assume that Data Warehouses meet the demands, because they:

1. Offer different perspectives of data (corresponds to demand a)
2. Provide a multi-dimensional viewpoint and navigation (corresponds to demand c and f)
3. Allow slicing the data out of records (similar to demand e, if framework elements are considered as records)
4. Enable abstractions through aggregations (corresponds to demand f)
5. Provide an integrative process that supports loading of different data (corresponds to demand g)
6. Offer tools for building cockpits for business scenarios (corresponds to demand h)
7. Big data processing (corresponds to demand i)

However, point b and d of the "demand list" in the prior section are not occurring in the current list of Data Warehouse capabilities, because those refer to the internal units and data representations of Data Warehouse. We propose to use *d* (detectable modularization techniques), to do the dimension - member- association in a Data Warehouse. Additionally, we propose to use *b* (interweaving of concerns in source code) as Data Warehouse facts. We explain this approach clearer in the following. First, we present an abstract approach and focus then on a concrete example with source code.

5.2.3. Hypermodelling

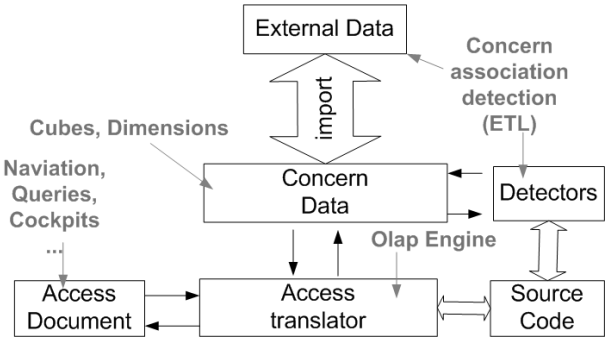
In the following, we first present an abstraction of our technique that meets the above mentioned demands. We describe our approach an abstract level, because different researchers specialize on concern detection methods that imply concerns that may be revealed by manual concern code associations or data mining techniques. In order to be open for such advancements, we present a generic approach that respects revealed concerns at a generic level. This way, we assure that our approach offers a transferable and adaptive framework. Hence, we show the general attempt to access concerns. In a later following example, we give insights about a concrete case with source code and concerns of code structure.

Therefore, we present the generic idea how use Data Warehouse technology in Figure 5.1. There, we see that the concerns within the source code are revealed by so called detectors. Detectors are

means like data mining, rules or manual associations that determine where in the source code which concerns are occurring. They store the revealed concerns in a schema, what we call Concern Data. Concern Data contains information about the occurrence of concerns in source code. In order to work together with concerns that are defined in external systems (e.g., Issue Tracking system), the detectors can use these data to associate it with source code. All together, the complete assembled concern data is a multi-dimensional association. This association is the dimensional structure of the data cubes within a Data Warehouse.

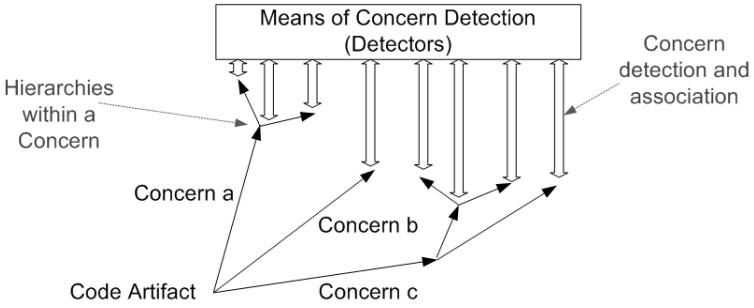
This way, the concerns in source code are accessible by Data Warehousing means. We divide the access in two parts. The first part is the Access Document that specifies a query for dimensions, e.g. concerns, and the perspective of those. The second part is the Access translator that realizes the direct concern access. The Access translator is the mean that translates the query and processes the desired data into the requested format to answer the request. Thereby, the Access translator uses the information (Concern Data) how the dimensions are structured and relate the Source Code. With this information, the Access translator assembles the desired Source Code artifacts together and generates a response for the request. In the following, we describe the relation of the different units to Data Warehouse components.

Figure 5.1. Hypermodelling



We show Hypermodelling as an abstract framework to classify and associate structures within Source Code with different concerns. Detectors reveal the concerns. The Detectors store the revealed code fragment associations within the Concern Data as multi-dimensional information. In order to generate responses for Access Documents, an Access translator processes the Source Code and the Concern data. The Access translator is capable to handle multi-dimensional operations for browsing and analysis of the Source Code. The descriptions above the arrows indicate relations to parts of the Data Warehouse technology.

Figure 5.2. Concern Detectors



We show the general idea that concerns of a code artifact can be revealed with detectors. The concerns can exist in different hierarchies and the detectors are responsible to detect the position within those hierarchies or the hierarchies themselves.

Right now, we match the abstract approach to concrete technology. All together, the Access translator is available in a Data Warehouse as OLAP Processor. The data cubes and dimensions

in a Warehouse correspond to the Concern Data. Detectors and integration of external information corresponds to data loading. We visualize detectors to associate code fragments with concerns in Figure 5.2. There, a detector is just a mean to detect the concerns of a code artifact and associates it to different concerns that are existing in different hierarchies and associations. We show with the different arrows that a fragment may belong to different points in hierarchies. The black arrows show the concern hierarchies and the broad ones show how the detectors sort the revealed concerns into these hierarchies.

All together, we stick the demands in the list in Section 5.2.1. Our proposed generic "Detector" mechanism generalizes claim *d* (detectable modularization techniques) and implies that different kinds of detection mechanisms may be used. Furthermore, we realize claim *b* (interweaving of concerns in source code) with a multi-dimensional arrangement of the Concern Data. Hence, Concern Data corresponds to the dimensions and the cubes in a Data Warehouse. Furthermore, this ratifies claim *g* (integrate data) and describes data integration. Additionally, we abstracted the dimensional access (*a*, perspectives) and multi-dimensionality (*c*) and the navigation (*f*) with the Access translator and the Access Document. Commonly such an Access translator is realized by an OLAP engine and the Access Document is an OLAP query. On top of this access, different viewpoints, dimensional navigation and reports can be realized with common technology that is normally shipped with Data Warehouses.

Out of this, we define Hypermodelling as a technique as follows:

Hypermodelling is the technique to structure and access concerns of source code. The access is done by an access translator that uses a multi-dimensional concern model. The access translator is an abstract machine that supports navigation, computation, aggregation, mining and any other kind of read, write or presentation operation on concerns. The multi-dimensional model is used to structure concerns and contains data about:

- Concerns defined within source code;
  AND / OR concerns that are specified by additional data, whereby this data contains a concern specification, associated with at least one region in source code
- At least one numeric indicator representation of at least one concern or a concern relation. Thereby, an indicator links one or more code fragments or modules to a concern or a relation
- At least one relation between concerns or code fragments,
  whereby the relation can be used to aggregate at least one sum of at least one concern relation;
  AND the relation maybe also a hierarchy

Additive, any mean of concern detection to retrieve concern indicator associations may be used.

Optimally, Data Warehouse technology is used to realize Hypermodelling. The access translator equals an OLAP Processor. The prior stated indicators equal Data Warehouse indicators. The Data Warehouse dimensions equal concerns. If multiple artifacts belong to a concern, e.g. dimension, those are the members of the dimension. The code hierarchies equal hierarchies within the Data Warehouse. The indicator concern association and relations in code equal facts.

All together, the right before described mapping is a Data Warehouse data cube and dimensions. This way, demanded access actions of Hypermodelling, such as navigation and code structure access can be done with means of a Data Warehouse.

Thus, Hypermodelling is a technique that supports multi-dimensional access and computations of concern associations. When we transfer this concern association to the area where concerns are encoded into modules, we recognize that concerns are associated with each other by module

compositions. Following the prior definitions, we can depict modules, representing a concern, to equal a dimension or a dimension member. If multiple modules belong to a concern, those are dimension members. The composition of multiple modules are facts that are tied together with an indicator. This can be just the indicator of '1' to specify an association between the modules exist. The hierarchies that occur within modules equal the hierarchies in a Data Warehouse.

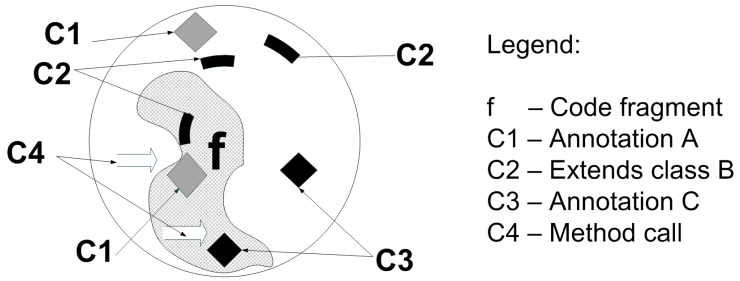
However, this explanation and definition is rather abstract. In the following section, we give a concrete example for the concern detection of a source code fragment.

5.2.4. Code Centric Hypermodelling Example

In order to apply the abstract technique we focus on concerns that are encapsulated and detectable through means of modularization. Hence, this section focuses on the concrete detection of concerns to give insights how the prior described detectors can work. In today's source code plenty of modularization techniques are used. A class or method module has different annotations, calls other methods and can extend other classes. All those modules, like parent classes and annotations are detectable by normal code parsing. Therefore, we see a program fragment as a unit that belongs to multiple detectable concerns and concentrate our effort on those.

The generalized view of the software fragment is shown in Figure 5.3. There, we see that the source code fragment **f** in the center belongs to different concerns at the same time; named as C1 to Cn. The different symbols express that concerns can be visualized with different means of modularization. We imagine that the diamonds are annotations, the black blocks are parent classes and the arrows are method calls. With this structural information in source code, the concerns and their associations detected automatically.

Figure 5.3. Concerns of a Code Fragment



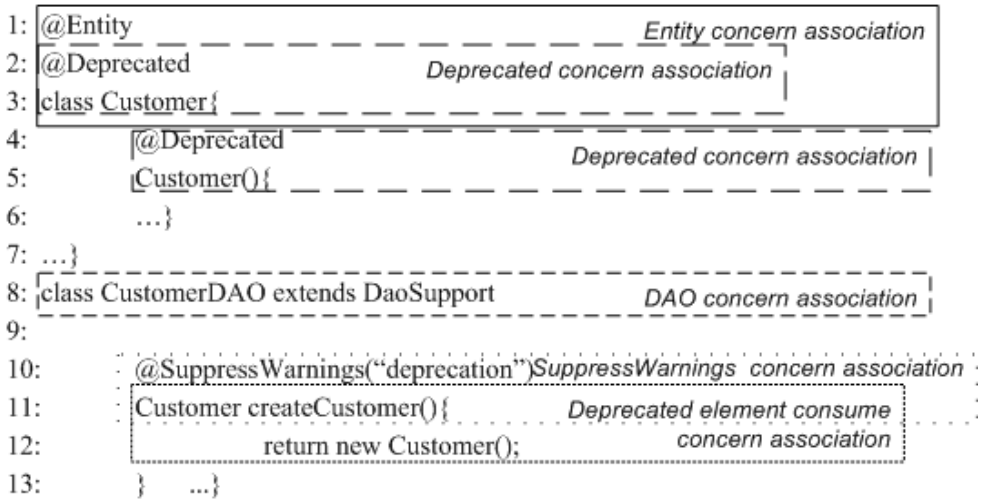
The circle symbolizes the code of a program and a specific fragment f. The different symbols (Cx) represent occurring concerns in the code that are encapsulated with different means of modularization. The code fragment f is affected by multiple concerns at the same time.

When we consider the diverse concerns, we also have to realize that they are located within different hierarchies. For instance, parent classes are located within their own inheritance hierarchies. Likewise, annotations are located within their package hierarchy. All of those concern associations and hierarchies are well-known and can be resolved out of the programming language structure.

In order to provide a more concrete example of our abstract framework, we show a sample source code fragment in Figure 5.4. There, the class Customer and a Data-Access-Object (CustomerDAO) are shown. The CustomerDAO class extends a helper class (DaoSupport), for table access. This is commonly done when frameworks are used. Both classes make heavy use of annotations. We show exemplary concern associations through the boxes. The elements are associated with various concerns at the same time. For instance, the Customer class is associated with the entity concern that belongs to the persistence part of Java. At the same time the Customer class is marked as deprecated. Hence, every programmer associates it with the concern that it should not be used any more by other code. Another example is the createCustomer method where compile errors about deprecation are suppressed. The createCustomer method is consuming the deprecated Customer and, therefore, a deprecated element consumer.

The slices in Figure 5.4 are detectable out of the Java language structure since normal means of modularization are used. Furthermore, we also see that other concerns can be detected through external information. We see an example of this, in line 11/12 where the consume of a deprecated marked class can be detected. In order to realize such an association, the compiler output can be associated with the source code. Out of Figure 5.4 and its detectable concerns, we derive our general idea in Table 5.1 as dependency matrix. There we can see that we use the relations within source code itself to express the connections within the code. Therefore, the table shows an alternative representation of Figure 5.4 and its associations. The rows are representing source code fragments and columns representing concerns. The columns show that the concerns are source code fragments themselves. Hence, the table shows the relations in the source code. The count of "1" indicates that a fragment is associated with a concern. For example the constructor of the class Customer is marked @Deprecated. Likewise, the rest of the table-listing associations can be done. This way, we can see that source code itself serves as code to be analyzed and as concern that can be used to build slices at the same time.

Figure 5.4. Exemplary Code Slices



We show a code fragment and its associations with difference concerns. We see that the same fragment belongs to different concerns at the same time. We call this concern associations concern slices or dimension.

Table 5.1. Source code as multi-dimensional Object of Concern associations

Element / Dimensions	extends DaoSupport	@Entity	@Deprecated	@Suppress Warnings	Deprecated consume
Customer	-	1	1	-	-
Customer.Customer()	-	-	1	-	-
CustomerDAO	1	-	-	-	-
CustomerDAO.createCustomer()	-	-	-	1	1

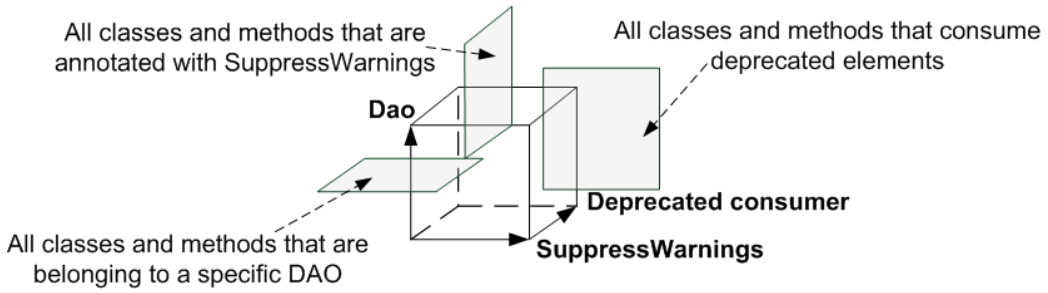
Out of the described associations of source code, we define: All the code associated to a specific concern is forming the corresponding concern slice. A concern is, like described before, similar to a dimension in a Data Warehouse. Therefore, we see that Customer and Customer.Customer() are belonging to the @Deprecated slice. Or CustomerDAO. createCustomer() is belonging to two

different concerns at the same time (@SuppressWarnings, Deprecated consume). This way, we credit the claim that a new technique should respect the interweaving of concerns in source code and the current limitations in programming (b).

Also, we can take advantage of the hierarchies that occur in source code and unite/aggregate associations along hierarchies. Aggregation among the hierarchies means that all members in a hierarchy are summed up to a higher hierarchy level. For instance, the aggregation among the Customer class and its members for deprecated annotations is the total amount of "2". We visualize the aggregation to all the associations occurring in the CustomerDao in Figure 5.5. There, we present the visual similarity to Data Warehouse cubes. The different slices indicate that a whole program can be viewed this way. All the fragments that belong to a concern form a slice together.

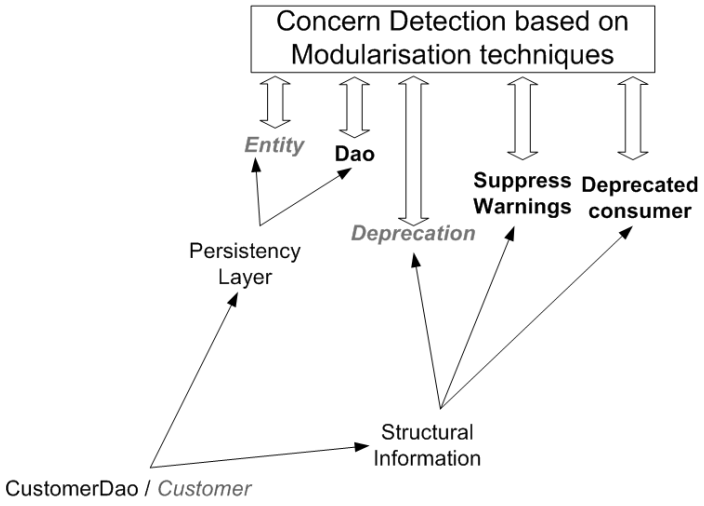
We present this slice visualization again to credit our prior mentioned attempt to use abstract detectors to classify source code in Figure 5.6. There we see that both classes are arranged into a hierarchical concern structure as example. Now, we imagine that this is applied to a whole program and all the program elements that belong to a concern get arranged in a similar concern structure. This structure is then the dimension data that is used to provide access and the usage of an OLAP processor. However, a detailed mappings of Java code to Data Warehouse cubes are described in the next chapter.

Figure 5.5. Source code Concerns as Cube



Exemplary dimensions of the prior shown CustomerDao class. The class is associated with different concerns at the same time that all form slices. This picture visualizes that all code fragments that belong to a concern are associated with the corresponding slice.

Figure 5.6. Classifying Source code with Detectors



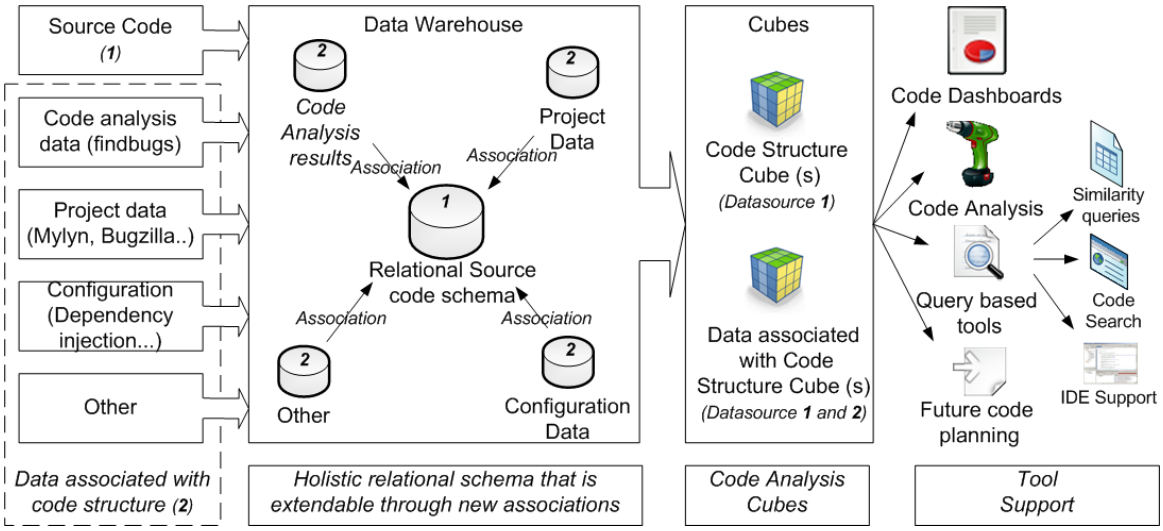
Exemplary detector based classification of code elements based on their means of separation. We show two classes and indicate where the detectors classify their concerns within the dimensional concern hierarchy through black and grey colors. We see that the detected concerns for both classes occur in the same hierarchies.

5.3. Hypermodelling Reference Architecture

We derived the Hypermodelling technique and provide an abstract approach, describing dimensional concern access to source code. In this section, we present a holistic reference architecture, showing details to a Data Warehouse infrastructure.

In order to use the whole toolbox of Data Warehousing technology, we abstract our application to a generic architecture framework that can be applied for the Data Warehouses of different vendors. We present this reference architecture in Figure 5.7. We show the different data origins for the Data Warehouse on the left. On top of the source data, we see the source code that is directly loaded into the relational schema, where it shows up in the center. The data beneath the source code has relations to the code and is likewise loaded into the relational schema. In the relational space is the additional data arranged around source code. The arrows visualize the relations to the source code. From the relational later the data gets loaded into data cubes. Thereby different computations about the associations are applied until a multi-dimensional cube model of the source code and associated data is available. Those cubes serve then as foundation to create different tools on top of them. Currently, we show the different kinds of tools that we present later in this work.

Figure 5.7. Hypermodelling Reference Architecture



The Hypermodelling reference architecture shows that data are extracted out of various sources and transformed into a relational model. In the center, we see the source code structure as central element. Data of the development process is associated with central code model. From there, the data is loaded into multi-dimensional data cubes that provide multi-dimensional concern based access to source code. On top of those cubes are different applications arranged, based on the access of those cubes.

As a matter of fact, we describe the architecture in more detail. In the following, we present the different parts of this architecture step by step. We start with the data source and explain the role of the relational model. Then, we go further to the data cubes. Lastly, we describe the different application kinds that can be built on the top of this infrastructure and conclude with a roundup.

5.3.1. The Relational Schema

We see that data of various sources are extracted into a Data Warehouse. We light out that not only source code is loaded into the Warehouse, but other data as well. The diverse data sources consist of the source code itself and of other data that is created within the development process, containing references to the code. We quote project data, code analysis results and test coverage as samples for this kind of data. However, we do not limit ourself to these data and since the central goal of

our reference architecture is to extract and integrate all the data that are useful for analysis together at one point. Therefore, the data can be seen as pure examples for other data that is coupled with the software development process.

In order to store and arrange the data in the relational model, the extraction can be done via various means. The simplest version can be code parsing and direct inserting. However, the insertion process can also be more complex and transform the data over various staging layers. The relational code schema and the details of the associated information may also be defined in a relational schema at various abstraction levels. One concrete schema may go into details like programming instructions, like "if-else" constructs. Others models may go only to the granularity of the method and the class level. But in general, the idea is always the same. The source code schema is arranged in the center and the other information is associated with it.

Out of the different granularity and diverse programming languages, our reference architecture works as meta- structure how the data associations are done. Additionally, one main reason for introducing a relational schema is to be compatible with most Data Warehouses that are commonly built on relational Databases. Therefore, the relational layer offers an intermediate addition that enables compatibility to common industry practice.

A developed source code schema enables integration and loading of data from various sources into the warehouse. Since the relational source code model is in the center, every kind of data that has a relation to it can reference it. This makes our approach extensible and enables further possibilities to integrate new kinds of data.

In order to provide a concrete instance of this central schema, our work provides a concrete relational schema for Java and creates this possibilities for Java.[4] In general, this approach can also be transferred to other programming languages with a similar structure and the general approach to have a code schema in the center defines the general step for the development of such a schema.

Within this context, we also refer to the development of concrete schemata in Data Warehouses. Together, with our later presented instance of such a schema in Chapter 6, we provide a concrete example how the knowledge to create the instance for the object-oriented Java programming language is applied. This keeps our approach generic and transferable. Also, our schema serves as example for other researchers that want to develop schemata for other programming languages and information association scenarios.

5.3.2. Code centric Data Cubes

Prior, we discussed the relational schema that serves as base to load data into the multi-dimensional cube structures. Through the arrangement towards a central code structure schema it is possible to load the data into multi-dimensional warehouse cubes and use the occurring relations in the source code and associated data out of the relational schema. Therefore, the prior described relational schema is contains the relational implementation of cubes. In this section, we focus on the general structure of the multi-dimensional cubes.

A cube shows a perspective of different relations of the hierarchies within source code and the data of tools. Samples are cubes for code structure and quality information. Such quality information can be style issues or anything else that is set into relation with the code structure. However, through the code structure schema the data of diverse tools is also set indirectly into relation. For instance, we imagine there is data about style issues and authors of code fragments. Through the code both of these facts have an indirect association. Therefore, we visualize two different kinds of cubes. One kind represents the code structure and others associate further facts with the code structure.

In order to map source code to data cubes, it needs to be divided into Data Warehouse facts and dimensions. We support to do this discrimination, by providing a generic guideline how source code can be arranged into facts and dimensions in the following. Our guideline is leaned on the principles and experiences that we made by the creation of a cube schema for the Java language. We provide this guideline, to offer a general set-up to create a cube model for a specific use case.

[4]See Chapter 6 for details.

In general, we recommend regarding source code and its modularization in the sense of cube structuring towards the principle how modularization techniques are used commonly in the sense of separation of concerns. Modularization was introduced to ease reuse, but today it is commonly used to categorize a certain set of functionality together. We also refer to the recommendation that separation of concerns often leads to the conclusion that one module should correspond to one concern on its own. Therefore, we see a module as kind of categorization technique. This categorization corresponds to the meaning of a dimension member within a Data Warehouse.

The facts in a Data Warehouse compose this dimensions members together. Therefore, we map Data Warehouse facts to compositions in programming. As a matter of fact, also hierarchies occur in a Data Warehouse. In source code, these hierarchies correspond to the structure that is commonly realized as compositional hierarchies. Examples for such compositional hierarchies are inheritance or call hierarchies. Additionally, programming languages offer special means to indicate hierarchies, like packages. Out of this reason, we cannot discriminate all hierarchies from normal facts and refer here to the capability of Data Warehouses to realize multiple cubes. Therefore, we have an additional mapping of compositions and normal hierarchies in source code to Data Warehouse hierarchies. All together, we note as generic guide:

Means in programming, like modules, that are used to categorize are similar to Dimensions in Data warehousing and should be realized as those.

Compositions in programming are similar to facts in a Data Warehouse and can be modeled as those.

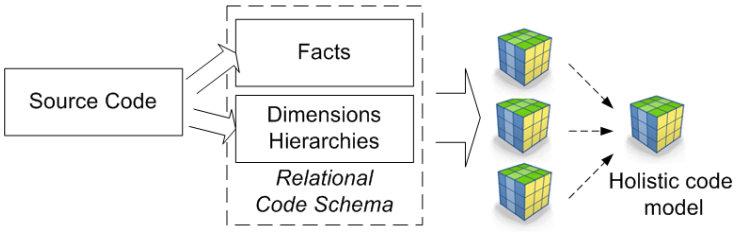
Hierarchies that occur in source code can be realized as facts. Also they can be realized as hierarchies of dimensions by using facts. The difference how hierarchies are modeled should be decided for the specific use case. In order to reflect such different cases, multiple cubes in a Data Warehouse can be created.

In the following, we provide detailed information, how this approach can be applied. Then we describe how the association of information with the source code structure is possible.

Code Cubes

The code schema consists of different means of modularization like classes or methods that are dimensions in Data Warehousing as we described before with the example (Figure 5.5). However, in source code plenty of compositions exists what means that source code consists out of plenty of cubes. In order to create a holistic model of source code the different facts, e.g., cubes, need then to be set into a relation. The relations are existing within the relational model and then those relations get arranged in the data cubes. In order to provide a generic approach for the cubes, we see the relational model as provider of facts, dimensions and hierarchies for the cube. Once a relational model is provided well known means like joins in the relational space can be used to provide those facts and dimensions directly to associate the data with cubes. We show the general approach to realize cubes on top of the relational space in Figure 5.8 and describe the picture in the following.

Figure 5.8. Holistic Code Cube



Code consists of different facts; therefore cubes need to be built to create a holistic model for source code. We see that different facts and dimensions are loaded out of the relational space into the data cubes.

In Figure 5.8, we see that the different facts of source code can be joined together in a holistic cube that expresses how all the different facts relate to each other. The different cubes split the source code in the into facts and dimensions. The facts express which elements relate to each other and hence represent compositions that occur in source code. The dimensions express entities. However, we do not show the hierarchies that occur in code. We spare this, because hierarchies are just another kind of relation between elements.

We mention that in Figure 5.8, the final model is named holistic, whereby Figure 5.7 names it code structure cubes. We did this on purpose, because in Data Warehousing it is uncommon to built to large cubes and often cubes are built just for one kind of specific analysis and then joined together into large "virtual" cubes. This is what we express with the holistic code cube.

We provide further information howto identify facts and dimensions in source code. Hence, Table 5.2 shows the guideline to discriminate source code into dimensions, hierarchies and facts. The questions describe which element in source code is what. They can be asked when cube schemata are constructed to determine how the element will be modeled. We clarify that the table has to be understood as guideline and not as all in all solution. The table was created to stick to the main abstraction level of modules and it is based on our experience in programming what we see as recommendable. We abstracted our experience in this guideline that it can be transferred to other granularity levels. The boundaries of facts-dimensions-hierarchies and compositions-modules is fuzzy and it is up to the specific modeling case and business goals how sharp the boundaries are finally set. Our guideline provides a foundation that can be sharpened for specific needs.

Table 5.2. Dimension and Fact Mapping Guideline

	Facts	**Hierarchies**	**Dimensions**
Question	Is something composed together? Is likely the association of elements to be altered?	Is it a cyclic fact? Can the relation be cyclic? Is recognized as hierarchy by developers?	Is it an Entity? Is it used as a unit that should provide a clear way of operation? Does it express how elements relate to each other? Can it be used for information hiding or abstraction?
Typical characteristics	Typically a M:N relation or 1:N. To change often. Compositions of elements.	Used to express specialization. Shares often also a duality with facts and a hierarchy of dimension members. Typically also grouping mechanisms that act as 1:N relation.	Instances are used for categorization and to define logical units that can stand alone. Used to group a certain set of functionality together. Relations that specify the how elements are composed together.
Change frequency (time)	Regular	Rare to Regular	Rare

Note, we also did some considerations to define facts as weak and dimensions as strong cohesion between elements. Then, the cohesion could have been measured over the evolution time of a software system to determine the facts and dimensions by machine means. However, there are many obstacles to achieve something automatic like this and we kept it aside for future research. Therefore, we decided to stick to the approach to present the guideline that we use in the technical implementation.

One central point in our guideline that decides about the granularity of the cube implementation is time. We introduce it in our guideline to provide an indicator that we see as necessary to sharpen the questions for a specific modeling scenario. Deciding which elements alter and move within their associations at a certain time frame impacts the modeling and the analysis vantage point. For instance, the frequency of a class that is moved from one package to another package can be considered from a monthly or yearly perspective. Imagine, a class is moved into the core

package within an application and afterwards into the utilities package. This whole movement happens within three months. From a monthly perspective the usage frequency is not higher as from yearly perspective. Therefore it is crucial to adapt the modeling to the regarded time interval. We recommend, that for a chosen time interval, the facts, hierarchies and dimensions are discriminated by the change frequency. We also emphasize, that modeling gets more complex as the interval gets smaller. Smaller intervals mean to model more associations and more granularity is reflected in the final model. Therefore, our guidelines and questions to determine the identification of dimensions, hierarchies and facts is a generic approach that need for specific cases to be sharpened by electing the corresponding time frame as vantage point. We provide short samples of the guideline application of Figure 5.4 in Table 5.3 for clarification, what we describe in the following.

Table 5.3. Guideline Application Sample

Meta description	Instance Samples	Sample Line	Mapping to	Change time horizon	Typical characteristic	Corresponding Question
Annotation at an element	@Deprecated Customer class association	1-3	Facts	Regular	Many Classes have many times the same Annotation	Is likely the association of elements to be altered?
Method calls	CustomerDao. createCustomer calls Customer. Customer()	11, 12	Facts	Regular	M:N, often to change	Is something composed together?
Method class association	CustomerDao. createCustomer()	11 - 13	Facts / Hierarchy	Some regular, others rare	Often to change, 1:N; Grouping	Is likely the association of elements to be altered?
Inheritance	CustomerDao extends DaoSupport	8	Hierarchy	Rare	Used to express specialization	Is recognized as hierarchy by developers?
Inheritance directive	extends, implements	8	Dimension	Rare	Relation that specifies how elements are composed together	Does it express how elements relate to each other?
Type definitions [a]	class Customer{...	3	Dimension	Rare	Logical unit	Can it be used for information hiding?
Methods	Customer. Customer()	5	Dimension	Rare	Logical unit	Is it used as a unit that should provide a clear way of operation?

[a]Classes, Interfaces, Annotations, Enums

The example in Table 5.3 shows the guideline applied. For the shown case, we assume a short time horizon where mainly compositions like method calls and similar change. We can imagine this time horizon to be the first creation of a trivial and simple application within one month, where no large scale architecture refactoring takes place. The table shows that sample depicted facts which consists out of the method calls and the annotation association to a class. We can imagine that this associations change a few times. As hierarchies we show the package hierarchies and calls and their member associations. We see that the member association is also a fact at the same time and its depending. Like described above, the instance of this duality has to be a fact and also a hierarchy, depending on the vantage point. As dimensions, we show the different entities, like classes. Also, we can see that the inheritance kind is realized as dimension. The reason for this is that the inheritance kind expresses the meaning of the relation between a class and a super type. If the supertype is an interface the relation follows a different meaning compared to extending a class.

All together, we showed code structure can be identified as facts, hierarchies and dimensions. In the following, we built on this and describe how identified facts and dimensions can be associated with further data.

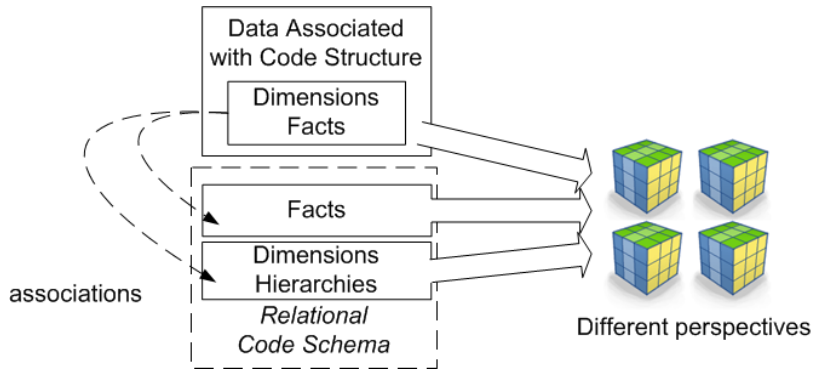
Code Structure Associations

Before, we described how the internal relations in source code can be realized in data cubes. Now, we show how further data can be associated with code in these code cubes.

In general, the attempt is the same like before to determine dimensions and facts. The difference is that just that the additional data is associated with code structure elements and association through new facts that relate. This way, the data relations to code elements would be arranged to the holistic code model like it is done for the various facts in code itself.

Figure 5.9 shows further data that associates with code structure. Facts and Dimensions of associated data relate to the additional dimensions and facts of code. This results in the possibility to create different cubes. Each of those cubes may cover a few dimensions and facts of the code structure and the associated data. This makes it possible to load different relations to code structure and associated data into cubes that represent different perspectives for further exploration. Like described above, here it would also be possible to create a holistic cube with all code structure. However, we recommend that smaller cubes are created, because they are easier to handle.

Figure 5.9. Additional Data Associations and Cubes



Associations of additional data with the code structure are extending the relational and multi-dimensional model. The center of the associations is the code structure. Out of this different data cubes can be created that represent different perspectives and starting points for investigations.

In order to provide a specific guideline how this data arrangement can be realized within a Data Warehouse, we present Figure 5.10. There, we show a cube for code structure with one fact association and two different dimensions (Code Structure Cube). This cube is then taken as foundation to "Dock" associated data to it. We show the "normal" code dimensions and facts in the "Code Structure Cube". Those are called Code structure Dimension what indicates dimensions for Types, Methods or any other element that occurs in source code. Those dimensions are associated with other code elements by the "Code Structure Fact" column "Fact-Dimension-Association". Hence, the upper left "Code Structure Cube" shows parts of the source code that is represented in the holistic code cube that we discussed before.

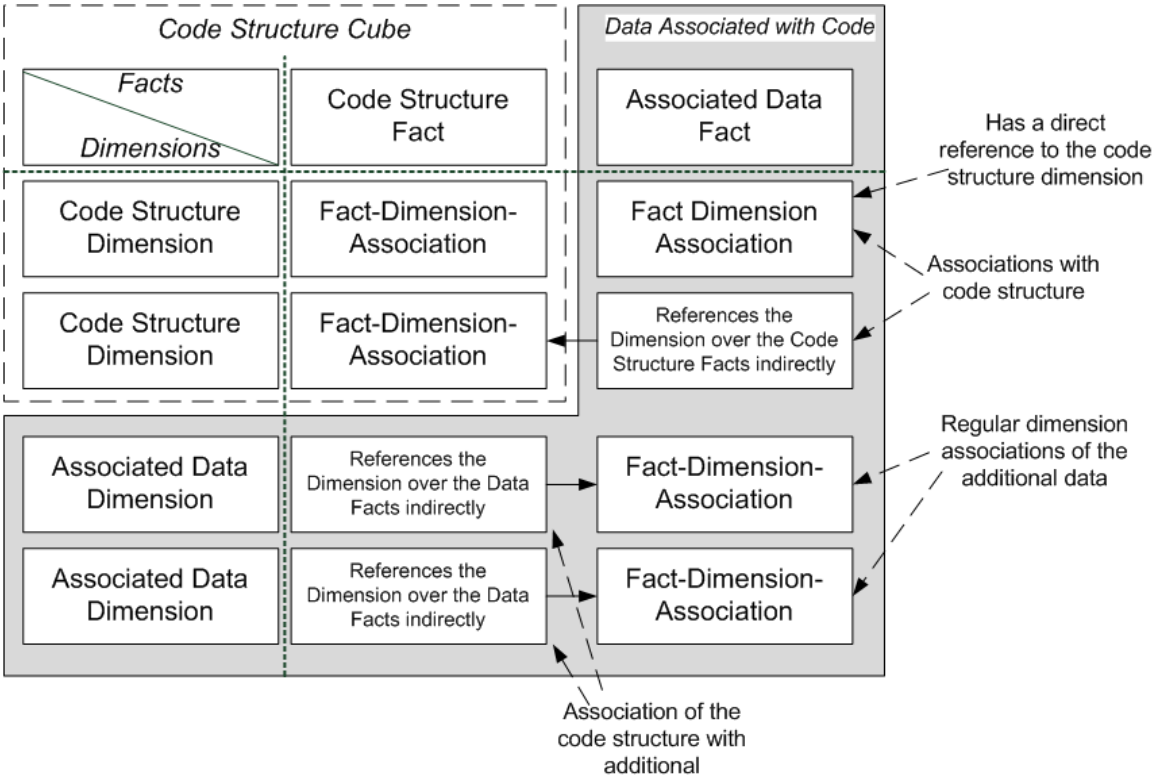
The docked associated data in Figure 5.10 consists of facts and two additional dimensions. In the upper right corner, a direct association with the dimensions of the source code structure exists. It is important that at least one element of the code structure also occurs in the additional data, since, logically, at least one connection is needed to relate the data to the code structure. The dimension beneath has no direct relation to code structure and the association is made by a reference to the facts within the code structure. We show this by an arrow to the left.

The technical background that a dimension can reference the code structure cube is as follows: Through the dimension that has a relation in the code structure cube and in the associated data a

common element, like in the code structure cube exists. The relation of these two dimensions within the code structure cube is known and can therefore be computed. This relation is likewise the same for the associated data. Hence, the computations of the code structure cube can be referenced.

Vice-versa is the relation of the additional associated data dimensions to the cube structure done. Here, we also take advantage of the dimension that occurs in both facts. Therefore, a connection of the code cube elements can be done just via a reference. Again, the computation works through the direct relation of the associated data facts to one of the code structure facts.

Figure 5.10. Additional Data Associations in Cubes Computations



The figure shows how further information can be related to the code structure. The computations of the relations are done by a dimension that occurs in both facts and by references to the facts.

We provide an instance of relations of code to associated data in Table 5.5. There, the deprecated consume slice is taken as sample of associated information. We imagine this information is available through compiler output and references to the method where the deprecated consume happens. This method associated can then reference the original method dimension of the code structure cube. The method dimension is associated with both member tables and can be used as discriminator.

We present an concrete instance of this associated data for Figure 5.4 in Table 5.4. There, we can see that such associated data may not only contain new dimensions, but also additional information. We show that the line number and defect rating can be specified. Indicators, like defect ratings can now be imagined to be used for computations.

Note, the same method can be used for advanced concern associations, too. For instance, one application can be to apply this association technique for concerns that have been revealed through concern mining techniques.

This way, means of different defects can be computed. A mean defect rating can be computed for types, packages and other elements of the code structure. However, different kinds of supplemental information and computations are possible. All together, the most important thing is that the relations to the code structure can be expressed and keep our approach extensible for different data.

Table 5.4. Deprecated consume Associated Data - instance with indicators

Compiler Facts			
Method association	**Defect-Dimension**	**Occurrence Line**	**Defect rating**
CustomerDAO. createCustomer()	Deprecated Consumer	12	3

Table 5.5. Deprecated consume Association Computations

Facts	**Method-To-Type association**	**Compiler output (Associated data)**
Fact references: Dimensions	**Dimension reference to Method and Type**	**Dimension reference to Method and Deprecated Consume**
Method	Facts reference dimension directly	Facts reference dimension directly
Type (class, interface)	Facts reference dimension directly	Reference to associated data facts
Deprecated consume	Reference to associated data facts	Facts reference dimension directly

5.3.3. Applications

Before, we described how the data cubes can be created. In this section we show different kinds of cubes that can utilize those data cubes of source code structure and associated data. The corresponding part in our reference architecture in Figure 5.7 is the tool area. There, we propose different applications: Reporting, mining, investigation and the evolution of a program. We propose these different areas, in order to verify that the areas that inspired the creation of Hypermodelling can also benefit from our efforts. However, we do not stick to specific samples here and focus rather on complete application areas to demonstrate the generic possibilities of our reference architecture.

In the following, we describe the different areas briefly and go from top to bottom. The corresponding sections are named like the descriptions in Figure 5.7.

Code Dashboards

One application on top of the data cubes are ready-to-use dashboards for source code investigations. In such cockpits various charts and figures can be shown to explore the code structure. For instance, can style issues and method calls be shown together as pie and bar charts to give users an indication about a potential correlation of those. However, generally Data Warehouses offer here ready to use tools sets to create such cockpits for data cubes with little effort for various scenarios. We propose to create cockpits structure relations of interest that need to be investigated regularly. Such can then be used to have an easy and integrated ad-hoc access to the desired data available.

Code Analysis

Another application area are code analysis tools. Often Data Warehouses offer integrated mining functionality that may now be used on top of the cubes. Additionally, Data Warehouses offer OLAP interfaces to explore the data relations analytically with queries. This enables users to explore and dig into a code base with structured queries. Hence, our application area proposal is to create tools for analysis on top of Hypermodelling and to use it for investigations of the relations within the code base.

Query based Tools

The prior mentioned OLAP interface leads to the third kind of applications. Query based tools can be built to navigate and explore the source code structure. In general, multi-dimensional queries offer now a base on which new tools can operate aside pure query and analysis investigation. One application is to query for similar code artifacts. Another is to create a code search engine that operates internally with queries. Lastly, also query based tools can be integrated within the development environment to allow queries therein. Therefore, we propose to develop tools, working internally with queries.

Code planning

As the last application kind, we propose future concern planning with Data Warehouse technology. Planning applications are quite popular within businesses to measure if a company sticks to the future plan. We propose to transfer these approaches with Hypermodelling to enable the planning of future code structures that are desired.

5.4. Summary and Conclusions

We described the key contributions of prior work and used them to state requirements on a new technique. We presented Hypermodelling as response. Hypermodelling provides concern based source code access. This makes it possible to represent concerns and their relations and facts and dimensions in a Data Warehouse. This way, we provide the answer to our first research question: "Is it possible to equal means to apply separation of concerns in object-oriented languages and their internal hierarchies and compositions, with data cubes, containing dimensions, dimension members, hierarchies and indicators of a Data Warehouse?".

Succeeding, we got into more details how Data Warehouse technology can be applied and how source code can be loaded into Data Warehouse cubes, by digging in into the details of the second research question: "If so, how does an abstract framework for the implementation look like?". We presented a framework and examples how multi-dimensional representations of source code with data cubes can be realized and how the whole Data Warehouse architecture can be used. Furthermore, we provided a guideline, containing further information about the association of external data with code structure.

Now, Data Warehouse technology can be used as integrative component to analyze the source code. Additionally, we proposed various application kinds on top of the Hypermodelling data cubes. Those applications indicate the answer to our third research question: "And is there a benefit of loading programs into a Data Warehouse and analyzing them with Data Warehouse technology?". In detail, the various application areas in code analysis, code dashboards, IDE tools, code search engines and future concern planning indicate further benefit.

However, the wide application range on top of Hypermodelling shows that further evaluations are needed. Therefore, we use the next chapters and provide a concrete implementation and use it to study the properties of a Hypermodelling. Therefore, we study and evaluate an implementation and different application scenarios.

Hence, in the next chapter, we apply Hypermodelling with a Data Warehouse infrastructure and provide an exemplary implementation for Java. Researchers can use this together with our in this chapter defined framework to derive implementations for further programming languages. In the then following chapters, the focus is set on the different application areas of Hypermodelling, wherein we demonstrate the capabilities and advancements. Finally, we present then the complete comparison of related applications and a discussion in Chapter 12.

6. Hypermodelling Technology

“Our races, united by a history long forgotten and a future we shall face together.”

Optimus Prime. Transformers: Revenge of the Fallen. 2009

This chapter shares information with:
"T. Frey, V. Köppen, G. Saake. Hypermodelling - Introducing Multi-dimensional Concern Reverse Engineering. In 2nd International ACM/GI Workshop on Digital Engineering (IWDE), Magdeburg, Germany, 2011."[101]

Abstract. Hypermodelling reveals the possibility to load code structure into the multi-dimensional models of a Data Warehouse. How is this done with a concrete Data Warehouse? How are the models for Java looking? Which indicators are used to represent the code structure? These questions might be asked by a researcher that read about Hypermodelling. In this chapter, we provide details about a relational model for the Java language and describe different cubes on top of it. We show an example how additional information can be associated with the models. Furthermore, we present the known shortcomings of the current implementation and give a discussion about the implementation complexity properties. Now, researchers have a template for Hypermodelling and can transfer it to other programming languages. Furthermore, they can use the shortcomings of the current implementation as starting point to advance our implementation.

6.1. Introduction

In the chapter before, we introduced Hypermodelling and gave insights how Data Warehouse technology can be used to represent source code structure. In this chapter, we show insights about a concrete Hypermodelling implementation for the Java language. The following chapters utilize this implementation in different scenarios. Therefore, this chapter represents the technical heart of this thesis and serves as exemplary implementation of our approach.

6.1.1. Target Audience

This chapter targets a technical audience that wants to know details how exactly Data Warehouse technology can be used to represent source code. It proofs that our framework is no research ivory tower and can be applied in practice. However, this chapter targets readers that have a certain degree of knowledge about the creation of Data Warehouse cubes with multiple fact tables, relational schemata and the Java programming language. Therefore, we point here to the discussion at the end of the chapter that wraps up the achievements and gaps of our implementation without deep technological insights. This enables the non-technical reader to gain some overview about our implementation. Additionally, the following chapters built on the outcome of this chapter, but do not demand to understand the technical details of this section. This way, we ensure that the rest of this thesis is readable without advanced technical Data Warehouse knowledge.

6.1.2. Contribution

In this chapter, we provide technical details how to apply the Hypermodelling approach for Java. We present concrete information how Data Warehouse can be used to store Java programs in it. This is a straightforward study of the application abilities of Hypermodelling and proofs that our approach is feasible with real world Data Warehouse technology.

Furthermore, we provide an example how further data can be associated with code structure what supports the claimed integrative abilities of our approach. Additionally, we provide a detailed discussion and insights about shortcomings of our implementation. All together, the

implementation in this chapter represents an evaluation of the static properties of the implementation of Hypermodelling. The information technology artifacts that we present in this chapter are the following:

a. A relational schema for the Java programming language structure and detailed explanations why the model is arranged the way it is
b. Four multi-dimensional cube models for parts of the Java language, their computation rules and exemplary examples of the outcome of those computation rules
c. A holistic multi-dimensional model for Java in two different graphical representations
d. A multi-dimensional model and example, how Style Issues can be associated with the Java Models in a multi-dimensional way. This serves as example to associate additional data with our model.
e. A discussion about the shortcomings of the provided models

6.1.3. Reading Guide

First, we classify the different sections of this chapter. Then, we present the relational model and discuss it. Following this model, we present the cubes and their computations. Thereby, we show indicator table computations with samples of source code to give impressions how code is finally represented in a Data Warehouse. Afterwards, we present a holistic model for Java that contains all the prior discussed cubes. In order to present details how further data associations with our model can be done, we present an example cube with Style Issues. That following, we get critical whereby we illuminate and discuss the static technical properties and the shortcomings. Lastly, we give a summary and do conclusions.

Aside of the general reading guide, we use the same strings that we use in figures and in tables in the text, when we refer to elements of those. This means, that when we refer to a "Field" in a table, we write Field in the text and use a capitalized first letter. This ensures that the reader can map the elements to the figures and tables.

6.2. Framework Application

In the last chapter, we described Hypermodelling and presented a generic framework how Data Warehouse technology can be applied to represent source code structure. In this chapter, we use this framework and present a technical implementation based on it. Therefore, we give deep technical insights how the models are structured and computed to have a reference and implementation.

We present the result of our efforts to derive a relational model for the Java language. Thereby, we used inspiration about relational schema development and focused on the snowflake schema approach that we introduced in the chapter about Data Warehousing.[1] In order to study and test if our schema works, we developed a source code parser that transforms the source code and stores it in the relational schema. We do not describe the parser and the schema development, because we focus on the outcome of our efforts and present the relational schema. This relational model is shown in the following section (Section 6.3).

Another part of the Hypermodelling framework is the multi-dimensional model and the corresponding computations. Therefore, we get from the relational schema to the data cubes of source code and the associated computations that link the relational schema together with the different cubes. We decided to present the source code structure as multiple cubes step by step to give a better overview. Lastly, we present a holistic model that ties all the smaller cubes together.

Like said before, we split the source code cube model into multiple ones[2]and discuss them step by step. Afterwards, we follow up with a graphic representation of a holistic multi-dimensional model of the Java structure. The interested reader can find the corresponding computations in the appendix.

[1]See Chapter 4
[2]Again, we use DFM as graphical representation for the cubes. See Chapter 4 for further explanations.

However, one part of Hypermodelling is also to association of additional data with the code structure. In order to give an example how this is done, we present a section about the association of Style Issues with source code, by showing a dimensional model for this.

6.3. The Relational Schema

Like said before, one possibility to realize Hypermodelling a relational schema for source code. In order to provide a valid schema, we get inspiration from the Eclipse internal Java model that is used widely and maintained by a large community what indicates a certain stability and quality.[3] Another reason, aside the stability of the model, is to lean the model towards the Eclipse model to be open for a portability of the Hypermodelling technique into Eclipse.

Nevertheless, the Java model of Eclipse is actually not created to serve as a model for a relational database. Therefore, we apply modifications to obtain a relational database model. Out of this reason, our relational model can just be seen as inspired by Eclipse and not as a one to one mapping. By building the model, we also perceived that some relations in the Eclipse model are based on the Java language specification logic. These differ from logical viewpoints of a programmer. For example, instances of annotations are not linked with the definition of the annotation type itself. This means, the occurrence of an annotation is not an instance of its type definition. These type of “logical” gaps also make a challenge for the transformation of the model to a relational representation. However, we prefer that the relational model should represent reality in the programmers meaning and not the Java language specification.

The schema in Figure 6.1 and Figure 6.2 shows the relational representation, whereby all fact tables, connecting different dimensions together, are emphasized in grey. These fact tables are the source for associations of various dimensions. Through multiple fact tables it is possible to realize complex relations. Like a type (e.g., class) that has multiple members and also multiple types that are used as parents for inheritance. To ease comprehension of the schema, we divide it into two different figures. First a description of the different kinds of types occurring in Java and inheritance and annotations relations is done (Figure 6.1). That following, the members like fields and functions and their relations with types are presented (Figure 6.2). We discuss the two schema parts in the following sections.

6.3.1. Inheritance and Annotations

The schema in Figure 6.1 introduces relations between annotations, types, and inheritance. In Java, primitive (e.g., integer and boolean) and complex (Classes, Enums, Annotations, Interfaces) types occur in source code. These are realized with AbstractType, ComplexType, and the TypeClassification table. The TypeClassification indicates the kind of the type. The AbstractType is defined to have the possibility to have a common base for complex and primitive types. As it can be seen, the ComplexType table, representing complex types, references the AbstractType and this way, indirectly, the TypeClassification.

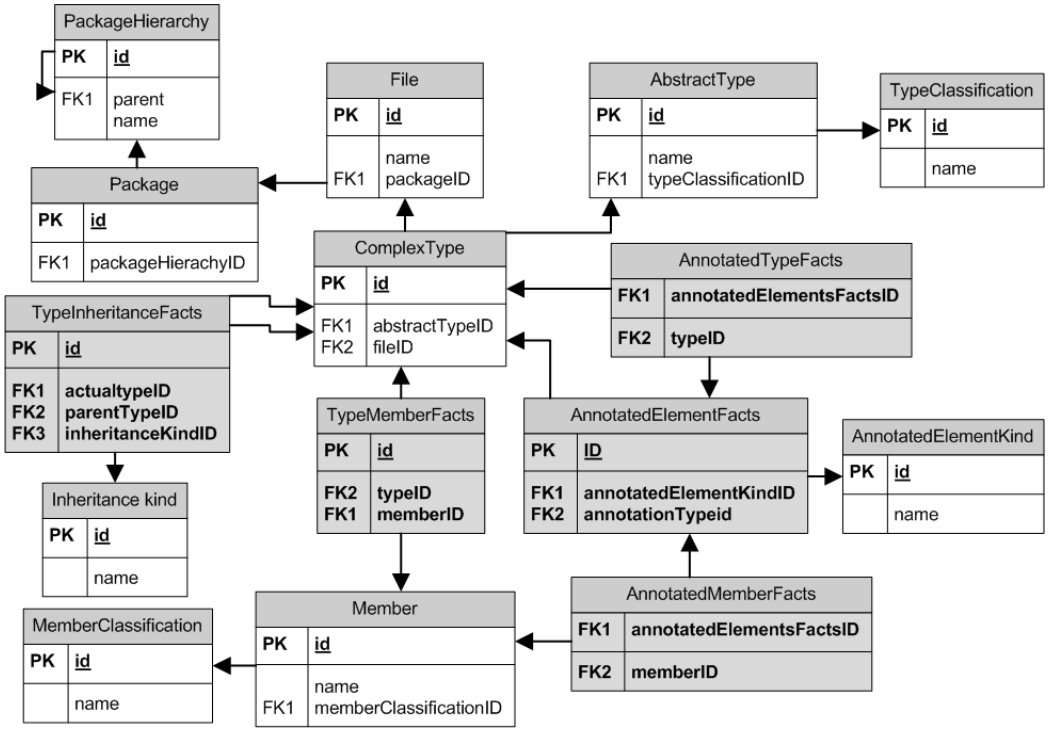
We note that AbstractType is also introduced to enable extensions of the model. Advanced associations between model elements can be done this way. For instance, method parameters can be defined to be of primitive or complex types. Through the AbstractType a generic mean is available to allow such associations that can be specialized.

However, a ComplexType can also have additional properties in contrast to a primitive type. Generally, a complex type is defined in a File that is, again, belonging to a Package which is furthermore a member in a taxonomic PackageHierarchy. Often different package roots for the same package exist in a project. Those package roots form a logical package together. In our table structure, we treat those roots as Packages which are unified together through the package hierarchy.

[3]For more information about the model see the Java Development Tools project of Eclipse that contains more information http://www.eclipse.org/jdt 11.11.2012

Thus, a File is not a direct member in a PackageHierarchy and the bonding to the Package table is done in between.

Figure 6.1. Relational Schema for Annotations and Inheritance



The figure shows the main code structure with annotations and inheritance. The fact tables are colored grey. The abbreviations of PK and FK mean primary key and foreign key. The arrows indicate the tables to which the foreign keys point.

As a matter of fact, a ComplexType can have multiple members. Such members are fields and methods. We realize this relation in a fact table. The approach of a fact table in between is chosen to credit reality; methods are often moved between complex types. A MemberClassification table is associated with the Member table to indicate the kind of the member. Hence, it is possible to extend the model for different member types, such as methods or fields, and mind their different properties. The implementation of interfaces and inheritance of classes are realized through the TypeInheritanceFacts table. We use this approach to avoid a self reference of the ComplexType table, because a type can implement multiple interfaces. To indicate the inheritance type, like implementing interfaces or extending a class, we introduce the InheritanceKind table.

We note that inheritance type can be derived from the classification of the parent type. Therefore, InheritanceKind indicating the kind of inheritance is redundant information. We introduce it regardless of redundancy, because explicit defined information is much easier to handle then implicit that needs to be derived by inference.

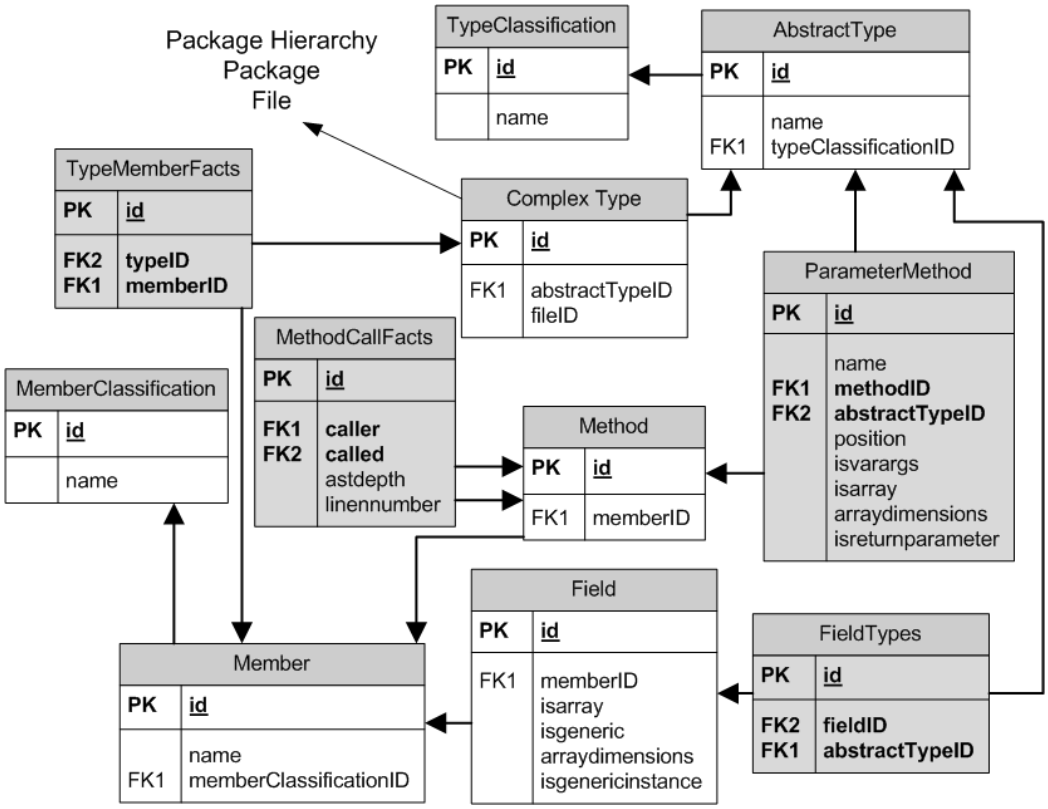
The AnnotatedElementFacts, AnnotatedTypeFacts, and AnnotatedMemberFacts show the annotated ComplexTypes and Members finally. AnnotatedElementFacts is used to associate an annotation, represented through a ComplexType itself, with a row in the fact table. The type of the annotated element is indicated by AnnotatedElementKind. AnnotatedTypeFacts and AnnotatedMember associate a fact with a complex type or a complex type member.

Again, we created the relational model for potential extensions. By using the abstract AnnotatedElementFacts, our model can be easily extended to add annotations for any other element, like method parameters.

6.3.2. Members – Methods, Fields, Calls

We complement Figure 6.1 with the detailed relation of ComplexTypes and their members in Figure 6.2. The arrow to PackageHierarchy, Package and File references the former explained relations in Figure 6.1, since both schemes are part of the same database and just visualized here divided to ease comprehension.

Figure 6.2. Relational Schema for Methods, Fields, and Calls



Complementing Table associations for Figure 6.1. This part of the schema concentrates on type members, like methods and fields.

The Field table references to the Member table. So, it is possible to extend the Field with attributes that differ from pure Member attributes. For demonstration issues, multiple attributes in the Field table, like isarray get shown as columns. Field has no direct name attribute, because it is derived from the Member table that contains the name of the member.

We recognize that the complete scheme, Figure 6.1 as well as Figure 6.2, shows nearly no attributes and the Field table ones are the first discussed. Expect of name, there are no attributes shown. However, the entire tables can be extended to contain other attributes. The main reason to cut the attributes out of the schema is: We prefer to show only a few ones to enable a better overview, since the main difficulty in schema development are relations.

A Field can have multiple types at the same time. We do so, to credit the occurrence of generics. Generics allow an association of parameterized types with a Field. Therefore, FieldTypes references AbstractType and Field at the same time. This way it is possible to associate multiple types with a field.

We note that generics can be modeled quite more complex if all rules how generics their parameterization can be realized would be respected. We decided to keep that possibilities aside and focus on the kind of generic types that a Field may have. Therefore, we neglect further details about generics for the sake of simplicity.

The reference of the FieldTypes to AbstractType manifests that Fields can also be declared to be of a primitive type. Through the AbstractType reference it is possible to associate both, primitive and class types with a Field. Either way, the model can be extended to support further specializations of the AbstractType. The FieldTypes table is marked grey to indicate that it represents a m:n association: Multiple Fields are associated with multiple Types and therefore a fact table is needed.

Similar to Fields, the Method table is defined. It references the Member table and can be seen as object-oriented derived from a Member. There is no necessity to name a method in the Method table itself, since this is already done in the Member table. A method has two main connections with other elements: First, a method can call other methods (Method) and second, it can have various parameters (MethodParameters).

MethodParameters represents each parameter type of a Method. Since parameters can contain generics, each parameter can appear multiple times in the MethodParameters table. [4] We unify method return values together with method parameters with a flag (isreturnparameter) that indicates if a parameter is return parameter or not. MethodParameters is also representing some kind of facts. Thus, the relation of methods and their parameters are modeled as m:n.

MethodCallFacts is a fact table that connects two methods. Exemplary, astdepth is added as sample for indicators. Astdepth expresses the nesting depth where the method call occurs. Caller is the method that calls the other one and called is the executed method.

Clearly, method calls could also contain parameter values. To provide a simple first implementation, such complex scenarios are discarded

All together, Figure 6.1 and Figure 6.2 describe a schema for pretty much of the structure within Java code. We realize java structure for inheritance, annotations, method calls, classifications and other logical structure of the associations that occur in code. In the following, we describe different cubes that can be placed on top of this schema.

6.4. Cube Structures

Right before, we described a relational schema for Java code, in the following, we present cube structures on top of that relational schema. We show dimensions, cube schemata and computational settings that are needed to process the cubes. Aus foundation in the relational area, we populate the prior discussed relational schema with data of a Java program load it into the cubes. As Java program data, to support verification and explanation, we choose a sample application that is available publicly and present code experts of it and how it looks like when loaded in the cube. However, in order to ease the comprehension of the following sections, wherein the different cube models are presented, we first explain typical dimensions in such cubes by examples in the following. Then, we present computations, aggregations, and concrete cube models.

The elected petclinic application is a demonstration of the capabilities of the spring framework that is applied widely in the industry. The petclinic application is described at its homepage.[5] The main reason to choose this demo application is: The spring framework is widely known as a reference for good application design. Mainly, the application is a layered web application consisting out of 31 Java files, containing application logic. It is making use of declarative transaction management, database access, and aspect oriented programming paradigms.

In Example 6.1, we show an excerpt of a class of the demo application. We list the corresponding dimensions in a Data Warehouse in Table 6.1. The source codes element are used as dimension member samples. We see that any structure of source code may be a dimension member.

[4]Like for the Fields more complex associations would be needed to represent the whole reality of generics.

Furthermore, we show that also artificial dimensions can be derived out of the source code structure, like it is done with the Member Classification. This dimension may be introduced to allow discriminations among different members (e.g., fields and methods) and still treat them as also belonging to the Member dimension. Furthermore, we see that artifacts of source code can be associated with different dimensions at the same time. For instance, this is true for AbstractType and Type. The reason is to enable users to use both abstraction levels and ensure integrity of the dimensional hierarchy.

Example 6.1. Excerpt of the petclinic application

```
//File: EntityManagerClinic.java
package org.springframework.samples.petclinic.jpa
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.samples.petclinic.Clinic;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

1: @Repository
2-: @Transactional
3: public class EntityManagerClinic implements Clinic {
4:      @PersistenceContext
5:     private EntityManager em;
6:     @Transactional(readOnly = true)
7:     @SuppressWarnings(-"unchecked")
8:     public Collection<Vet>getVets(-) {…
```

Table 6.1. Sample Dimensions

Dimensions	Description	Example Dimension Member
Abstract Type	Generic type: complex ones and primitives Table: Abstract Type joined to reference Types	EntityManagerClinic; PersistenceContext; boolean, int
Type Classification	The kind of the type	class, interface, enum, primitive
Type/ Complex Type	A common Java complex type	EntityManagerClinic; PersistenceContext; Clinic;
Type File	A file a type is located within	EntityManagerClinic.java; SuppressWarnings.class
Package	A package fragment	org.springframework.example
Package Hierarchy	Hierarchical structure of the packages	org;org.springframework; org.java
Inheritance Kind	The inheritance directive	implements, extends
Member	A member of a type	em, getVets
Member Classification	The kind of the member	method, field
Field	A field of a type	em
Method	A method of a type	getVets
Method parameters	Parameters and return types of a method	Collection<Vet>

After giving this sample of dimensions, we follow up directly into the cube structures. First, we describe a cube that contains a model and computations about inheritance. Then, we get into annotations what is succeeded by a cube about class fields. Lastly, before we get into the holistic Java cube in the next section, we present a cube with methods, method calls and method parameters.

The computation rules, in case of MS SQL Server, get called “dimension usage”. We mention this here, because often Data Warehousing solutions use different vocabulary and we want to spare the interested reader the search process, where related material to recreate our solution can be found. Therefore, we refer to literature [58, 224] that

uses the term "dimension usage" and gives details how the details of this chapter can be implemented with a MS SQL server. Therefore, all our computation rules apply for this vendor specific solution. However, by giving references to literature and all computations, researchers can transfer it to other vendor specific solutions.

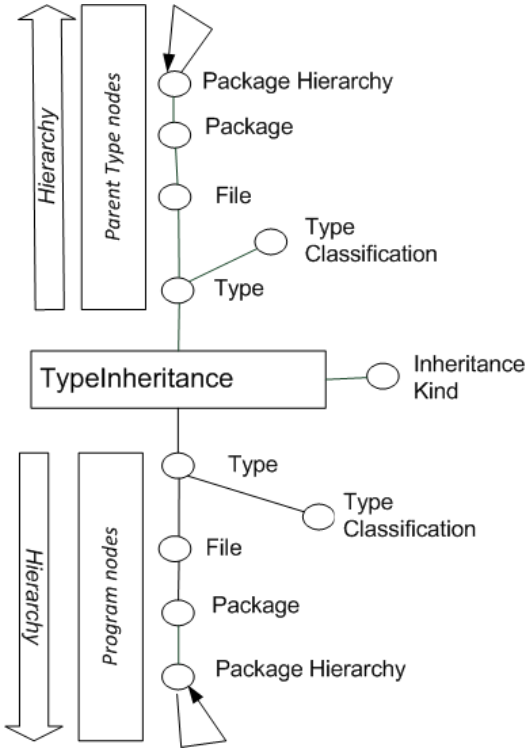
6.4.1. Inheritance

In the following, we present a cube for object-oriented inheritance of types (implementation and extending) and the associated computations. Then, we provide sample indicator computations.

Cube

In Figure 6.3, we present the inheritance measurement as cube. The Types (classes, interfaces) of a program are related to the Types (superclasses, superinterfaces) they inherit. The "children" of a superclass nodes of a program are located at the bottom, beneath the facts (Program nodes). They are the affected elements by inheritance. Parent Types can be divided into different classifications (Type Classification - classes, interfaces). Likewise, both Types - child and parent - can be arranged in a hierarchy from Type down to Package Hierarchy. The File dimension is the physical file wherein a Type is located. Such a File is located within a package. Often, modern programming environments allow having fragments of the same Package at different locations. Therefore, the Package Hierarchy is used, allowing browsing the hierarchy independently from its location.

Figure 6.3. Inheritance Cube



The cube shows relations of a Type (classes, interfaces) to its parent Types. The inheriting Type is classified as Program node, since it is the origin of the relation to the supertype. The dimensions are arranged hierarchically and an Inheritance Kind dimension describes if a supertype is implemented or extended.

Computation

Table 6.2 shows the computation directives to process the cube structure. We associate dimensions with facts through identifiers and column names in the brackets. Here, and in the following, we

always use this column names to indicate how we connect the dimensions to the corresponding relational table. One may wonder which tables we use to populate the dimensions members. Its straight forward: The dimensions are named somewhat similar to the corresponding table names. The dimensions are enlisted in the vertical. The facts tables that we use are always shown horizontally, as columns at the table. Those fact tables are named equally to the tables in the schema. In case we use joins of different tables, we indicate this by description.

Anyway, in the computations, we see "Reference" (short Ref.) dimensions. Those dimensions are used to reference an intermediate dimension/table for their association with the fact table. Thus, the attribute of the intermediate dimension and corresponding table is given in the brackets. Additionally, the reference is indicated by the coordinates of the table.

Directly associated with the facts is the Type and the Parent Type. All the Dimensions named with parent at the beginning indicate that they are belonging to the hierarchy of the type that gets inherited. We show that File, Package and Package hierarchy built a reference hierarchy to Type. Likewise, a hierarchy exists for the Parent Type.

Table 6.2. Cube Computations for Inheritance

Dimensions / Measures	TypeInheritanceFacts (A)
(1) Package Hierarchy	Reference Package (packageHierarchyID – A2)
(2) Package	Reference File (packageID - A3)
(3) File	Reference Type (fileID – A5)
(4) Type Classification	Reference Type (typeClassificationID – A5)
(5) Type	Type (actualtypeID)
(6) Parent Type	Parent Type (parentTypeID)
(7) Parent - Type Classification	Reference Parent Type (typeClassificationID – A6)
(8) Parent – File	Reference Type (FileID – A7)
(9) Parent – Package	Reference File (packageID – A8)
(10) Parent - Package Hierarchy	Reference Package (packageHierarchyID – A9)

Sample Measures

In Table 6.3, we present the source code listing from Example 6.1 as sample indicators. The *Object* class is an ancestor of *EntityManagerClinic*, since every class without direct parents inherits in Java from this class. Additionally, the *Clinic* interface is specified as supertype. The package hierarchy indicates that drill downs over various hierarchy levels are possible. These hierarchies exist for type and corresponding parent. The main dimensions Type, Parent Type, Type Classification, and Parent Type Classification are arranged vertically. This shows the orthogonality of the dimensions.

Table 6.3. Inheritance Sample Measures

			Parent – Type Classification	
			class	interface
		Package	java.lang	org. springframework. samples.petclinic
		Parent Type	Object	Clinic
Type Classification	**Package**	**Type**	*TypeInheritance*	
class	org. springframework. samples.petclinic	EntityManager Clinic	1	1

Additionally, Table 6.4, Table 6.6 and Table 6.5 present examples for different aggregations. Table 6.4 shows the total count of Parent Types. Table 6.6 shows the abstraction to the Classification

dimension. Finally, Table 6.5 shows the abstraction to the Package Hierarchy. Clearly, all this abstractions can be combined to investigate all kinds of relations.

Table 6.4. Total Type Inheritance

Type	Parent Types
EntityManagerClinic	2

Table 6.5. Parent Count split into their Type Kind

	Parent Type Classification	
	class	interface
Type Classification	*TypeInheritance (A)*	
class	1	1

Table 6.6. Aggregation to the Parent Package

	Parent Hierarchy	
	org	java
Hierarchy	*TypeInheritance (A)*	
org	1	1

6.4.2. Annotations

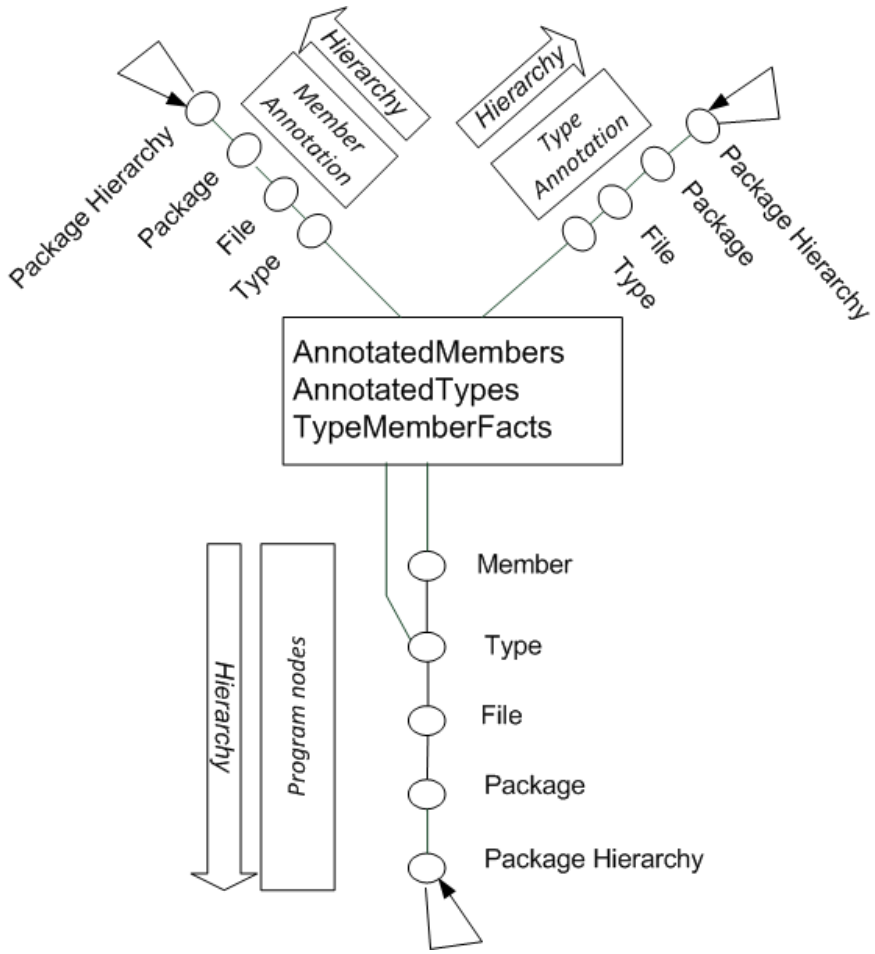
The second cube describes annotated elements in source code. In order to realize this kind of dimensions in a cube, we used a join of the AnnotatedElements table and the AnnotatedMembers table of the relational schema (see Figure 6.1 and Figure 6.2) to get a simpler cube model. Like before, we present the cube, then, the computations and stick, finally, to the indicator examples that represent the cubes relations.

Cube

Figure 6.4 shows the graph visualization of the cube. Like before in the inheritance cube, we present the relating dimensions as Program nodes. As addition, we introduce the Member dimension. This allows associations of annotations directly to Members. Thus, the Member dimension is arranged parallel the Type dimension and directly connected to the facts to allow its association with an annotation. The Member dimension is also connected to the Type dimension, out of the hierarchic relation between Members and Types. The necessary calculations, needed to aggregate the Member dimension to the type level, can be done through the TypeMemberFacts indicator. In order to discriminate annotations that appear at type or at a member, two different hierarchies are connected with the facts.

One may recognize that the Type Classification dimension is missing for the annotation hierarchies. The reason for that is programming logic: Annotations are a fixed instance ("annotations") of the classification. Thus, the Type Classification is not needed for them. The Member Classification indicates the kind of the member such as field or method.

Figure 6.4. Annotations Cube



The cube shows how elements of a program get connected with annotations. In order to compute the different associations, multiple facts are used. AnnotatedMembers count which Members (e.g. Fields, Methods) get annotated. Likewise, AnnotatedTypes do this for classes and similar. The Annotations are split in Type and Member annotations to allow different computations for both.

Computation

We present the computations of the cubes relations in Table 6.7. The TypeMemberFacts are used to calculate the hierarchy of the Program nodes. Therefore, the relation of annotations at the Member level references this hierarchy. Since every Type can have multiple Members and every Member can have multiple annotations this relationship is based on a m:n mapping. The dimension used to ensure a proper calculation is the Member dimension, where the memberID is equal in both tables (AnnotatedMemberFacts and TypeMemberfacts). The annotations associated with the Members are referenced through the fact table directly and their hierarchy down to the Package Hierarchy is realized through referencing dimensions.

Similarly to the annotations appearing at Members, the Type dimension is connected to annotations. Again, the facts are realized as a join. In this case, a reference to the TypeMemberFacts is not necessary to be used, since the Type itself is directly known through the facts. Therefore, the hierarchy of the program nodes is realized directly through referenced dimensions. The calculations for members and their classification is M:N, referencing the TypeMemberFacts. The reason to do so is to enable further associations in queries, where types can be retrieved that contain certain Members and annotations.

The TypeMemberFacts represent the structure of program nodes. C8 – C11 is a point of interest that associates annotations with members or types. By referencing via M:N type or member annotations it is possible to discriminate this facts with appearing annotations. We also restricted the presentation

of the relationships for simplification: The annotation dimensions would have to appear twice in the calculations to credit their appearance at types and at members.

We show exemplary calculations for this Types and Members in Table 6.8. This discrimination of annotations at Types and at Members enable queries like: Types that have a certain annotation at a member and another defined annotation at the type itself. However, we restricted Table 6.8 to a few dimensions that can be depicted as sample and it can be extended in the same way, like Table 6.7 is structured for each of the annotation relations.

Table 6.7. Computations for Annotations

Dimensions / Measures	AnnotatedMemberFacts (Join:AnnotatedMemberFacts &AnnotatedElementFacts) (A)	AnnotatedTypesFacts (Join:AnnotatedTypeFacts &AnnotatedElementFacts) (B)	TypeMemberFacts (C)
(1) Package Hierarchy	M:N – TypeMemberFacts (C1)	Reference Package (packageHierarchyID – 2B)	Reference Package (packageHierarchyID – 2C)
(2) Package	M:N – TypeMemberFacts (C2)	Reference File (packageID 3B)	Reference File (packageID 3C)
(3) File	M:N – TypeMemberFacts (C3)	Reference Type (FileID – 4B)	Reference Type (FileID – 5C)
(4) Type Classification	Reference Type (FileID – 5C)	Reference Type (typeClassificationID– 4B)	Reference Type (typeClassificationID– 5C)
(5) Type	M:N – TypeMemberFacts (C5)	Type (typeID)	Type (typeID)
(6) Member	Member (memberID)	M:N – TypeMemberFacts (C6)	Member (memberID)
(7) Member Classification	Reference Member (memberClassificationID – A6)	M:N – TypeMemberFacts (C7)	Reference Member (memberClassificationID C6)
(8)Member/Type Annotation - Type	Annotation Type (annotationTypeID)	Annotation Type (annotationTypeID)	M:N - AnnotatedMemberFacts or AnnotatedTypesFacts
(9) Member/Type Annotation - File	Reference Type (FileID – A8)	Reference Type (FileID – B8)	
(10) Member/Type Annotation - Package	Reference File (packageID A9)	Reference File (packageID B9)	
(11) Member/Type Annotation - Package Hierachy	Reference Package (packageHierarchyID – A19)	Reference Package (packageHierarchyID – B10)	

Table 6.8. Additional AnnotatedMemberFacts or AnnotatedTypesFacts

Dimensions / Measures	Annotated MemberFacts (A)	Annotated TypesFacts (B)	TypeMemberFacts (C)
(1) Type	M:N – TypeMemberFacts (C1)	Type (typeID)	Type (typeID)
(2) Type - Annotation - Type	M:N - TypeMemberFacts (C2)	Annotation Type (annotationTypeID)	M:N – AnnotatedTypesFacts (B2)
(3) Member	Member (memberID)	M:N – TypeMemberFacts (C3)	Member (memberID)
(4) Member Annotation - Type	Annotation Type (annotationTypeID)	M:N – TypeMemberFacts (C4)	M:N – AnnotatedMemberFacts (A4)

Sample Measures

We show exemplary tables with cube content based on Example 6.1 in Table 6.9 and Table 6.10. Table 6.9 shows results of calculations for annotated members. The amount of the appearance of

annotations at the members is counted. The hierarchies through referenced dimensions are used to build totals at various levels. At the bottom the totals show the appearance count of members for an annotation. Clearly, like in the inheritance sample, aggregations at all shown levels can be done.

Table 6.10 shows another perspective on the cube, by using the TypeMemberFacts. The Member Classification is discriminated vertically and the Members are listed on the left, horizontally. They are directly associated with the annotations appearing at those. In fact, the hierarchy shown in this table is presented from the annotations down to Members to present the interchangeability of dimensions through the completeness of their relations in the cube. Thus, the two methods having SuppressWarnings and Transactional annotations appear at each annotation. But even that the annotations appear multiple times, we show that the computation of them is correct, when building totals. We see it at getVets and findOwners that are affected by different annotations. Thus, the total count is two and it does not matter how often they occur in rows, because the amount of members is counted and duplicates are still just computed as one. Likewise, this is the same instance for the fields together with methods. This way, it is possible to compute the amount of members that are affected by annotations.

Table 6.9. Member Annotations query Result

			Member Annotation			
			Transactional	Suppress Warnings	Persistence Context	Total
Type	**Member Classification**	**Member**	*AnnotatedMemberFacts (A)*			
Entity Manager Clinic	Method	getVets	1	1	-	2
		findOwners	1	1	-	2
		Total	2	2	-	4
	Field	em	-	-	1	1
		Total	-	-	1	1

Table 6.10. Amount of Members discriminated by Annotations

		Member Classification		
		Method	Field	Total
Member Annotation	**Member**	*TypeMemberFacts (C)*		
Transactional	getVets	1	-	1
	findOwners	1	-	1
Suppress Warnings	getVets	1	-	1
	findOwners	1	-	1
Persistence Context	em	-	1	1
Total		2	1	3

6.4.3. Fields Cube

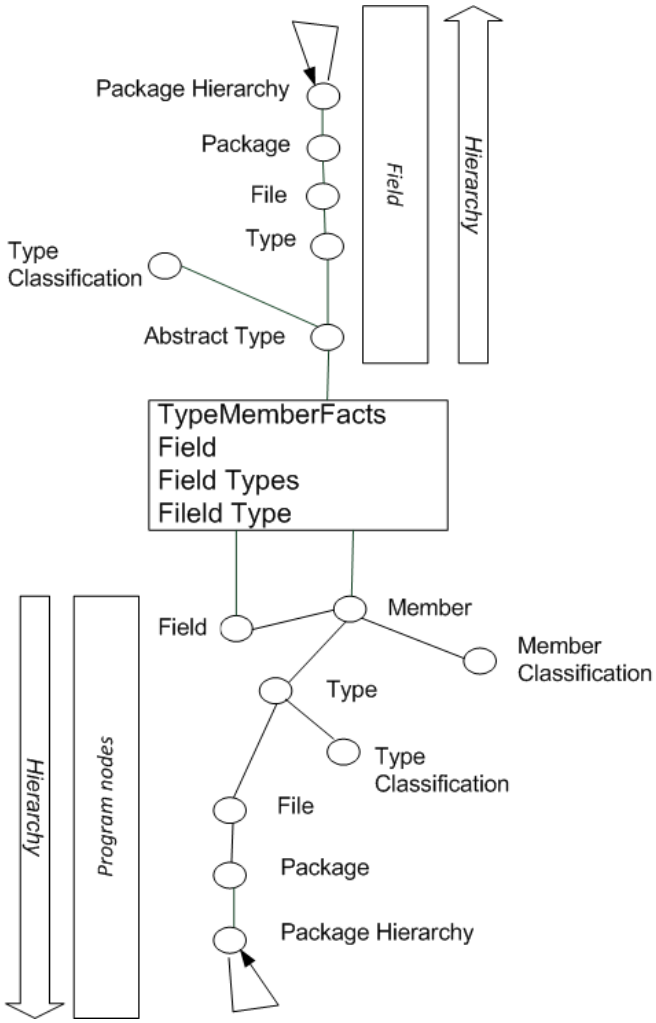
Fields are a subtype a kind of a Member. In order to realize the Field dimension in a cube, multiple facts are needed to calculate the necessary relationships. Like before, we present first the cube, then the computations and stick, finally, to the indicators that represent the cubes relations.

Cube

We present a visualization of the relationships of Fields in Figure 6.5. Like before, the Program nodes are shown, where the hierarchy of Members down to the Package Hierarchy is described. The Member is directly connected with the fact table. The Field dimension is referencing the Member dimension to enable the usage of Fields as Members. The Member Classification allows discriminating Fields from other members, like methods. Due the different measures of the cube,

the Field dimension is directly connected to the facts itself. This enables advanced calculations and M:N relations of Fields. A Field is naturally defined as be of a certain Type. Such a Type can be a primitive (e.g. boolean) or a complex type (class, interface, enum). Likewise, generic Types are possible. We credit all those variants and introduce the Abstract Type that can be a primitive type or a normal and occur multiple times as a field types trough the Field Types facts to credit generics. The kind of the type gets discriminated by the Type Classification dimension that enables the user to slice for different kinds of Types. Clearly, the normal hierarchy for the Type dimension down to package is visualized as a connection of dimensions.

Figure 6.5. Field Cube



The cube shows how fields are connected to their specific types. Such Types can be classes, like Object*, or also primitives like* boolean *or* integer*. Different facts are used to realize all necessary computations to set all elements in relation.*

Computation

We present the computations for Fields and their relations in Table 6.11. The TypeMemberFacts are used like in the annotation case to realize the association of Types and their Members. Thus, Field facts relate to this hierarchy from B1-B5. The association of TypeMemberFacts to the appearance of Fields is done vice versa the M:N mappings to the Field facts in A8-A14. This is necessary, because the TypeMemberFacts computations need to consider the types of the members. This way, it is possible to retrieve Members, where the corresponding field is defined of a certain type. The Field facts, themselves, reference directly the Field. This enables resolving corresponding member and its classification of the Field. The Types of a Field are realized though an M:N mapping with the

FieldTypes table. The FieldTypes are the Facts referencing all the Types associated with a specific Field. This fact is needed to allow the usage of generic parameterized types. This way, multiple Types can be associated with a field.

The hierarchy of AbstractType and Type can be ragged; a class has an abstract type but a primitive type has no normal type. For this issues, the amount of Types is computed through the ComplexType facts (D). Without this approach the aggregation mechanism of the MS SQL server would drop the primitive types. Through this realization the FieldTypes realize this ragged dimensional hierarchy by using the M:N relationship to the ComplexType facts. Anyway, ComplexType and FieldTypes have M:N relations to the other corresponding facts (FieldTypes C1-C6; ComplexType C1-C8). This is done to allow not only one direction of relationship in queries, but also to relate everything to everything.

Table 6.11. Cube Computations for Fields

Dimensions / Measures	TypeMemberFacts (A)	Field (B)	FieldTypes (C)	ComplexType (D)
(1) Package Hierarchy	Ref.: Package (packageHierarchyID – 2B)	M:N – TypeMemberFacts (A1)	M:N – Field (B1)	M:N – Field Types (C1)
(2) Package	Ref.: File (packageID 3B)	M:N – TypeMemberFacts (A2)	M:N – Field (B2)	M:N – Field Types (C2)
(3) File	Ref.: Type (FileID – 5A)	M:N – TypeMemberFacts (A3)	M:N – Field (B3)	M:N – Field Types (C3)
(4) Type Classification	Ref.: Type (typeClassificationID – 5A)	M:N – TypeMemberFacts (A4)	M:N – Field (B4)	M:N – Field Types (C4)
(5) Type	Type (typeID)	M:N – TypeMemberFacts (A5)	M:N – Field (B5)	M:N – Field Types (C5)
(6) Member	Member (memberID)	Ref.: Field (B8)	Ref.: Field (C8)	M:N – Field Types (C6)
(7) Member Classification	Ref.: Member (member ClassificationID -6A)	Ref.: Member (B6)	Ref.: Member (C6)	M:N – Field Types (C7)
(8) Field	M:N – Field (B08)	Field (id)	Field (id)	M:N – Field Types (C8)
(9) Field - Abstract Type	M:N – Field (B09)	M:N – Field Types (C9)	AbstractType (abstractTypeID)	AbstractType (abstractTypeID)
(10) Field – Type Classification	M:N – Field (B10)	M:N – Field Types (C10)	Ref.: Field - Abstract Type (typeclassificationID - C9)	Ref.: Field - Abstract Type (typeclassificationID - D9)
(11) Field – Type	M:N – Field (B11)	M:N – Field (C11)	M:N – Field (D11)	Field Type (id)
(12) Field – File	M:N – Field (B11)	M:N – Field Types (C11)	M:N – Field Types (D12)	Ref.: Field Complex Type (fileID – D11)
(13) Field - Package	M:N – Field (B12)	M:N – Field Types (C12)	M:N – Field Types (D13)	Ref.: Field - File (packageID – D12)
(14) Field - Package Hierachy	M:N – Field (B13)	M:N – Field Types(C13)	M:N – Field Types(D14)	Ref.: Field - Package (packageHierarchyID – D13)

Sample Measures

We show exemplary tables with cube content based on Example 6.1 in Table 6.12 and Table 6.13. Table 6.12 shows calculations for fields and their types. The count of one shows that exactly one Field with one Type exists in the source code. The EntityManagerClinic class contains a Field of the Type EntityManager. Both measures, Field and Field Types, show the same count of one because the field in Example 6.1 one has only one type.

In order to show that different counts of Types and Fields can occur in a cube we show another and additional excerpt of the petlinic application in Example 6.2. The listing shows a Field vets that is a typed generic List of the Vet class. In Table 6.13 the listing is visualized. We show that the count one appears for both types, because the field has both types at the same time. The totals show now the difference between the measures. The Field measure is still '1' because there is only one field. The Field Types is '2', because the Field has '2' different Types. Therefore, the Field types measure can be used as indicator how many Types are used for fields.

Table 6.12. Field Type relation Computation Results

			Field type	
			EntityManager	
Type	**Member Classification**	**Field**	*Field (B)*	*Field Types (C)*
Entity Manager Clinic	Field	em	1	1

Example 6.2. Additional excerpt of the petclinic application

```
public class SimpleJdbcClinic implements Clinic, SimpleJdbcClinicMBean {
    -...
    private List<Vet> vets = new ArrayList<Vet>(-)-;
    -...}
...}
```

Table 6.13. Supplemental Field Type relation Computation Results

			Field Type					
			List		Vet		Total	
Type	**Member Classification**	**Field**	*Field (B)*	*Field Types (C)*	*Field (B)*	*Field Types (C)*	*Field (B)*	*Field Types (C)*
Simple Jdbc Clinic	Field	vets	1	1	1	1	1	2

6.4.4. Methods

Methods are “subtypes” of the Member dimension, similar to fields. This way, multiple fact tables are needed to realize all relationships coming along with methods. Here, we show methods, their parameters and the method calls. Like before, we present first the cube, then the computations and stick, finally, to the indicators that represent the cubes relations.

Cube

We present a visualization of the Method cube in Figure 6.6. At the bottom, like in the cubes before, we show the hierarchy of Members down to their Package Hierarchy. Similar to the Field cube, the Method dimension is connected directly to the fact tables. On the right, the parameters associated with a Method are shown. On the left the, relation of methods and their called methods is visualized. The TypeMemberFacts are needed twice: As relation to the program node hierarchy and for the Method Calls.

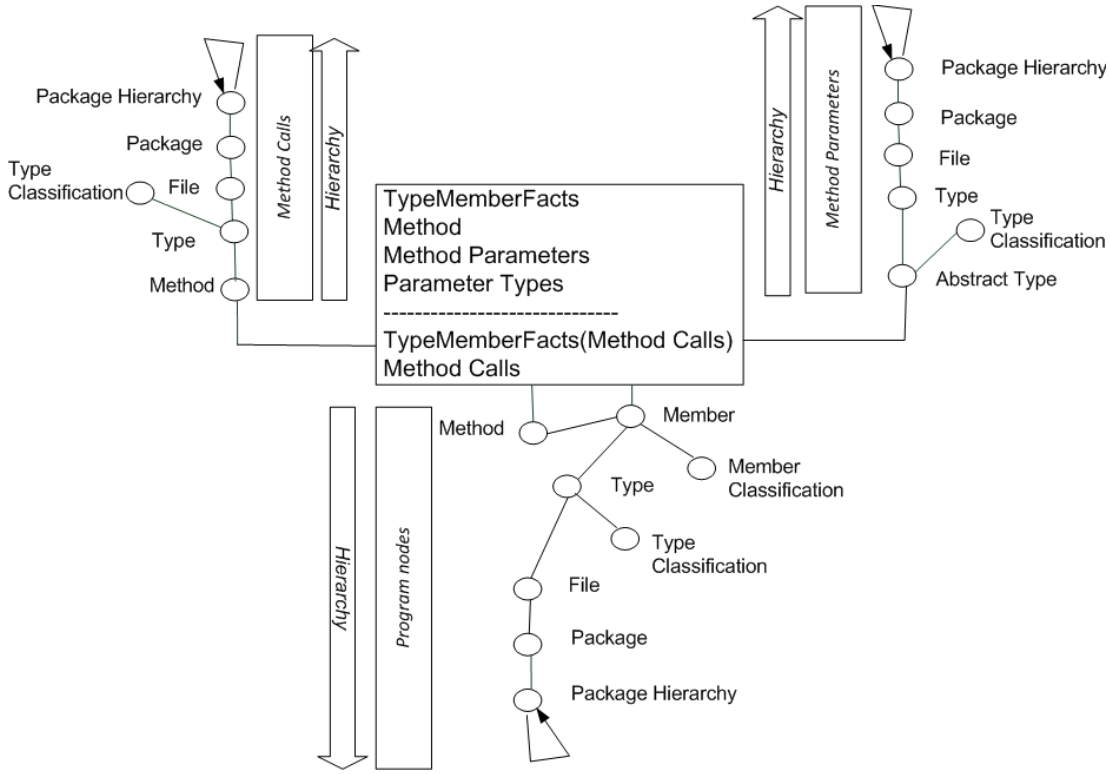
We separate two different measure groups to divide them for the Method Calls and for the Method Parameters. Hence, the first measure group (upper the dotted line, e.g. TypeMemberFacts, Method, Method Parameters, Parameter Types) is used to associate the parameters with a method and to allow the aggregation of Method parameters to the Type level. The second one (beneath the dotted line, e.g. TypeMemberFacts, Method Calls) is used for computations for the executed methods.

In detail, we use the Method Parameters Hierarchy to connect different parameter Types with the Method dimension. A Method can have multiple parameters and each parameter can be defined through various types through generics. All over, it is absolutely similar to the Field description before, just with the main difference that multiple parameters are associated with one method. Therefore, the Method Parameters dimension allows discriminating the different parameters of a

method. This way, the attributes describing the relation of the parameter Type or Abstract Type can be defined. An instance of such parameter attributes can be: The position (first, second third and so on), if the parameter is a return type or if the parameter is an array. Anyway, it is the choice of the implementation to degenerate such "factual" attributes into an own dimension instead using it directly as fact. With an own dimension other attributes can just be added in the future to this dimension, without having the necessity to change the facts, but it makes queries more complicated.

The Method Calls Hierarchy is straightforward expressing the calls of other methods that occur within a Method. Therefore, it expresses what is commonly called the call hierarchy of a program. Since a called Method belongs to its own Type here, also this hierarchy is shown in an expanded way. Again, the Type Classification dimension occurs. This allows respecting the Classification within queries, what can be used to determine Methods that call mostly interfaces or classes.

Figure 6.6. Method Cube



The cube shows Methods associations with their Parameters and the Method Calls defined within a Method.

Computation

We present two sections about computation in the following, because we distinct between the Method Calls and the Method Parameters, to ease comprehension. Furthermore, this clear separation makes it possible to realize two separated cubes, if a speedup is needed.

However, before we go on, we need to note that the Method dimension breaks with the prior practice to use only the similar named table for the dimension. The Method dimension uses also the Member dimension and its corresponding table to get the name and to have a direct fact association. We mainly do this to spare making the cube more complex than it needs to be.

Like the fields cube before, a Method is a Member of a Type. Thus, there is also the same hierarchy existing like for Fields. Methods are declared within a Type that is, itself, located in a File and the File, is located within a Package. In order to simplify the presentation and not to recap a hierarchy that has already been described, we do not show this hierarchy as separate cells in the following. Therefore, dimension 1 and 6 in Table 6.14 and 1 and 9 in Table 6.15 in the following subsections symbolize that kind of hierarchy. How to implement this hierarchy via referenced dimensions is

given in the Fields cube. However, in the following, we first explain the computations for the method calls and afterwards the Method Parameters.

Method Calls

Table 6.14 contains the computation rules for Method Calls that are executed within a method. The MethodCallFacts are the association of a method caller and callee. In order to realize the hierarchy and association with a type of the calling method as well as the called method, the TypeMemberFacts are needed two times. One time, these facts are used to realize the association of the called Method with a Type and the other time to realize the Type association with the calling Method. We show that both of this facts are referenced via M:N mappings in the TypemethodFacts (A). This way, computations for a specific called package or a calling package can be done.

Table 6.14. Cube Computations for Method Calls

Dimensions / Measures	MethodCallFacts (A)	Type Member Facts (B)	Type Member Facts (Method Calls) (C)
(1) File / Package / Hierarchy /Type classification (Similar to the Field table)	M:N – TypeMemberFacts (B1)	Ref.: File (Type – 2A)	-
(2) Type	M:N – TypeMemberFacts (B2)	Type (typeID)	-
(3) Method (Join Member & Method table)	Method (caller)	Method (memberID)	-
(4) Called Method	M:N – Method (B4)	-	Method (memberID)
(5) Called Type	M:N – TypeMemberFacts (C5)	-	Type (typeID)
(6) Called File / Package / Hierarchy /Type classification (Similar to the Field table)	M:N – TypeMemberFacts (C6)	-	Ref.: File (Type – 5C)

Method Parameters

Table 6.15 presents how method parameters associations are computed. The Method hierarchy is degenerated into one column to ease comprehension (row 1 and 9). The association of a method and a type is expressed via the TypeMemberFacts. The association of a Method with its parameters is done via the Method Parameters (C). In order to connect the parameters of a method with the corresponding dimension, the ComplexType table is used as fact table. The TypeMemberFacts are associated with this dimension from line 1- 3. The Method table is directly referenced via M:N. Again, we also use the Member dimension to enable merging the Fields and Method cube together.

The Method Facts (B) are associated via M:N with the TypeMemberFacts for B1 and B2. We did this to enable queries for Methods, Types and their Packages. In order to support multiple parameters for a method, the Method Parameters facts (C) get referenced via M:N mappings in the corresponding cells in B.

In order to support class types and also primitive ones as method parameters, the Abstract Member Dimension is joining the normal types table and the ComplexType table. This is needed because normal types like integer and Boolean are just existing in the AbstractType table. This is important to allow queries for complex and primitive types at the same time. Furthermore, the hierarchy of a complex type, like its package, needs to be enabled for queries.

We introduce the Complex Type measure (D) that the computations can be done for both: Complex class Types and primitive types. This way, the primitive types (int, boolean) without hierarchy are a direct fact in the Method Parameters facts and the class Types are referenced via M:N from the Complex Types facts. The Complex Type facts themselves, reference in other matters the Method Parameters facts. This is done to have a valid association of method parameters and types the other way around.

Table 6.15. Cube Computations for Methods

Dimensions / Measures	TypeMemberFacts (A)	Method (B)	Method Parameters (C)	Complex Type (D)
(1) File / Package / Hierarchy /Type classification (Similar to the Field table)	Ref.: File (Type – 2A)	M:N – TypeMemberFacts (A1)	M:N – Method (B1)	M:N –Method Parameters (C1)
(2) Type	Type (typeID)	M:N – TypeMemberFacts (A2)	M:N – Method (B2)	M:N –Method Parameters (C2)
(3) Member	Member (memberID)	Ref.: Method (memberID - B4)	Ref.: Method (memberID - C4)	M:N –Method Parameters (C3)
(4) Method	M:N – Method (B4)	Method (id)	Method (id)	M:N –Method Parameters (C4)
(5) Method Parameters	M:N – Method (B5)	M:N - Method Parameters (C5)	Method Parameters (id)	M:N –Method Parameters (C5)
(6) Method - Abstract Type (Join with IType)	M:N – Method (B6)	M:N - Method Parameters (C6)	AbstractType (abstractTypeID)	AbstractType (abstractTypeID)
(7) Method - Type Classification	M:N – Method (B7)	M:N - Method Parameters (C7)	Ref.: Method Parameters - Abstract Type (typeclassificationID – C6)	Ref.: Method - Abstract Type (typeclassificationID – D6)
(8) Method – Complex Type	M:N – Method (B8)	M:N - Method Parameters (C8)	Ref.: Method Parameters - Abstract Type (typeclassificationID – C6)	Ref.: Method - Abstract Type (typeID– D6)
(9) Method – File / Package /Hierarchy / Type classification (Similar to the Field table)	M:N – Method (B9)	M:N - Method Parameters (C9)	M:N – Complex Type (D9)	Ref.: Method-Complex Type (fileID – D8)

Sample Measures

We show exemplary tables with cube content based on Example 6.1. Furthermore, we extend the prior listing with Example 6.3 to have more content for the computation outcomes. We use this addition to present a method that also contains calls to other methods. Therefore, we show the additional findOwners method of the EntityManagerClinic class of Example 6.1. Hence, our following descriptions and tables are based on this code. In the following, we first dig into the method calls and then go on to the method parameters.

Example 6.3. Additional excerpt for the Method Cube

```
public class EntityManagerClinic implements Clinic {
...
    public Collection<Owner> findOwners(String lastName) {
     Query query = this.em.createQuery
          ("SELECT owner FROM Owner"+
             "owner WHERE owner.lastName LIKE -:lastName")-;
     query.setParameter("lastName", lastName + "%-")-;
     return query.getResultList(-)-;
 }
```

Method Calls

Table 6.16 shows the findOwners method calls of the EnitityManagerClinic class. Since every Method is belonging to a type, the different Types of the called methods are arranged above the fact

Method Call indicator. Every method that is called gets the indicator of '1'. These indicators can be aggregated together at the type level, like it is shown with the (sub) totals. Likewise, the total amount of called methods can be computed.

We note that our approach enables the computation for the whole hierarchy of the methods and types. We do not focus on this, because we showed it already in the prior cubes and ease comprehension through the focus on easy case.

Table 6.16. EntityManagerClinic Method Calls

	Type				
	Query			EntityManager	Total
	Methods				
	getResultList	setParameter	Total	createQuery	
Method	*Method Calls*				
findOwners()	1	1	2	1	3

Method Parameters

Table 6.17 shows the parameters of a method. We show method getVets () of Example 6.1 and findOwners() of Example 6.3. Horizontally, the different Types of the Method Parameters are arranged. The Method Parameter fact is '1' for the various Method Parameter and Method combinations. The generic Types are treated as normal Types of a parameter. The totals show that the total amount of Types is computed on a per Type base. This way, a parameter can have multiple types at the same time.

Additionally, we show the same table for the Method Facts in Table 6.18. The existence of a method with a certain parameter is indicated by the count of '1'. But, the main difference to the Method Parameter fact is that the computation is done per Method and not per Type. We underline this with the totals that the existence of a method is indicated with a parameter by '1'.

Table 6.17. Method Parameters Computations of EntityManagerClinic

	Type				
	Collection	Owner	Vet	String	Total
Method	*Method Parameters*				
findOwners()	1	1	-	1	3
getVets()	1	-	1	-	2

Table 6.18. Method Parameters as slicer for the Method Count of EntityManagerClinic

	Type				
	Collection	Owner	Vet	String	Total
Method	*Method Parameters*				
findOwners()	1	1	-	1	1
getVets()	1	-	1	-	1

6.5. Holistic Cube

Right before, we described different cubes for different structures within source code. Now, we merge all the former described cubes together in a multi-provider cube that relates the different facts of the single cubes together. All the computations that are needed to realize the relations of all the facts to each other are shown in the appendix.[6] We do not describe those computations here, because

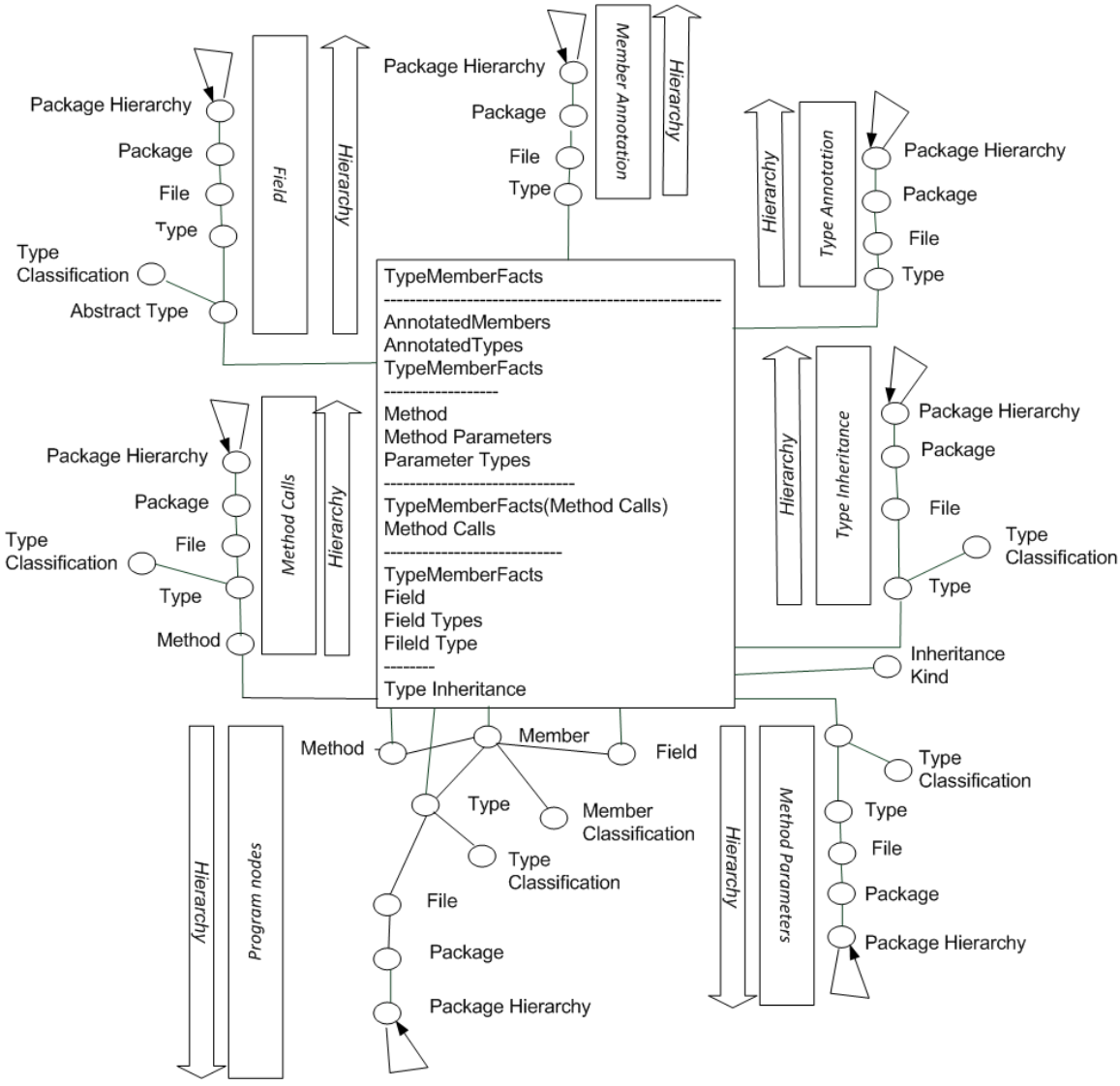
[6]See Section B.2.

they follow the concept of the relations within a single cube that are described earlier. Furthermore, the amount of relations is immense and would not help the clearness at this point. First, we present the holistic cube as a model with fact and then stick to a networked model that is more intuitively understandable.

6.5.1. Holistic Fact Java Model

We show a scheme of a holistic cube in Figure 6.7, containing all the prior described cubes, what we discuss in the following.

Figure 6.7. Holistic Code Cube



A holistic multi-dimensional data cube model for the Java language, created by composing all the small cubes of this chapter together.

Figure 6.7 shows cube dimensions that we described in the earlier sections are arranged around the facts, located in the center. Through the central facts, we relate all the different cubes to each other. In the left corner, the Field cube dimension gets arranged. The Annotations that occur at the member level like field and methods are shown in the middle. Likewise, we show the Annotations at the Type level and their hierarchy, at the right. Right beneath them, the Inheritance is shown. Thereby, the dimension inheritance kind allows discriminating between extending classes and implementing interfaces in queries. Just at the right bottom, the parameter types and their hierarchy of methods are

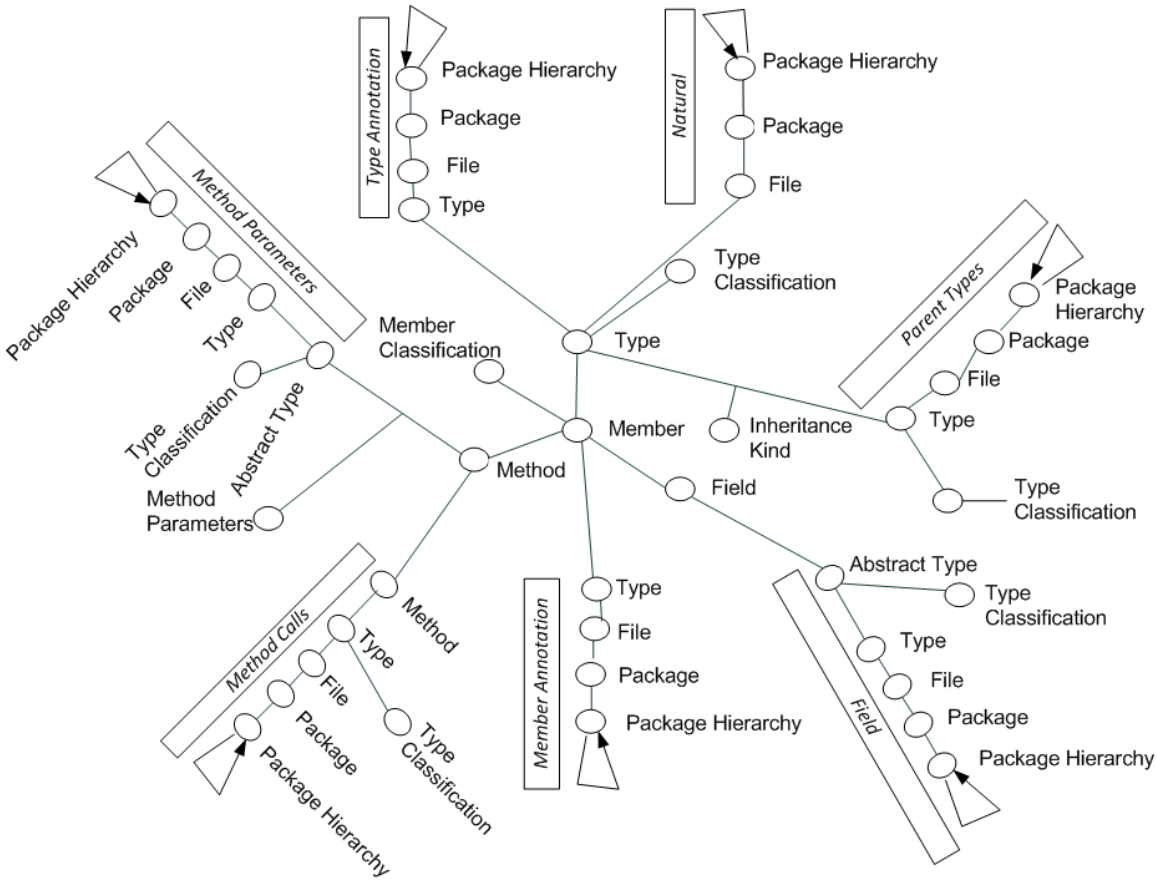
arranged. Right beside this, we arranged the central primary hierarchy of a program. Lastly, above the primary hierarchy the called methods and their corresponding hierarchy is shown.

We see that the primary hierarchy of a program (Program nodes) are the elements that allow to connect all the different dimensions of the various cubes together. The primary hierarchy is the package, class and member hierarchy that is commonly explored via a tree view within an integrated development environment. The distinct dimensions, like annotations or super types relate through the facts to this primary hierarchy. This enables queries that are containing relations of the various cubes at the same time. For instance, this enables searching types that have a method with a certain string parameter "get", a @Deprecated annotation and that extend a Clinic superclass. Thus, this query contains the different dimensions out of the diverse cubes presented before at the same time. This enables the discrimination of a result set for all the different facts at the same time.

6.5.2. Networked Code Cube Model

However, the visualization of the holistic cube also is hard to comprehend. Hence, we present an alternative representation in Figure 6.8. This presentation adapts the cube representation to a networked model and credits the connections of elements within source code better. We present the primary hierarchy of elements in the center of the model and show the associations with the distinct hierarchies. The elements of the primary hierarchy are directly connected with the source code element dimensions that are associated with them. For instance, the Type as also the Member is associated with the annotations that can occur on those.

Figure 6.8. Networked Cube Model of Code



This networked code model omits the different fact relations, occurring in source code, in favor for a better comprehension. As replacement for the facts, the model focuses on the relations within source code and moves the primary hierarchy of Packages, Types and their members into the center of the model. This center is used to make connections to all the other hierarchies that occur in source code.

Generally, all the paths in Figure 6.8 describe kinds of hierarchies that occur in source code. For instance, a package contains types and those types contain members. Members can be divided into Fields and Methods. The method call hierarchy are the called methods that are, themselves, belonging to a certain Type that is, again, described in a package. Similarly, other hierarchy relations to each other can be derived from the model. In order to symbolize the different hierarchies that relate to the central hierarchy of a type and its member better, all the hierarchies got a block over them that describe the kind of the hierarchy.

Moreover, the model shows that attributed relations, like Inheritance Kind, can occur in source code and be expressed in such a model. We see that this relations are attributed, because its the relation edge is associated with another edge. Such attribute dimensions allow to discriminate the distinct kinds of associated hierarchies and to slice all dimensions therein. For instance, the inheritance kind dimension specifies if there is an interface implemented or a superclass is extended.

All together, also this kind of representation of a cube for source code is not perfect at all. But, we believe it represents a meaningful addition to the pure fact model before. It helps to overview the associations in source code more easy. This makes it is easier to design a query, because the associations can be seen directly in a graph. Thus, we see this model as additional documentation for the holistic model.

6.6. Advanced Data Associations

In the last section, we presented a holistic multi-dimensional model for Java. In this section, we present how we use this model and associate further data with it. In special, we present how detected Style Issues of an external tool and information about source code authors can be integrated into the model.

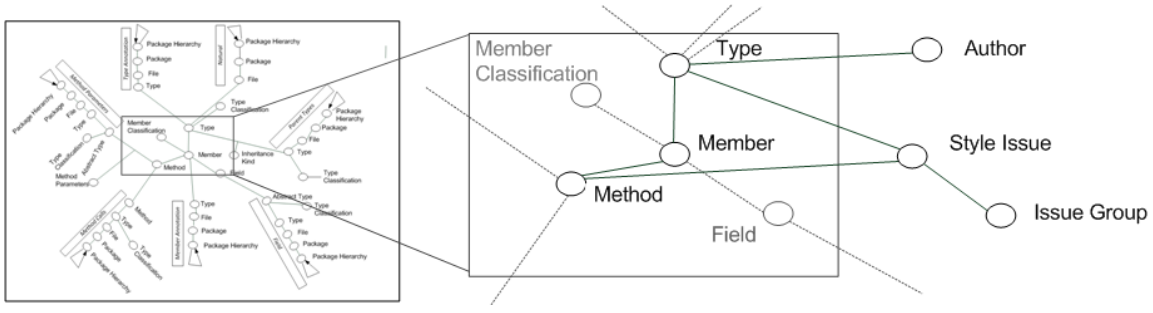
First, we describe the general approach associating data. Then, we present the relational model extension and present a cube on top of it. Finally, we give an example how the associations manifest in queries.

6.6.1. General approach

The general approach to associate further data with programming nodes is to use parts of the prior described multi-dimensional Java model and 'dock' relations to these nodes. We present the approach in Figure 6.9. There, we zoom into a part of the Java model in a box and grey out the nodes that do not have a direct relation with the associated data. Outside the box, dimensions that refer to nodes of the program are visualized. We see that Types can have Authors and Style Issues can be associated with Types and Methods. Additionally, Style Issues can be grouped together.

However, we did not come up with this example out of nowhere. Rather we lean this example on the data produced by findbugs[7] and created the associations corresponding to its data.

Figure 6.9. Associated Data with Code Structure



We show a miniature of the networked Java cube and mark a small area. The zoom into this area shows how different nodes are associated with the Style Issue Dimension and the corresponding Authors. We see that those dimensions also can have hierarchies, like implied with the Issue Group.

[7] See [212]

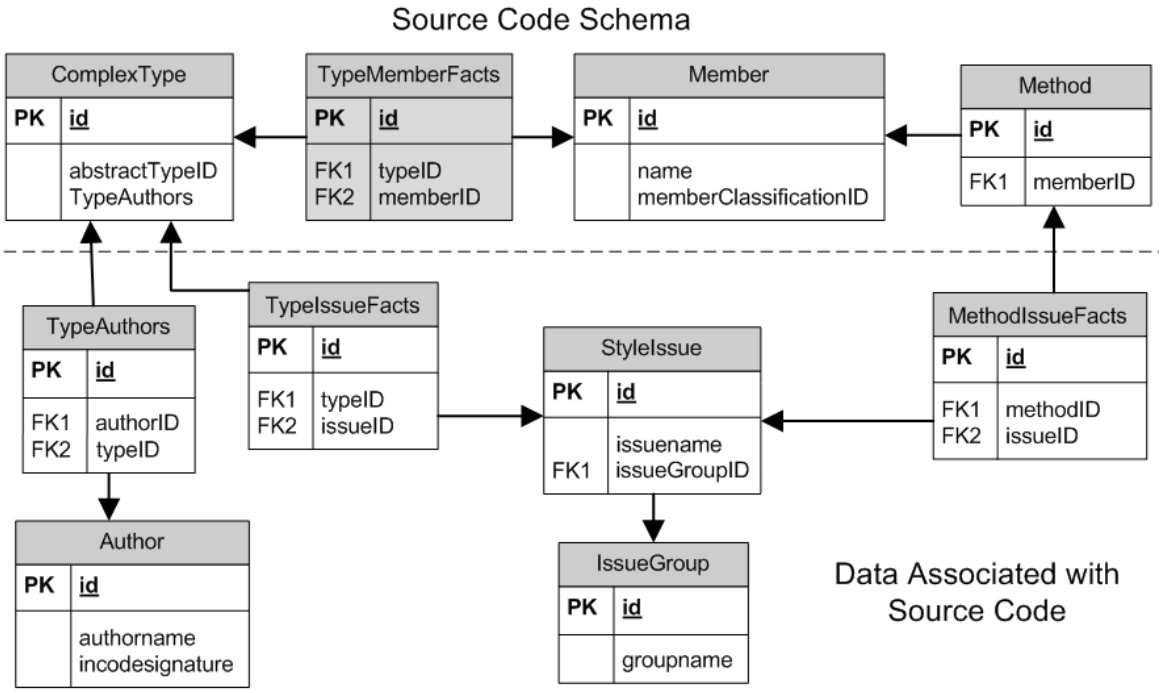
Similarly to the Style Issue dimension, we created the Author dimension. In order to gain information which Authors wrote which Types, data from version control systems can be extracted and references to the code can be uncovered. For exemplary use cases, we used the Author tags, occurring in source code and used them as information which authors wrote which Types.

All together, the data that we associate in this chapter serves as example that nodes of the Java model can be associated with further information. In the following, we dig into the details how the relational model is extended and get into the corresponding cube and its computations. With this examples it is now clear, how additional data can be integrated with Hypermodelling.

6.6.2. Relational model

We present how additional data is associated with our relational model in Figure 6.10. There, we show the elements of the relational code structure schema that we used above the dotted line. We see that the Authors table is associated with the CompexType through a fact table, called TypeAuthors. Likewise, the same is done for the Style Issues. In case of Style Issues, two different fact tables are used: TypeIssueFacts and MethodIssueFacts. The hierarchy from Style Issues to Issue Groups is realized though a relation to the IssueGroup. Therefore, we see that the additional data is just 'docked' onto the relational model and there is no need to alter the original model.

Figure 6.10. Relational Model Extension

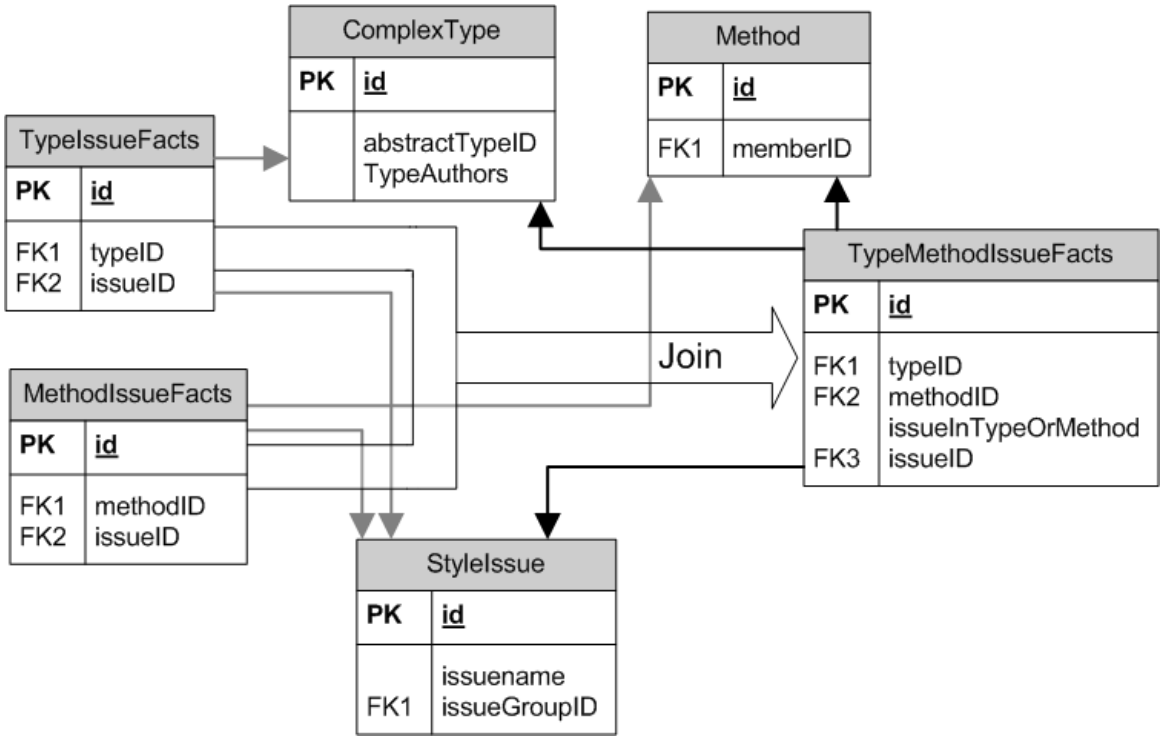


The relational schema shows that additional data is associated with the relational Java source code model. The dimensions of Author and Style Issues are associated with the code structure by the TypeAuthors, the TypeIssueFacts and the MethodIssueFacts. Those 'linking' tables are fact tables that allow to associate the nodes of a program with multiple dimension values.

However, in order to create a concise cube on top of this relational model (Figure 6.10) that respects the circumstance that Style Issues can appear at Types and also at Methods, we ease building a cube on top, by creating a view from TypeIssueFacts and the MethodIssueFacts. We visualize this join in Figure 6.11. There, we see that the TypeMethodIssueFacts get the additional property of issueInTypeOrMethod. We create this property to assure proper aggregation from Issues from a Method to its Type. Without this additional property it would still be possible to discriminate if an issue appeared at a method or a Type, but it would be harder to comprehend. Through the additional

property an easy to use slicing value is available. Therefore, we create our cube in the following on the joined table.

Figure 6.11. Joined Relational Model Extension



The TypeIssueFacts and the MethodIssueFacts are joined into the TypeMethodIssuefacts. This is done to credit the fact that logically one would refer to Style Issue associations to be one fact table. However, the pure relational model does not support object-like structures of inheritance or generalization.[8] *Since we stick in this work to this model, we realize the association in one table via a join, by adding the issueInTypeOrMethod column to the joined table for discrimination.*

6.6.3. Cube

We show the cube on top of the relational structure of the section before in Figure 6.12. At the bottom the normal primary hierarchy, like it was shown in the prior cubes, is visualized. Since Methods and also Types can be associated with the Style Issues, both are linked directly to the facts. In order to credit the hierarchy of Types and Members and to allow computations about the associated data with those, the TypeMemberFacts are one fact of the cube.

At the right side, we show the hierarchy of the Style Issue related dimensions that are connected through the TypeMethodIssueFacts. We see that we use the property where a Style Issue occurs as dimension that describes the Style Issue associations.

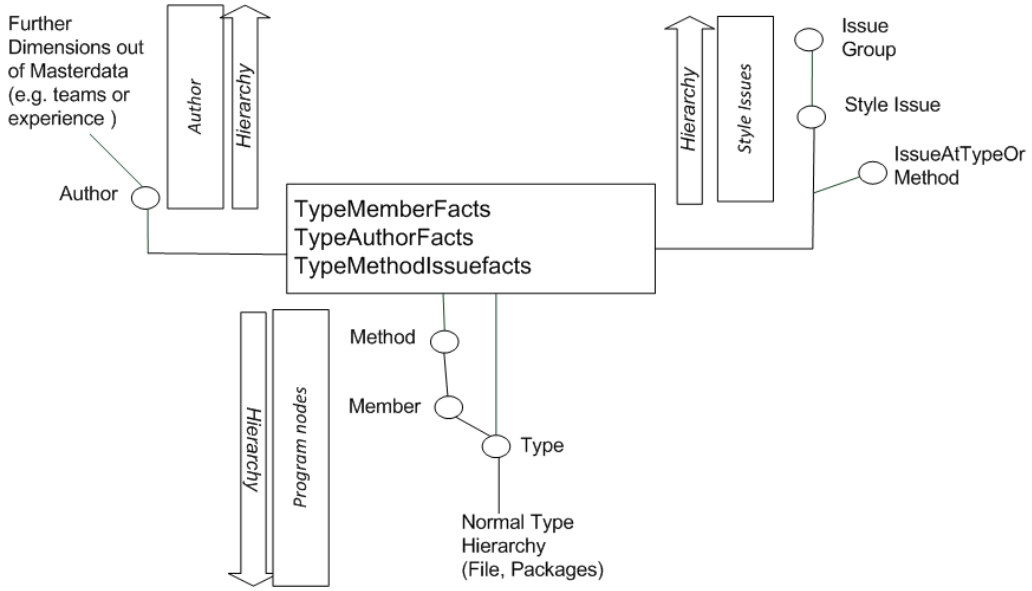
Clearly, it is a choice of modeling, if an own dimension is used for this or if the IssueAtTypeOrMethod is just used as attribute of the Style Issue dimension. We decided to present it as own dimension to emphasize the potential slicing character.

On the left side, we arranged the Author dimension and link it to the code by the TypeAuthorFacts that corresponds to the TypeAuthors table. We also credit that further information about others can come out of other systems. Potential master data for future abstractions can be information which

[8]We merely focus on using capabilities of most databases and not to special features, like object-oriented databases, for compatibility issues.

Authors belong to which teams. By integrating such data, further abstractions can be done in the future. Right here, we focus only on the direct association of external data with code structure.

Figure 6.12. Style Issues and Authors Cube



The cube shows the different facts that are needed to associate external data with code structure. On the left side, we show that Authors are associated with code structure. On the right side, we describe that Style Issues are also associated with code structure.

We note that we used just relations of the primary hierarchy (packages down to types and their members) in code to describe those relations. Therefore, we mention that the association with the Type-Member relations how they occur in code by the TypeMemberFacts are just an example. Clearly, any other relation in code can be used to associate external data with the various hierarchies that occur in source code. Right here, we used this obvious relation, because it is a straightforward to follow example and other relations are realized similarly.

6.6.4. Computations

We show the computations of the cube in Table 6.19. We see that the TypeMemberFacts in column A and the Method table in B serve to represent the code hierarchy. A1-3 and B1-3 are arranged like in the prior described method cube in the corresponding section (Section 6.4.4). The TypeMethodIssueFacts do not refer to this structure, because it has direct relations to the elements through its fact table. However, Author does not and has no relation to method and Members. Since Authors are associated with Types, it has a direct relation to the Type dimension in D1.

In order to allow queries that allow to discriminate the source code structure based on Issues, TypeMemberFacts (A4-A6) and Method (B4-6) relate via an M:N relation directly to the TypeMethodIssueFacts (C4-6C). Likewise the same is done to relate the TypeMemberFacts and the Method in line 7 to the TypeAuthors (D). This way, the code structure is related to Style Issues and to Authors.

This means, Authors and Issues have no direct relation, but since they are related to the code structure, they both relate (via M:N) to the code. This means an indirect relation between both can be computed. This works, because their fact tables relate to the code structure or directly to the code elements. The back references from the code structure, in A4-6 to C4-6 for Style Issue dimensions and in A7 to D7 for Authors, assure that the code structure is capable to discriminate by the associated data dimensions. Therefore, in cases where a direct relation to the other facts is

desired, just a reference to the code structure is needed. Then the Warehouse is capable to compute internally the discrimination and the indirect relation.

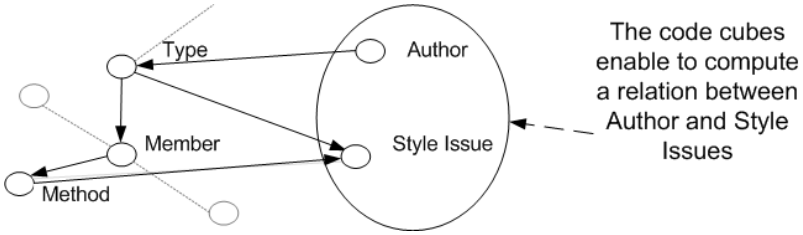
We visualize this possibility to create relations between elements that have originally no relation and get related though code structure in Figure 6.13. There, we can see that Author shares, through code structure, a relation to Style Issues at types and Methods.

All together, we show how relations of external data with code structure can be computed, following the prior Hypermodelling framework. Thereby, we also credit how non related elements can be related though computations that use code structure. In the following, we give small query example.

Table 6.19. Cube Computations for Associated Elements

Dimensions / Measures	TypeMemberFacts (A)	Method (B)	TypeMethod IssueFacts (C)	TypeAuthors (D)
(1) Type	Type (typeID)	M:N – TypeMemberFacts (A1)	Type (typeID)	Type (typeID)
(2) Member	Member (memberID)	Ref.: Method (memberID - B3)	Ref.: Method (memberID - C3)	M:N – TypeMemberFacts (A2)
(3) Method	M:N – Method (B3)	Method (id)	Method (methodID)	M:N – TypeMemberFacts (A3)
(4) Style Issue	M:N – TypeMethodIssueFacts (C4)	M:N – TypeMethodIssueFacts (C4)	Style Issue (issueID)	M:N – TypeMethodIssueFacts (A4)
(5) IssueAtType OrMethod	M:N – TypeMethodIssueFacts (C5)	M:N – TypeMethodIssueFacts (C5)	IssueAtTypeOrMethod (issueAtTypeOrMethod)	M:N – TypeMethodIssueFacts (A5)
(6) Issue Group	M:N – TypeMethodIssueFacts (C6)	M:N – TypeMethodIssueFacts (C6)	Ref.: Style Issue(issueGroupID - C4)	M:N – TypeMethodIssueFacts (A6)
(7) Author	M:N – TypeAuthorFacts (D4)	M:N – TypeMemberFacts (A4)	M:N – TypeAuthorFacts (A7)	Author (authorID)

Figure 6.13. Computations of Indirect Relations



Authors and Style Issues have no direct relation to each other. Both are just associated with the code structure. The general idea is that both relate to the code structure and that such relation can be used to compute their indirect relation.

6.6.5. Sample Data

With the prior described computations, we show now a sample of simple measures that can be revealed with a query. In order to have a valid sample we present a source code sample in Example 6.4. We see that the SimpleJdbcClinic type has two different authors. Furthermore, findbugs reveals that new Integer(id) is inefficient and should be replaced by Integer.valueOf(id). We represent this Style Issue in a query result in Table 6.20. There, we see that the corresponding method, loadOwner, gets the indicator of '1' for the Style Issue Dimension with the value that a static method invocation would be better.

The Author facts, represented in a query result can look like Table 6.21. There, we see that every Author is associated with the Type as indicated by the count of '1'. Clearly, in all cases the total

sums works for these indicators. For instance, it is possible to get now all the methods that were written by a certain Author and so on.

The next table (Table 6.22), shows the prior mentioned association between data that comes out of two different sources. There, we relate Authors with Style Issues. We see that it is now possible to compute Style Issues in correlation to Authors.

Example 6.4. Code Sample for Style Issues and Authors

```
/**...
* @author Max Mustermann
* @author John Doe
*/
...
public class SimpleJdbcClinic implements Clinic,
                                SimpleJdbcClinicMBean {
    -...
    public Owner loadOwner(int id) throws DataAccessException {
        -... new Integer(id)-...
```

Table 6.20. Sample Style Issue Indicators

		Style Issue
		TypeMethodIsseFacts
Type	**Method**	Better cast by static Method available
SimpleJdbcClinic	loadOwner	1

Table 6.21. Exemplary Author Associations

	Type
	TypeMethodIsseFacts
Author	SimpleJdbcClinic
Max Mustermann	1
John Doe	1

Table 6.22. Style Issues Associated with Authors

	Style Issue
	StyleIssueFacts
Author	Better cast by static Method available
John Doe	1
Max Mustermann	1

6.7. Known Shortcomings

In the following, we present the known limitations and shortcomings of our Data Warehouse Java model.

The provided models focus mostly on the pure object-oriented features of Java and annotations. Therefore, we see the model as easily transferable to similar languages. However, plenty of other working languages, like functional programming (e.g. Scala [196]), are existing. At this point, we cannot answer if our models work for those or can be adapted. Therefore, further research needs to be done in this area.

Another limitation of the source code data model is the association of types (classes, interfaces) with other elements. Currently, we treat generic types just as a multi-type association. Therefore, if something like List<Map<String,String>> is in our cube, it is the same like Map<String,List<String>>. In reality the second definition is a completely different data structure. Currently, we ignore the order of the generic types. We did that out of two main reasons: First, we wanted to simplify reality and not start with a full blown model. Secondly, we are confident that in

most cases the order of generic parameterized types is not so important and the focus is more on the used types. However, for future versions we see the need to represent this information in our models and to determine if its implementation would be beneficial.

Similarly, another limitation is about the loss of code structures in our model. For instance, we loose information about the inner blocks within a method. Information about variables, how the blocks are nested within each other, loops and similar things are not represented within our relational model. We neglected that information on purpose, because we see our Hypermodelling technique as abstraction technique and not as one to one representation of code. In order to credit this information in future versions, we see the need for investigations how these things can be abstracted as facts.

Additionally, we loose the sequence how method calls are done within our model. This means, the facts about method calls do not necessarily credit the order how methods are executed within a method. We did this on purpose, since we do not see the motivation to load inner code structure for the sake of abstraction. But still, it we need to look into it in the future.

However, our models also missed some relations out of computation-comprehension- difficulties. For instance, we presented the cube about inheritance, where queries can determine the supertypes. Supertypes can have multiple supertypes themselves. Therefore, these kind of relations also represent a hierarchy. But the hierarchy is not a normal hierarchy. It is a hierarchy where each parent can hold multiple parents. Therefore, the hierarchies are M:N. Literature [224] describes details how this kind of M:N hierarchies can be computed and realized within a Warehouse. Some tests application test of M:N relations that we did, resulted in working and right computations, but also in hard to understand queries. Therefore, we see the need to focus on this issue in future work .

An additional shortcoming is that we do not represent information about logical associations in source code that can be inferred out of language constructs. Such logical associations are overridden methods that have the same signature as in the supertype. It can be beneficial to have such connections directly materialized as relations within the relational and multi-dimensional structure. Currently, such associations need to be revealed through queries. Future investigations can identify such connections and determine if those are beneficial to be represented as facts.

Also, we did not mention calculated members. Calculated members are indicators that are computed out of queries or other computation rules and then serve as normal indicators. Future research should identify which of the prior mentioned limitations, such as logical associations, can be solved with calculated members. Those can advance the current version without having to change to much on the model. Therefore, we see investigations about those as important to improve the current model.

Nevertheless, the current models did not credit many dimension attributes and we focused mainly on the relations within source code. Future investigations should identify which additional attributes in source code should be part of the models.

Currently, the models lack to express different revisions of source code and it is needed to investigate how the models need to be altered to express "time". In first thoughts about representing "time" we came up that there are possibly plenty of so called slowly changing dimensions within source code, where dimensions are moved in relations to other dimensions.

Another shortcoming of our models is that they are dependent on the relational layer, what makes it possible to utilize the whole tool chain of Data Warehouses. But even though, we see investigations how to 'dock' the multi-dimensional cubes directly to source code as important.

6.8. Discussion

We described the application of Hypermodelling for Java and presented the necessary models. We also gave insights about the shortcomings that our implementation revealed. In the following, we discuss our Hypermodelling implementation and compare its complexity to an imagined source code parser (custom solution for a specific use case) that extracts and computes the same facts like a Hypermodelling query.

Potential failures. We realized the multi-dimensional access of code structure with means of Data Warehousing. Clearly, the same multi-dimensional access can be realized by a custom

program. In contrast to our solution, custom codings need a lot of hand-written source code. A lot of source code leads to possible mistakes and a custom application program interface, with lots of documentation. Our solution shifted this complexity into the internals of a Data Warehouse. Therefore, the computations and the models are well documented and conform to the Warehouse logic. The computations are done within the Data Warehouse which is applied in so many use cases and can be assumed to have less failures then a newly created custom program. Main failures in this scenario can only come from wrong specified computation directives. A custom program can have errors in the internal computations itself. Therefore, we claim that our implementation is potentially less error-prone compared to a custom implementation that computes facts out of a program.

Adaption Complexity. However, users need to use Data Warehouse query languages to investigate code with our approach. Furthermore, Data Warehouse knowledge is also needed if our models should be extended or modified. We argue that in a comparable custom solution also knowledge about its internals needed. Additionally, we provided also plenty of examples and how additional data can be 'docked' onto our models. Therefore, the complexity to extend our models is manageable. The main difference to a custom solution is that in the case of Hypermodelling the documentation is mainly in the form of Data Warehouse techniques. Several books exist about this topic. With a custom solution a custom documentation will be needed.

Solution Complexity. Another point to judge the complexity of our model is to compare it to custom cases of cube design in Data Warehousing. Our models are more complex, compared to typical Data Warehouse cubes. We estimate that, because we use quite often many to many relations that were originally introduced in MS SQL server 2005. Therefore, we assume that many to many relations are an advanced use case, because 2005 has not passed so long. Hence, we classify our Warehouse models at the upper end of complex cubes. However, when the computations are done in a comparable custom solution, they still exist in a custom solution. Therefore, we see both solutions as complex.

Implementation Effort -Flexibility. The implementation effort that we needed to create the implementation also serves as an indicator to judge our solution. In order to create the relational model, the cubes and their computations about three months of full time work was needed. When just one fact of source code is analyzed with a custom solution, less work compared to our spent time to create the models would need to be done. However, as we show in the following chapters, we do not target one single use case with our implementation and focus on the flexibility that can be achieved with Hypermodelling. With custom solution every time the fact context is switched, additional effort needs to be spent to extend the solution. Thus, we see the flexibility of Hypermodelling and the implementation effort that we need as the trade offs. For a single use case there is no need for Hypermodelling and a custom coding approach is better. If flexibility and easy adjustments are wanted, Hypermodelling is superior, because queries can easily be altered. Additionally, the needed effort enable Hypermodelling for Java has been already spent by us. All thoughts are within our models and the first creation effort is already spent. For other languages, researchers can decide if they want to take the same effort. Additionally, since object-oriented languages are somewhat similar, we mark out that the time to adapt the Java models for other object-oriented languages will result in less effort, compared to our creation from scratch for Java. Therefore, we see the advantages of Hypermodelling compared to a custom coding solution in its flexibility.

Model Completeness. Another issue is the incompleteness of semantics of the models that are a bijective representation of Java. Our models are defined on an abstract level and reduce the Java language structure to a more abstract level. Therefore, the complexity of our current model is less complex to the Java language, because Java can be transformed into our model, but not back. A custom coded solution can deal with the specifics for the needed certain facts of interest. Therefore, if in detail facts are wanted to be analyzed, a custom solution is needed.

Roundup. All together, we wrap up the complexity of the comparison in Table 6.23. There, we see that Hypermodelling has some advantages compared to a custom solution. However, we are careful not to claim that Hypermodelling is superior in all areas. The main advantage of Hypermodelling is its flexibility that it uses a Data Warehouse's architecture, what outranks a

custom solution. We are software engineers and follow the main principle to use of tested code and architectures is less error prone then to recreate codings from scratch. Therefore, we see this as key point for Hypermodelling.

Additionally, the flexibility of Hypermodelling exceeds the one of a custom solution that is built for just one use case. We show further insights about this flexibility in the next chapters. There, we show different kinds of applications can be created on top of our implementation, what would not be possible with a custom solution. On the contrary, in order to realize all of this scenarios, a lot of custom codings would be needed and the effort to create those would exceed the effort to create Hypermodelling in wide ranges.

Table 6.23. Complexity Comparison of Hypermodelling

Complexity Criteria	Custom solution, created to investigate one point of interest	Hypermodelling
Potential failures	High risk, because plenty of code would be needed	Only configuration based mistakes in the computations. The internal computation units are well proven since they are widely applied
Adaption complexity	Knowledge dependent on specific solution	Dependent on the knowledge of Data Warehousing
Solution complexity	Complex computations in code are needed.	Usage of advanced cubes
Implementation effort	Medium	High effort
Flexibility	Needs manually to be adapted and extended with code	Easy to switch to new facts through queries
Model representation completeness	For the desired facts possible	Uses an abstraction level to raise comprehensibility

6.9. Summary and Conclusions

We described the relational schema for Java that is needed to apply Hypermodelling within common Data Warehouse solutions. Then, we presented different data cubes and computations how Java code relations can be realized as facts and dimensions in a Data Warehouse. Afterwards, we pointed out that all the small data cubes can be put together into a holistic one that represents nearly the entire Java structure at once. We followed with a concrete example how further facts can be 'docked' to the Java model. Then, we gave remainders about the incompleteness of our work and pointed out which questions about modeling stay open for future research. Lastly, we gave a discussion about the static properties of our described solution.

All in all, this section was very technology centered and the main remainder was to give concrete implementation details how Data Warehouse technology can be used to implement Hypermodelling for Java. Right now, other researchers can investigate our advance our implementation and identify advancements and shortcomings.

However, when we derived Hypermodelling in the prior chapter, we described the necessity to evaluate its implementation. Here, we presented an implementation and recognized that Hypermodellings main benefit is that the effort to create data models is just needed once, when it is compared with a custom solution for a specific use case. Furthermore, custom solutions that offer similar investigations is likely to be more error prone. Hypermodelling on the contrary uses an well tested Data Warehouse infrastructure. However, we are careful not to claim Hypermodelling to be superior in all cases and it offers more a trade off, compared to a custom solution. In general, to apply Hypermodelling for a single use case seems to be more effort than to apply a custom solution. Therefore, Hypermodellings advantage lies in its flexibility to be applied for different use cases. Therefore, we use the implementation, presented here, as black box and focus on the usage of this box in the next chapters. There, we give an examples how we apply Hypermodelling for different use cases.

Part III. Evaluation and Applications

7. Hypermodelling Analysis

“What we leave behind is not as important as how we've lived.”
Jean-Luc Picard. Star Trek - Generations. 1994

This chapter shares information with: "*T. Frey, V. Köppen. Exploring Software Variance with Hypermodelling - An exemplary approach. In 5. Arbeitstagung Programmiersprachen (ATPS'12). Im Rahmen der Software Engineering 2012 (SE2012). Gesellschaft für Informatik (GI). Berlin, Germany. 2012.*"[104]

Abstract. Framework manufacturers face the challenge to determine which parts of frameworks are used and varied. Application developers want to know on which framework elements their application is depending on. Currently, programs need to be parsed to extract information about framework usage what consumes time and effort and makes information mining inflexible. Hypermodelling utilizes Data Warehouse technologies for source code investigations to overcome the current limitations. In this chapter, we demonstrate that Hypermodelling is suitable to explore software variance. Furthermore, we illuminate examples how dependencies can be expressed and how statistics can be computed. We present reports based on real application data and reveal multiple facts about a large program. This supports our assumption that Hypermodelling can be used to explore software by queries in an easy way.

7.1. Introduction

Right before, we introduced Hypermodelling and provided insights about a concrete implementation. In this chapter, we study the scenario to do an source code investigation with the approach. This evaluates the feasibility of Hypermodelling to do ad-hoc source code investigations.

Our investigation targets source code reuse and quality to stick to the main topics of this thesis. Source code reuse is one of the key concepts in modern application development. Programmers develop applications by using already encoded functionalities of frameworks. Commonly, object-oriented inheritance of framework classes or interfaces is used to employ predefined functionalities [228]. The types that get inherited are in Java called superclasses and superinterfaces. In the following, we refer to both as supertypes. Developers vary such supertypes by adding or altering functionalities in subclasses, what we call children or inheritors [54]. Hence, all distinct inheritors, i.e. extenders or implementers, create a huge amount of diversification of the original superclasses of the framework.

Framework manufacturers and application developers face the challenge to understand how, where, and which framework elements are used. For instance, there is the desire to know which methods are commonly implemented and overridden in subclasses. The implemented methods point out a variation of the originally offered methods by the frameworks supertype. Developers want to point out how different frameworks are used within an application. The diversity of inherited classes and implemented methods gives developers details on the variation of the frameworks used within the application. The variation is important to draw conclusions on dependencies within a framework.

However, researchers put a lot of effort on source code mining [54, 107, 235]. Source code mining is about extracting facts of source code and associated artifacts to analyze them via machine learning algorithms [54]. A main issue in mining is that there is no standardized infrastructure to extract facts. Thus, before mining starts, facts need to be specified and extracted. This consumes a lot of time and effort. In order to advance source code analysis, we developed the Hypermodelling approach in the prior chapters. Hypermodelling utilizes Data Warehouse technologies to enable and infrastructure an analysis of source code with Online Analytical Processing (OLAP) queries [114]. This enables fast and flexible investigations of source code through queries without having a complex fact-extract scenario. In this chapter, we use Hypermodelling to analyze different perspectives about software. In the following, we give an overview of the main contributions and a reading guide.

7.1.1. Contributions

In this chapter, we show that Hypermodelling can be used to explore diverse facts on variances of frameworks. We depict reports and visualizations that reveal facts on software variances, dependencies and method call statistics. Hence, our contribution is to demonstrate that Hypermodelling can be used to do investigations about software variance and to reveal several facts about a large programs architecture. Our reports give first clues what types of queries can be composed with Hypermodelling. This supports our hypothesis that Data Warehouse mechanisms can be beneficial for software investigations. All together, our contribution is as follows:

- Usage evaluation of Hypermodelling for investigations

  Through the usage of Hypermodelling throughout the source code investigations, we verify that Hypermodelling is suitable for this kind of investigations.
- A detailed investigation about software variance

  We provide details of the investigation results and how variance is structured within a large project. Researchers can now use the revealed facts to do further and in-detail investigations and conclusions about variance.
- Different investigations about dependencies

  We provide exemplary investigations about dependency investigations based on method calls as complementary to the variance investigation. Those can now be used to do an in-detail comparison with dependency investigation tools.
- A statistic about method calls and the amount of method parameters

  As addition to the investigations, we provide a short investigation about method parameters and method calls. This illuminates that further investigations about this appearance are needed.

7.1.2. Chapter reading guide

In the following, we present first a brief background on the investigated project. We follow up with the investigation and present various reports about source code. First, we provide an overview on project inheritance statistics. Then, we discuss how and which packages get varied. Afterwards, we drill down to method and type level. Then, we show examples how Hypermodelling can be used to investigate dependencies. Finally, we show how it Hypermodelling can be used to compute first statistics about the amount of method parameters and method use. At the end, we draw conclusions and give an outlook to future work.

7.2. The Alfresco Project

In order to have valid and real world application data, we downloaded a head revision of the Alfresco project[1] from the corresponding version control system. Alfresco is a popular open source content management system, written in Java. Then, we loaded two subprojects (core and repository) of the whole Alfresco project into our Data Warehouse. Thereby, we also created types, coming out of other projects or the Java standard that were referenced by classes of the Alfresco project. The following measures were computed by queries against the processed cubes that were generated out of the loaded data. The total amount of types in the Alfresco package is 3,651. These 3,651 types consist out of 2,866 classes, 658 interfaces, 124 enumerations, and 3 annotations. The total amount of members is 42,823. These members are divided into three different annotation parameters, 13,694 fields and 29,126 methods. It is interesting that the interfaces declared in the Alfresco project define 4,612 members from which 3,770 are methods and 842 are fields. This means, a huge amount of constants is defined by interfaces. Classes define 25,279 methods and 12,852 fields. Enumerations define 77 methods and zero fields. The three Annotations defined in the project, define three parameters altogether. In the following, we use this project present exemplary reports with the Hypermodelling approach.

[1] http://www.alfresco.com - 7.11.2012

7.3. Inheritance Overview

We present the amount of total inherited types and distinct inherited types in Table 7.1. Types that get inherited are divided into superclasses and superinterfaces. The total inherited types are the amount of children classes. The distinct types depict how many different types serve as supertypes. The supertype and superinterface row indicates the kinds of used supertypes. It can easily been seen that more classes are extended (2,798) than interfaces are implemented (1,435). We compare the distinct types and see that the situation is different for those. The diversity of classes is less as that for interfaces, but they get more often extended than interfaces get implemented

Table 7.1. Which Kind of Type is used as a Supertype

	Total inherited types	Distinct inherited types	Implementation ratio (Total/Distinct)
Superclasses	2798	377	7,42
Superinterfaces	1435	625	2,3
Total	4233	1002	4,22

We use the indicator implementation ratio. With it, we enable a more concrete comparison of distinct supertypes to the amount of subclassed types: The indicator enables to compare the amount of distinct types with the total inherited types in one number. Since classes or interfaces that implement or extend other types add commonly new functionality to the existing one, they vary the original types. Therefore, we see figure of implementation ratio as one indicator to measure the variance of supertypes.

The figure of the implementation ratio is useful to depict a standard variation. A high or low value is a first indicator how intense supertypes are varied. When the figure is at a high value, we can conclude that the defined standard of the supertype gets aligned a lot. This means that the same types are often implemented or extended. Every time a type is used it gets adapted to the specific application needs. This way, developers have a starting point for further investigations to determine the types, responsible for high variance. This is especially useful, when varied supertypes are updated. If supertypes with a high ratio get updated, many children will be dependent on them. Thus, it is recommendable to investigate how the children adjust the supertype to keep its future version compatible to the inheritors. Furthermore, framework manufacturers can use the ratio indicator to determine which types are mostly adapted. With that information they can investigate how most developers varied the types. It is possible to depict if there is a common use or functionality in the extending types. If so, this functionality can be encapsulated into a new version of the supertype.

Additionally, we provide another perspective in Table 7.2. There, we show what type of children inherits a supertype. As before, the figure of the implementation ratio is presented. It is important to note that this table does not distinguish between type of the supertype, i.e., if the supertype is a class or an interface.[2] In total, most of the implementing types are classes, followed by interfaces, and enumerations. Likewise, this is the same for the distinct inherited types.

As a matter of fact, the implementation ratio is very interesting in this table. We assume indicators are led by classes, but get surprised by enumerations. We can see that 128 enumerations exist in the project and extend only four different supertypes. Since the number of implementing interfaces is in total not much higher (171), we do a drill down (refinement in the hierarchy level). There, we see that 104 children are directly derived from the Enum class. 18 from the EnumLabel enum, two from the Comparable interface and two from Serializable interface. Enumerations that have no supertype are derived, by the Java language specification, from the “pure” enum supertype. Therefore, the enumerations show a higher diversity than classes or interfaces. However, we learn from the abstract views of Table 7.1 and Table 7.2 that the implementation ratio of classes exceeds interfaces. This means: Classes seem to have a higher variance then interfaces.

[2]We leaved this out to avoid a complex anything with anything table.

Table 7.2. Which Kind of Types are the Children Types

	Total inherited	Distinct inherited	Implementation ratio (Total/Distinct)
Implementing classes	3934	967	4,07
Implementing interfaces	171	75	2,28
Implementing enumerations	128	4	32
Total	4233	1002	4,22

7.4. Inheritance and Packages

We present a more detailed view of the variance of inherited types at the package level in this section. First, we describe the source data that we used to generate various diagrams. Secondly, we present an analysis which package is used most in total numbers. Afterwards, the same analysis is done for distinct numbers. Finally, we show an overview that depicts packages with a high implementation ratio.

7.4.1. Source Data

Table 7.3 presents an excerpt of the source data that we use to create our reports. The table shows a query for packages and inherited type indicators. The package hierarchy is expanded to a level where a developer can derive the original or meaning of the package. In total, we split the report into 38 packages that contained the 1,002 distinct supertypes.

We show, that the supertypes used within the project are from the Alfresco project itself. Nearly, half of the total used types is from org.alfresco. Likewise, over ¾ of the distinct types are out of the project itself. Therefore, we compute the implementation ratio without the project itself: It increases over 11.5.

Remarkably is also the ratio for the java.lang package. We perform a drill down into the used types and discover that 1,195 classes inherit the object class. Likewise, there are enumerations extending the “main” Enum and other classes that are extending language functionalities. We conclude that also the java.lang package should be excluded. When it is excluded the average implementation ratio drops down to 4.68. These operations are comparable to drill down, roll-up, and drill-across in the Data Warehouse domain. This illustrates the application of the Hypermodelling approach. Note that all indicators are directly computed within the cube and only “navigation” is required.

Table 7.3. Exemplary Source data of Inheritance at the Package level

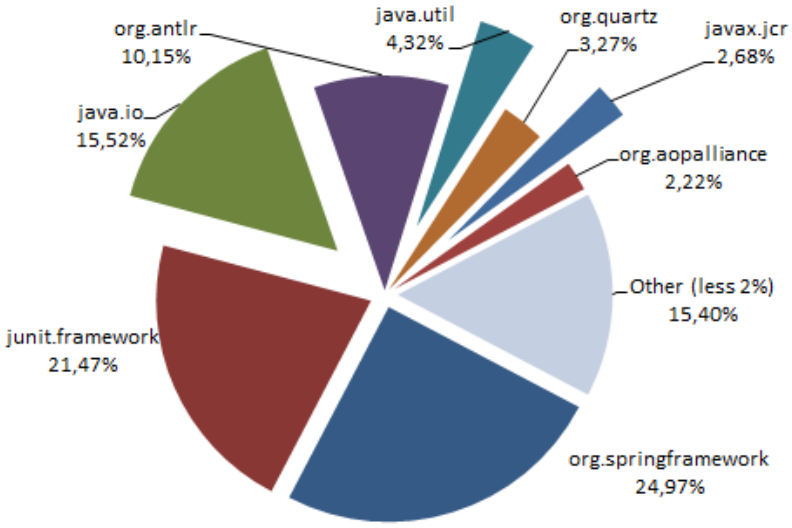
Package hierarchy	Total inherited types	Distinct inherited types	Implementation ratio (Total/Distinct)
org.alfresco	1972	808	2,441
java.lang	1404	11	127,64
org.springframework	214	40	5,35
junit.framework	184	2	92,0
org.antlr	87	6	14,5
Total	4233	1002	4,22
Without org.alfresco	2261	194	11,65
Without org.alfresco and java.lang	857	183	4,68

7.4.2. Dependency on Total Packages

In Figure 7.1, we visualize dependencies of the Alfresco project on super types. As described before, the java.lang package and the Alfresco package are dominant, therefore we kept them aside. The three emphasized slices, java.io, java.util, and javax.jcr visualize dependencies on java.lang itself.

We can also identify a heavy type usage of the spring framework. Furthermore, we can conclude that the index code base contains unit tests, since nearly a quarter is depending on the junit framework.

Figure 7.1. Dependency on Total Used Types

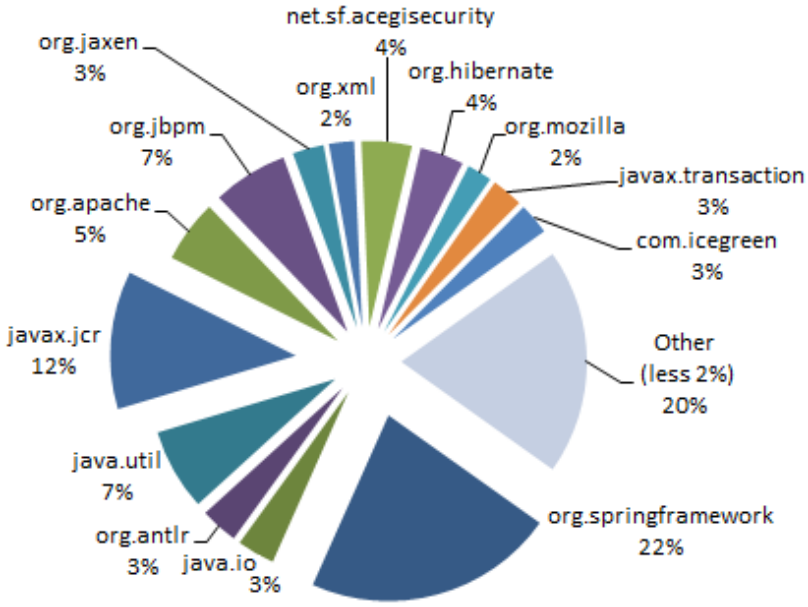


7.4.3. Dependency on Distinct Packages

Figure 7.2 visualizes the distinct used types within the project. The two main used packages are the Spring framework and the javax.jcr package. The jcr package is widely used, because the analyzed application is a content management system.

Jcr contains the java content repository types. The Spring framework is used because it is a web and Java enterprise application. All together it is interesting that the chart of distinct types differs from Figure 7.1. For instance, the Junit framework that is extensively used does not occur, because not many different types of it seem to be used.

Figure 7.2. Dependency on Distinct Types



7.4.4. Package Variance

In order to visualize variances of the packages we need a different kind of graphic that does not limit the visualization of one variance on the costs of another. Therefore, we use a bar chart. Figure 7.3

shows a bar chart of the implementation ratio of the different packages in the project. We only show three packages with the ratio of one as proxy out of space issues. Furthermore, we excluded the java.lang package and the junit.framework package out of their high indicator value. Such outliers would crush the diagram view and differences would hard to be recognized. Lastly, we show the average implementation ratio line with value 4.68 (mean).

Figure 7.3. Implementation Ratio for Packages

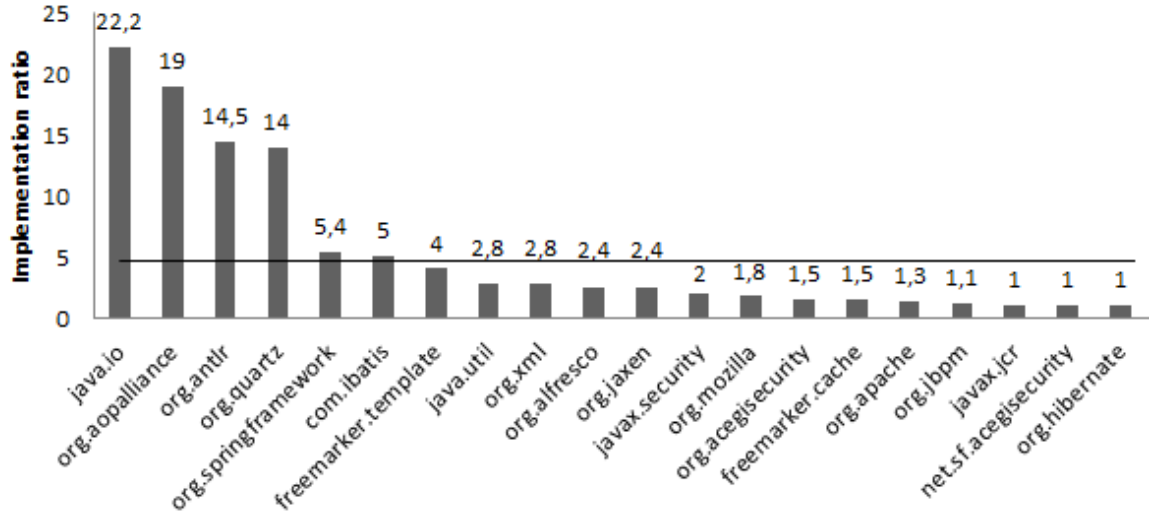


Figure 7.3 enables to depict the packages containing types that have a high variance in their implementation. Therefore, we can see that java.io has a high variation. Likewise, not much distinct types are used often of the aopalliance package. Probably, these are the aspect enhancements. Antlr provides a grammar parser. The high ratio indicates that various grammars are parsed. Additionally, quartz offers functionality to built timers into the application, what indicates that the Alfresco application uses timers.

Lastly, it is interesting that ibatis, a database table-class mapper, and the Spring framework are used at an average ratio. We interpret this out of the circumstance that spring and ibatis are large frameworks. Thus, the total amount in the package of spring and ibatis is much larger what leads to the different ratio.

At this point it would be necessary to compare the children packages of the frameworks with the packages of quartz and so on. We do not show them, to stick to the main focus to demonstrate exemplary applications of our approach.

However, like all other figures, the data of Figure 7.3 has been retrieved with a simple MDX-Query. We show the query to give an impression that not much effort is needed to retrieve the data in Figure 7.3.

Example 7.1. The query of Figure 7.3

```
with member [Implementation Ratio] as
    [Measures].[Type Inheritance Count] -/
    (count( ([Parent -- Type].[Id].children,
        [Measures].[Type Inheritance Count])-, EXCLUDEEMPTY)

SELECT {[Implementation Ratio]} ON COLUMNS,
    {[Parent -- Package hierarchy].[Parent].[All].children} ON ROWS
FROM [Inheritance-Annotations-Cube]
WHERE ([Package Hierarchy].[Parent].&['org.alfresco'])
```

7.5. Method Variance Drilldown

With the information that is uncovered in the previous section we know that a huge amount of types of the Spring framework is used within the application. Furthermore, a few types out of Junit are intensively used. We present the top used types of the two packages in Table 7.4. Three columns are already known and show the origin package of supertypes, supertype name, and total inheriting types. We introduce a new indicator of distinct method names in inheriting types to credit our drill down. This measure counts the distinct method names occurring in types. This way, we see how many types subclass a supertype and how much a supertype is varied in total.

Clearly, this enables to create an abstract indicator similar to the implementation ratio at the type level. We do not show it to focus on further possibilities to stick new ways of exploration.

Table 7.4. Excerpt of the Top Used Types of the spring and junit package

Supertype package	Supertypename	Distinct method names occurring in inheriters	Total Inheriting
org.springframework	ApplicationContext Aware	580	26
	InitializingBean	578	56
	AbstractLifecycle Bean	351	42
	ApplicationListener	175	7
	… in total 40 types	… in total 2374	… in total 214
junit.framework	TestCase	1326	170
	TestSuite	2	14
	Total 2 types	Total :1328	Total: 184

7.5.1. Method Variance

In order to explore the concrete usage of the InitializingBean and TestCase in detail, we generate a report for supertypes and method names of their children. We present an excerpt of the report, sorted after the most implemented method name, in Table 7.5, what we discuss in the following.

Table 7.5. Excerpt of the Most Used Method names in Supertypes Children

Supertype package	Supertype	Methodname	Distinct inherited types (i.e.: extenders sharing the method name)
org. springframework	InitializingBean	afterPropertiesSet	55
		setAuthentication Service	10
		setNodeService	9
		setBeanName	7
		destroy	6
		… in total 578 distinct member names	Total distinct types: 56
junit.framework	TestCase	setUp	139
		tearDown	73
		testSetUp	35
		create	6
		… in total 1326 distinct member names	Total distinct types: 184

The column distinct inherited types, in Table 7.5, shows the amount of children types that use the same method name. For instance 55 of 56 children of the InitializingBean use the name

afterPropertiesSet for a method. We also see that a huge amount of 578 method names for the InitializingBean and 1,326 method names for the TestCase turn up in their subclasses. Currently, we can depict common method names from the table. However, we cannot derive if the intention of a developer is actually to override a method of a supertype. Therefore, we show in the following section how we enhance this approach with source code metadata to get common methods for improved reporting.

7.5.2. Slice by @Override

Before, we address the problem of the immense amount of different method names within the children of a class, we propose to solve this problem by the usage of metadata annotations that are defined within the Java programming language [46]. Java's annotations allow developers to enrich source code with various structural and logical information. Specifically, the @Override annotation out of the Java standard is of interest in our current case. The Java standard describes the Annotation usage as follows:

@Override. "Indicates that a method declaration is intended to override a method declaration in a superclass. If a method is annotated with this annotation type but does not override a superclass method, compilers are required to generate an error message."[3]

This means that the override annotation can be used, but there is no compulsion to do so. Thus, @Override marked methods override a method for sure, but overriding can also take place without this annotation. In fact, modern development environments, like Eclipse[4] remind programmers through messages and markers to use the annotation.

Hence, we assume that most overriding methods are marked with the annotation in modern programs. We query how often the annotation is occurring on methods of the Alfresco project. 2,067 methods are annotated with this annotation. In total 3,139 annotations are occurring at methods of the analyzed project. Therefore, we assume that we can use annotation to specialize our former query.

7.5.3. The Report Filtered by @Override

We present the modified query result for the @Override marked methods of children of TestCase and InitializingBean in Table 7.6 that we describe in the following.

Table 7.6. Excerpt of the Most Used Method names in Supertypes Children

Supertype package	Supertype	Methodname	Distinct inherited types (i.e.: extenders sharing the method name)
org. springframework	InitializingBean	equals	2
		Implementation AllowsGuestLogin	2
		toString	2
		transformInternal	2
		afterPropertiesSet	1
		… in total 21 distinct method	Total distinct types that got an Override Annotation somewhere: 15
junit.framework	TestCase	setUp	95
		tearDown	53
		… in total 1326 distinct member names	Total distinct types that got an Override Annotation: 113

Table 7.6 shows, in contrast to our expectation, the most overridden methods of InitializingBean are not marked @Override. In fact, the annotation is used scarcely, like the distinct inherited

[3] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Override.html - 7.11.2012
[4] http://eclipse.org - 7.11.2012

type numbers indicate. Nevertheless, we get the methods that are intended to be overridden by developers. The most interesting fact about the distribution of the @Override annotation is that 21 distinct types seem to use the override annotation at different methods. Only at four methods the override annotation is occurring at the same method name. In contrast to the InitializingBean children, project developers seem to have been more sincere about annotating test cases with the annotation. 113 types out of 184 in total got at least one annotation.

Lastly, the result gets even more interesting, when regarding at the Javadoc of the two parent classes. InitializingBean defines only one method: afterpropertiesSet[5]. TestCase[6] defines the two @Override marked methods. Hence, InitializingBean extenders override methods out of the supertypes supertype.

7.5.4. An Additional Type-Method Report Filtered by @Override

Surprised, by the previous results of overridden methods, we depict further inherited (children) types that have a huge amount of methods with the same name (see the following Table 7.7). Additionally, we investigate the methods that are defined by the supertype themselves and compared those method names with the ones defined by the inheritors. In the following, we present different types shown in Table 7.7.

Table 7.7. Common Inherited Types and @Override marked Methods in Children

Supertype	Methodname	Appearance in children Types	Exists in Supertype?	Additional methods in supertype
Serializable	toString	42	-	-
	equals	33	-	
	hashCode	32	-	
	getInviteeEmail	2	-	
	getInvitee FirstName	2	-	
	getInvitee LastName	2	-	
	clone	2	-	
	get InvitationType	2	-	
	initialiseHandler	1	-	
Transaction Listener Adapter	afterCommit	21	X	flush; beforeCompletion
	afterRollback	11	X	
	beforeCommit	6	X	
	equals	4	-	
	hashCode	3	-	
	toString	1	-	
Entity Lookup Callback DAOAdaptor	updateValue	5	X	deleteByValue
	findByValue	5	X	
	getValueKey	5	X	
	deleteByKey	2	X	

Serializable is a well known Java flag interface. However, the Serializable interface does not define any methods. It is interesting that inheritors seem to be attracted to override toString, equals, and hashCode that are originally coming out of the object class. Thus, if we ask an Alfresco programmer which methods serializable class needs, he will probably answer: toString, equals, and hashCode.

[5]http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/beans/factory/InitializingBean.html - 7.11.2012
[6]http://www.junit.org/apidocs/junit/framework/TestCase.html - 7.11.2012

The TransactionListenerAdapter offers five methods whereby three get overridden by subclasses. All together, it seems if the after the transaction methods are more common to be adjusted by the application logic. Methods of the Object class get overridden what brings up the idea that those should maybe excluded in a query if the focus is purely set to be on other types.

The EntityLookupCallbackDAOAdapter is highly customized within the application. This is quite logical, since data access objects (DAO) are used to persist different classes of the domain model of an application. All together is seems that the delete methods seem to work more generically than the retrial or update methods since they do not get adapted this much.

7.5.5. Method Variance Roll Up Enriched with @Override

Overall, we learn that override annotations are used within the project. Even if they are not applied consequently the amount of 2,067 annotated methods is immense. Hence, we present a roll up to all types that contain overridden methods.

Report Source Data

Table 7.8 shows an excerpt of the report that we generate through a roll up from the Spring and Junit package. The distinct inheriting types specify the amount of types that extend or implement the supertype. The overridden methods specify how many methods of the children carry the @Override annotation. The total methods amount is sliced by the method names (distinct method names). The C and I indicate if the supertype is a class or a method. Lastly, we introduce the overridden method implementation ratio. This ratio indicates the average amount of overridden methods per type. A higher ratio means that more methods in total of the type get overridden and the variation of the children is probably higher. Such, framework manufacturers can determine the types that get varied the most by developers.

Table 7.8. Excerpt of the Most Used Method names with @Override

Class/ Interface	Type name	Distinct inheriting types	Total overridden methods	Distinct method names	Implementation ratio (Total methods / Distinct types)
C	Object	184	353	47	1,918
C	TestCase	113	150	2	1,327
I	Serializable	42	88	9	2,095
C	Transaction ListenerAdapter	26	46	6	1,769

Relations with Figures

We present how different figures of the report relate to each other in Figure 7.4. We eliminate outliers from the source report data to have a clear distribution of the points of the data: We exclude the Object class that serves as parent class, if no other is given and TestCase out of its extreme usage. Additionally, we exclude types having less children then three to avoid an intense cluster. All together, 63 adequate values left as base for visualization. Founded on those values, we present the charts to have a first look for trends that maybe excel out of the diverse variance figures.

Figure 7.4 a shows the relation of the total amount of overridden methods to inheriting types. We see that the amount of overridden methods grows with the types. We added a linear trend based on these values. The big picture is that a linear growth seems to exist. We conclude that it seems when types inherit a supertype they also override methods.

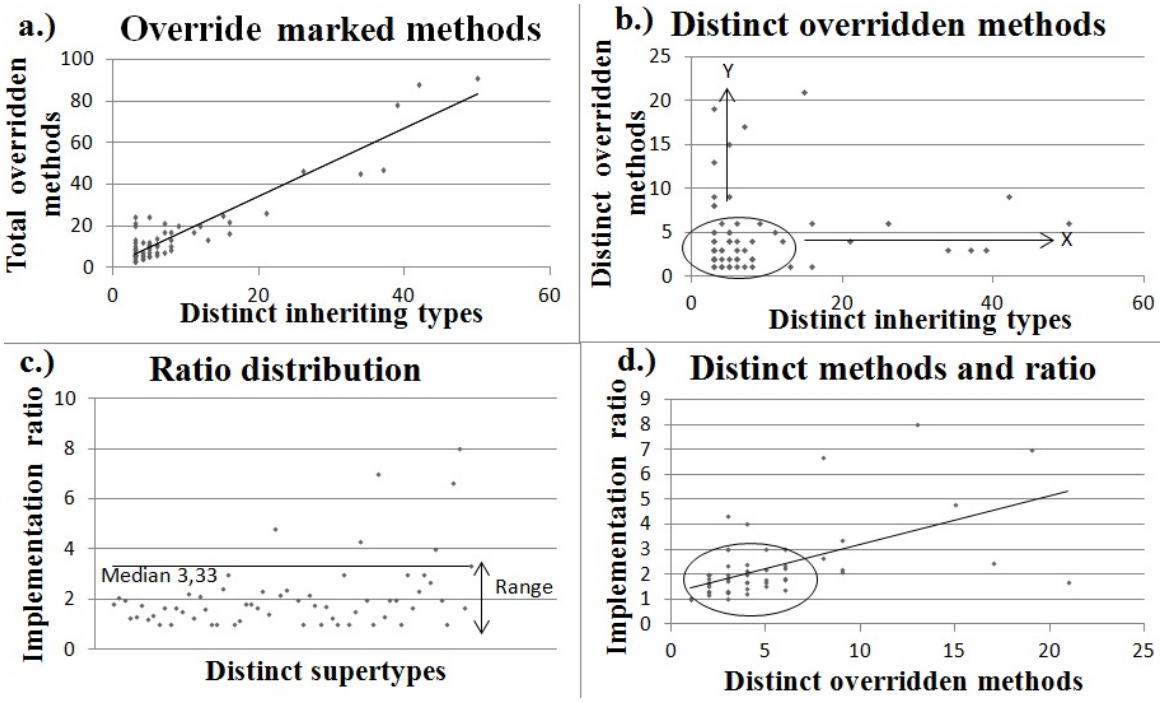
Figure 7.4b shows the relation of inheriting types to distinct overridden methods within the children types. We assume that the circled space marks a cluster. This would be logical, since we guess that the amount of methods of a type is in most cases within a certain range. However, there are outliers in the X and Y direction. The Y direction seems to be the case for a tiny amount of inheritors, what means: A large amount of methods of supertypes is overridden, but not many other types inherit

from the same supertype. In contrast to the Y direction, the X direction shows many inheritors that map to a small amount of distinct overridden method names.

We present the implementation ratio distribution for various types in Figure 7.4-c. The inheritance ratio values are sequenced following the amount of the inheritors of a type. For us it seems that the ratio values are mostly arranged within a certain range and a few outliers exist. All the more, the median of all values is arranged at 3.33, what is on top of the guessed range.

Out of the guessed range of the ratio before, we reorder the ratio values in Figure 7.4d. The ratio expresses the average overridden methods for a type. Since distinct method names are not expressed in this relation, we order the ratio values following the amount of types in Figure 7.4d. The outliers seem to grow in value in relation to methods. In the graphic we show a linear trend for this. However, the trend is not supported by many values to make any predictions. There exists a cluster that we emphasize through a circle. This cluster seems to map the prior mentioned range in Figure 7.4 c to one spot. Thus, we assume all our values seem to have a relation and be dependent on each other. Though, more detailed investigations are required to verify our assumptions we depict from the diagrams.

Figure 7.4. Implementation Ratio - Different Angles



7.6. Dependencies Investigation

Right before, we described dependencies based on inheritance. Now, we focus on the use case of method invocation dependencies and explain practical uses cases for this analysis.

Literature reveals beneficial design factors to built dependencies in the direction form unstable to stable or to create mostly independent components that have no dependencies to each other [169,170,171, 172, 173]. One reason to do so is that stable packages are summoned with function calls or inheritance from the unstable packages. Hence, when the stable packages are altered, a lot of implications follow up. Also, call based dependency measurement is likely of interest, when we imagine that an application should been sliced into different subprojects. Thereby, developers face the challenge to discriminate which packages are dependent on each other. This kind of dependency information is then used to decide how strong the packages coupled and if they can be separated or if they have to be shipped together.

Furthermore, managers might ask questions about migration effort and the dependence of parts of source code to various libraries. Imagine an application is programmed with hibernate and you want

to switch to ibatis for persistence.[7] In order to do any time planning of such a migration project, a manager would first like to gain at least an overview how strong the application is coupled to the hibernate package. "Which packages are using hibernate - is the use scattered over my whole application? Do we use many method calls to the hibernate classes?" Those are questions that a manager might ask to get a first insight about the potential effort.

However, in the following we illuminate a few points about dependencies without having the claim of completeness or a full discussion. Our main goal is to discuss a few aspects about dependencies and show how Hypermodelling can be used to investigate such issues. Again, we use the prior mentioned excerpts of the alfresco project as samples.

First, we focus on internal dependencies within the packages of the application and show general statistics about them. Then, we show how those can be utilized to built a dependency graph. We follow up, by describing the possibility to drill down into the concrete dependencies on specific types. Afterwards, we lift the viewpoint and describe the dependencies on libraries what can be used to determine indications about vendor lock ins. Finally, we do a small conclusion about the scenario and the usage of Hypermodelling for that case.

7.6.1. Internal Package Dependency

In order to reveal the dependency of packages of the alfresco application, we compute which packages have been called by which package. Thereby, we exclude classes and packages of external libraries and focus purely on the internal dependencies within the project itself. This means, we created a cross table where the vertical package calls the horizontal package. We show an excerpt of the result in Table 7.9. There, we see that 43 method calls in the org.alfresco package are calling methods within the same package. The org.alfresco.util package calls one method out of the encoding package and 10 methods out of the org.alfresco.error package. Equally, the rest of the table can be interpreted.

Note, we took only the top level packages to ease comprehension and spare complexity. Therefore, the calls deeper within the hierarchy have been aggregated to the top level packages.

Table 7.9. Package Dependency Matrix

Number of calls Origin/destination	org.alfresco	org.alfresco.config	org.alfresco.encoding	org.alfresco.error	...
org.alfresco	43	-	-	-	...
org.alfresco.config	-	4	-	-	...
org.alfresco.util	-	-	1	10	...
...	...	...	...	...	...

Out of Table 7.9, we compute the total calls of a package into other packages (this means the totals of a row) in Table 7.10. Therefore, we get the number of internal calls and the amount of total calls. The last column of the table expresses the percentage that is dependent on other packages. This measure is the ratio of the calls within the package to the total calls out of the package. This ensures that we respect the size of the package and get a percentage ratio. Then, we subtract the result from '1' to have the percentage of calls that depend on other packages. This means, the dependency ratio expresses the degree of independence of a package from other packages.

We visualize the dependency ratio in Figure 7.5. There, we see that the org.alfresco.filesys package is with 67% of all method calls depending on other packages. In average, the median dependency on other packages is 48,1%. Some packages, like org.alfresco.encoding or org.alfresco.hibernate have no method calls to other packages within the org.alfresco hierarchy. Hence, we depict, easily, that

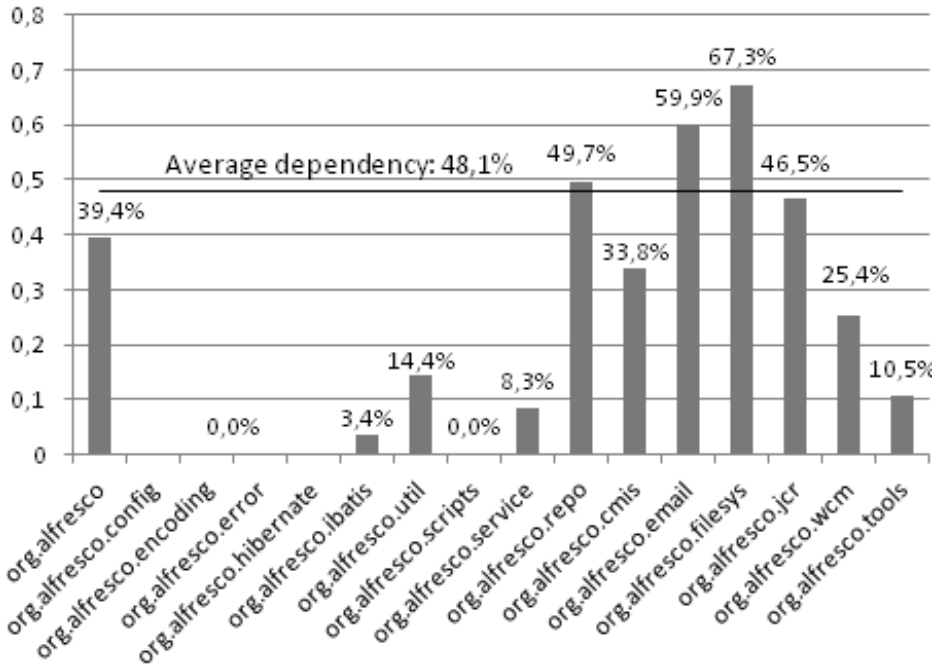
[7]Both are persistence frameworks for Java.

those packages can exist without any other packages of the application. Additionally, if we want to extract the packages with a high dependency percentage in their own projects, we can estimate that the effort will be likely higher as for the packages with a low dependency percentage. Furthermore, we can use this diagram to decide for which packages more detailed analysis are required.

Table 7.10. Dependency Ratio

Package	Self calls	Total calls	Dependency ratio
	(In-package)	(in+out package)	(1-selfCalls/Totalcalls)
org.alfresco	43	71	0,394
org.alfresco.config	4	4	0
org.alfresco.encoding	3	3	0
org.alfresco.error	11	11	0
org.alfresco.hibernate	1	1	0
org.alfresco.ibatis	28	29	0,034
org.alfresco.util	440	514	0,144
org.alfresco.scripts	4	4	0
org.alfresco.service	33	36	0,083
org.alfresco.repo	34869	69343	0,497
org.alfresco.cmis	4064	6135	0,338
org.alfresco.email	91	227	0,599
org.alfresco.filesys	1343	4102	0,673
org.alfresco.jcr	820	1532	0,465
org.alfresco.wcm	2475	3319	0,254
org.alfresco.tools	119	133	0,105
Total	44348	85464	0,481

Figure 7.5. Dependency of Packages as Bar Chart



We see the different packages of the loaded source code and how much they depend on other packages of the application. Packages with a high percentage have a high dependency on other packages. A low percentage means they are relatively independent.

7.6.2. Dependency Tree

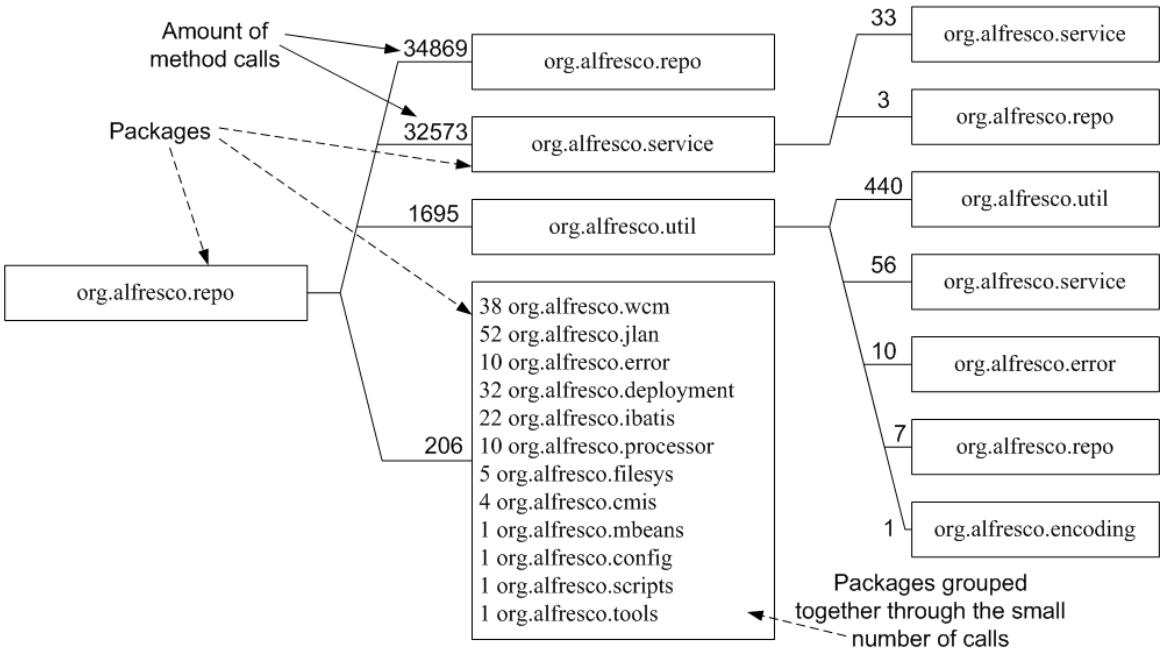
Out of the prior immense amount of method calls, we provide additional visualizations about dependencies. Therefore, we give an example how such dependencies can be visualized in a dependency tree. We computed the tree out of Table 7.9 in the prior section and show it in Figure 7.6. Right before each node of the tree, we specify the amount of incoming method calls form the calling package, arranged higher in the hierarchy.

For the sake of a better overview, we group 206 incoming method calls in a larger box together. We also add a second level of dependency on this main called packages. This way, we can see that the org.alfresco.repo package is heavily dependent on the org.alfresco.service package and. The org.alfresco.service package is mainly depending on itself and with a small amount of calls on the org.alfresco.repo package. This way, we know that the org.alfresco.service package could be grouped together with the alfresco.repo package together without inflicting further dependencies. However, the org.alfresco.repo package is also dependent on the org.alfresco.util package. We can see that the org.alfresco.util package, itself, is depending on plenty of other packages.

All together, such a graph can be used to compute also cycles and similar things to determine which packages are highly dependent and which ones are likely to be separated without a lot of effort. Additionally, such visualization can be used to control easily if the dependencies are from unstable to stable packages. The concrete applications on top of this package tree are beyond the scope of this chapter, since we rather focus which foundation analyses are generally possible with Hypermodelling. Therefore, back to topic.

The org.alfresco.util package depends to a high degree on itself (440 calls) and the dependency on the other packages varies from 1 to 56 method calls. We imagine that out of those 56 methods calls, the decision is made to do an in-detail exploration about the dependencies with the potential goal to get rid of those. Hence, we see the need to present the possibility to explore the dependency on a more detailed level. We present a drill down to the type level in the next section.

Figure 7.6. Dependency Tree



We show the dependency of packages by method calls. We see how often one package calls another. For the sake of better overview, we group the packages with less than 53 method calls together.

7.6.3. Type Drill-down

Like we discussed before, it can be useful to drill into the specific methods that are consumed by a package to investigate the low level dependencies. Now, we use the dependency on the

org.alfresco.wcm package. We do this, because some investigations of the dependencies of the org.alfresco.util showed up to be harder to explain then the usage of org.alfresco.wcm for what we show in and explain the following.

In detail, we recognized that a lot of different types were used in org.alfresco.util and in case of the org.alfresco.wcm package it were less different types. Therefore, we switch the package, because it is much easier to show the org.alfresco.wcm package.

Table 7.11. Method Calls to Types of the wcm package

Caller	Called	Called types	Method Calls
org.alfresco.repo	org.alfresco.wcm	WCMUtil	17
		SandboxService	5
		AbstractWCMService ImplTest	4
		PreviewURIService	2
		WebProjectService	2
		AssetInfo	1
		SandboxFactory	1
		WebProjectInfo	1
		SandboxInfo	1
		AssetService	1
Total		10	38

In Table 7.11, we show a query which specific types are called how often. We see that the far most often called class is the WCMUtil. As a matter of fact, 17 times gets this class called from the org.alfresco.repo package. All together 10 different types are called from the org.alfresco.repo package. In order to separate the org.alfresco.repo package from the org.alfresco.wcm package, these types would need to be available in the org.alfresco.repo package.

Note, this number is not defined as distinct methods. So it can be also 17 times the same method. Generally, also queries for distinct methods can be done with Hypermodelling. Here, we focus on the totals to stick to the approach what kind of investigations can be done.

By moving those 10 types into the org.alfresco.repo package, the dependence to the org.alfresco.wcm package is resolved. But even though, there is also the problem that the 10 different types of the org.alfresco.wcm package have dependencies on other types, themselves. For instance, the types of the org.alfresco.wcm package call other types. Such issues can be resolved with queries again. We show an exemplary query result for the called methods types of the WCMUtil in Table 7.12.

There, we see that the destination of the calls of WCMUtil class is located within diverse packages. The highest amount are self calls (30), followed by the AVMUtil (9 calls). This way, we can go down the whole dependency hierarchy to determine the degree of dependence. Right now, we assume that one key point in such an investigation is that resolving and moving dependencies is not necessary, when interfaces are called and not concrete types. We assume this, because, often it is considered not as coupling, if method calls are programmed against an interface. Therefore we show the option that the Hypermodelling approach can be used to indicate what kind of type is called. The query result in Table 7.12 presents an indicator if a class or interface is summoned.

Sadly, we see that mostly a dependency to classes exists. Those can have calls to others and we could use queries to follow down this path to resolve all called types. With similar queries like before all of

those dependencies can be resolved. This process can be repeated until the complete dependencies are resolved. All together, those facts help to gain insights about the possibility to move the types into other packages and to determine the multi-leveled dependencies easily. The determined graph can then be used to define cut off points, where the couplings are replaced with other types or logic.

Table 7.12. WCMUtil class method calls

Package	Called Class	CalledTypeKind	Calls
org.alfresco.wcm	WCMUtil	class	30
org.alfresco.util	VirtServerUtils	class	3
org.alfresco.service	QName	class	2
	NodeService	interface	3
	ChildAssociationRef	interface	1
	AVMService	class	2
org.alfresco.repo	PropertyValue	class	1
	AVMUtil	class	9
org.alfresco.mbeans	VirtServerRegistry	class	3

7.6.4. Library Dependency

Right before, we described how queries can be used to investigate dependencies on concrete types. Now, we lift the view to gain insights about the dependencies on external libraries. Often managers fear vendor lock ins. They want to know how much their systems are depending on artifacts of specific vendors or how much the effort would be to switch to a solution of another vendor. In the specific case of software development, the vendor lock in can appear by consuming functionality of specific libraries. Often there is the desire to estimate how much effort it is to switch to other libraries or to get a general overview of a project and its dependencies.

We query the amount of calls from the org.alfresco package to the different library packages and present the result in Table 7.13. We also compute the percentage of the total calls to indicate different relations better.

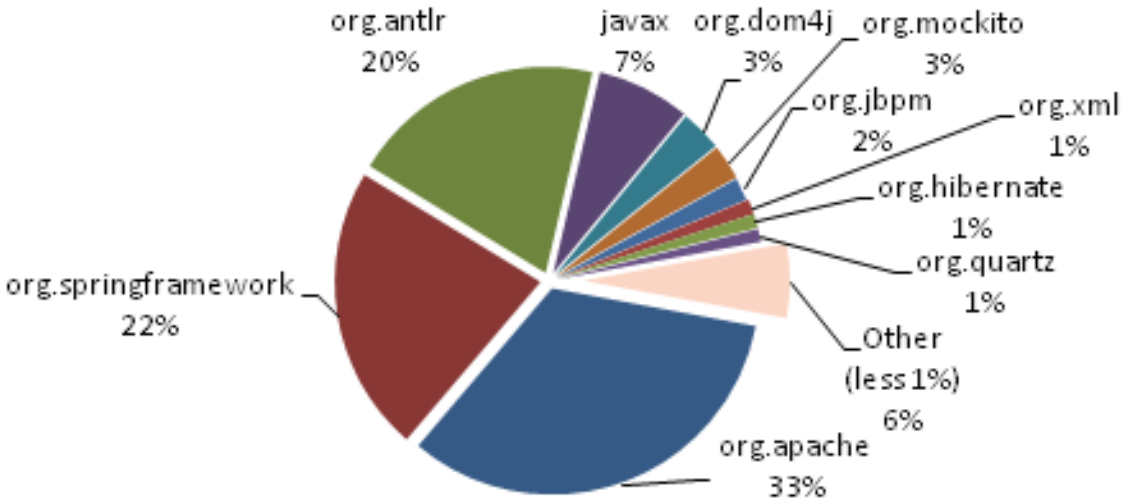
Table 7.13. Excerpt method calls originating the org.alfresco package

Called Package	Calls from org.alfresco	Relative to total calls
org.alfresco	85464	0,553731324
java	35155	0,227773386
junit	16106	0,104352671
org.apache	5871	0,038038901
org.springframework	3955	0,025624911
org.antlr	3517	0,022787057
javax	1290	0,008358062
org.dom4j	575	0,003725493
org.mockito	488	0,003161809
org.jbpm	317	0,00205388
…	…	…
Total 44 Packages	154342	1

Because of the low number of method calls to many packages, we only show them partially. All together 44 vendor packages are called in total. Hence, the table shown is just an excerpt of the original data.

We use the data of Table 7.13 as source to generate a graphical overview for managers, what we show as pie chart in Figure 7.7. We excluded the org.alfresco, the java and the junit package out of their dominance what results in some the percentage shifts to the original data. We do this, because logically, the application is dependent on the java language and its own application logic. Furthermore, junit is just responsible for the tests and not application logic. Thus, we see in the figure that the org.apache package is called most often. This is followed by the spring framework and antlr. Then the dependency rate drops down to 7% for javax package.[8] However, also the block of other called packages (less then one percent) shows that a lot of different packages are called, but not intensely. We have clearly dominant called packages within the application and potential vendor lock ins of spring, apache and antlr.

Figure 7.7. Called Vendor Packages



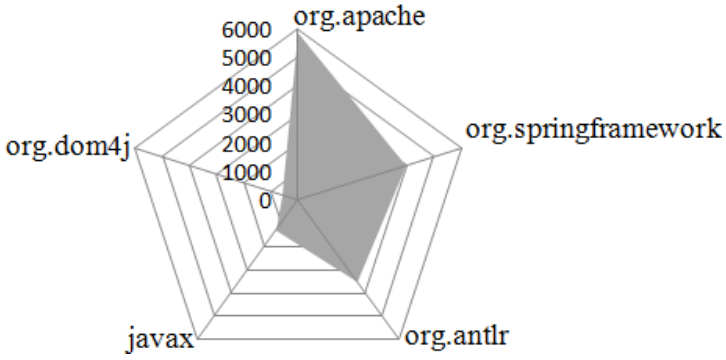
We present a pie chart that shows the most called packages that are not part of the application itself. We see that the usage of org.apache is dominant, followed by spring and antlr. Managers can now depict easily on which libraries their application depends most.

However, the taste for figures and visualizations is always different and we credit this by proposing an additional kind of chart. Since the huge difference between the method calls to the different packages is so immense, we chose to present it in a net diagram to emphasize the difference. We use the first five values of the pie chart and visualize the number of method calls in Figure 7.8. We see the intense difference between the amounts of method calls within the application. If a manger sees this chart he can easily depict that (hopefully) a lot of project employees have an immense knowledge how to use the apache and spring package. We say this, because if libraries get used so widely like those, it is highly recommendable that developers know them in detail. Furthermore, if a company is in lack of lack developers, and wants to employ new ones, this chart can be used to define the desired skills of new team members.

In this moment, one may come up with the remark that we did the vendor lock in charts on the foundation of packages and not libraries. We credit and appreciate this thought and agree. In general, Hypermodelling can be used to do queries not only at the package and also on the library level (indicated by the technical description and the relational model). We just focus on the packages here, because the results are easier to see on this level. Clearly, in specific scenarios drill downs will be needed. Right here, we just showed the general approach that such issues can be investigated.

[8]We only excluded the java package and not the javax package, because it is not part of the jdk.

Figure 7.8. Top five called Vendor Packages



We present the top five most called packages within the application in a net chart. Here it can be depicted easily that the org.apache libraries are used so intense that the project members better have in depth knowledge of the corresponding libraries. Furthermore, to work in this project it is recommendable that spring knowledge is available at hand.

7.7. Method Parameters and Method Invocation

Before, we discussed that method based coupling is relevant for source code quality. Right now, we demonstrate a first example how we do statistics about method parameters.

We use method calls and parameters, because it is said in literature to be another quality indicator. In fact the usage of too many method parameters is considered to be a bad coding style [173]. We agree that methods with many parameters are hard to use, but ask the question about potential implications of many method parameters. Saying hard to use does not imply "getting used". That said, we consider the reuse of software components as a widely agreed quality indicator. Therefore, we assume that if many method parameters means, it is less frequently used, it violates that principle that reuse should occur often. If we can find indicators for a less reuse, when the amount of parameters grows, we have support for the thesis that it is in fact a bad coding style.

In order to present statistics that correspond to this question, we used the prior describe projects that we loaded into the Data Warehouse and executed a query. The query result set consisted of the method, the amount of parameters and how often the method is invoked within the code base. From this result, we computed two charts that we discuss in the following.

We present the first chart in Figure 7.9 on the next page. The dots represent methods, which get calibrated by their method parameters and the amount of their invocations. The amount how often the method gets called by other methods is the Y axis and many parameters a method has is X. We can see directly is that many methods define less than five parameters. We see also that for few parameters the possible invocation rate is in many cases higher then for many parameters. It nearly sees descending with the amount of parameters.As addition to the pure graphical impression, we compute a correlation. We exclude methods that are not invoked at all and get 15163 methods of 32806 in total. We exclude those, because we assume that they are probably called in other parts of the project that we did not load for the investigation. For the left 15163 methods, we compute a Pearson correlation of the amount of parameters and the amount how often they get called. As result we get: r = -0,009 and p = 0,246 . So we can say, that there seems to be no Person correlation significance. We assume that "no significance comes" out of the reason of a too small dataset and there may. Additionally, there are maybe more complex relations. Future investigations need to investigate this issue.

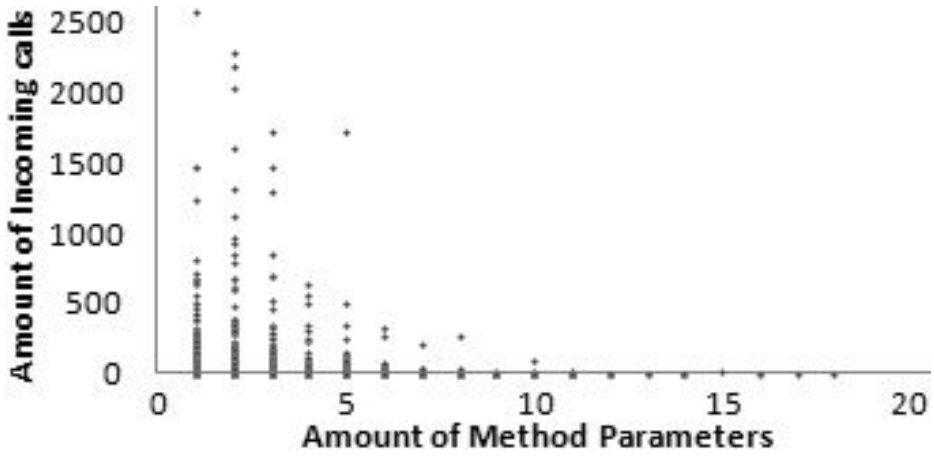
Hence, out of the prior interpretation, we cannot infer that methods with less parameters get used more often. We look in the data and see also that there are much more methods with less than five parameters invoked than with more. In order to consider different parameter amounts equally, we compute average "called" values for every amount of parameters what we show in Figure 7.10. There, we see one point for every amount of parameters and a linear trend line computed out of the

values. We recognize that methods that have eight parameters still get six invokes in average. If they have nine or more parameters, they get invoked at most four times. In general, we assume a decrease.

Again, we compute a Pearson correlation. For the average getting called values we get now a correlation of r = -0,861 and p<0,001 for 20 averages. Therefore, we see a first indication that methods with many parameters get less likely used then ones with less parameters.

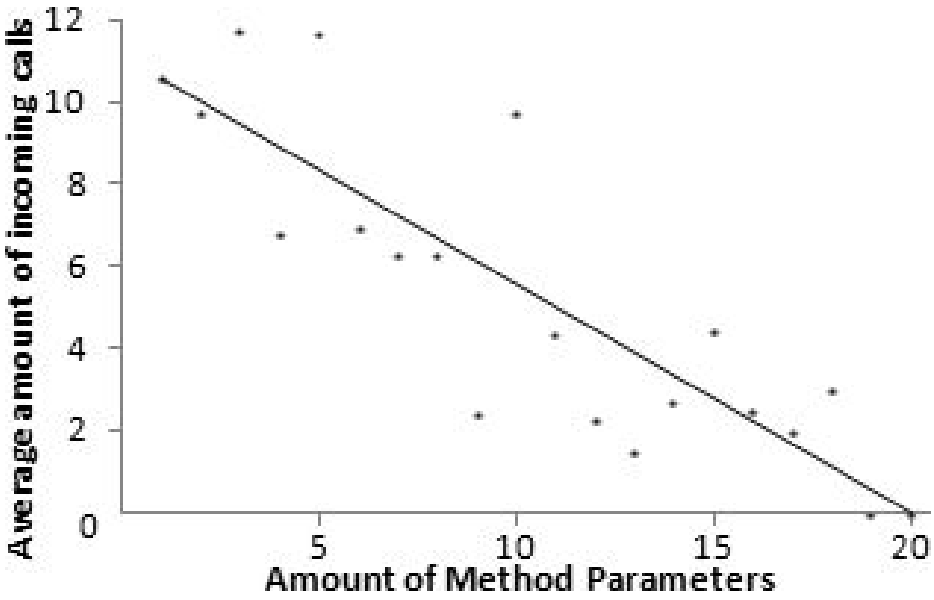
Anyway, we cannot know for certain from this few statistics, but we can see that we find support for the theory that methods with many parameters are not beneficial. Therefore, future research needs to investigate more and larger projects and do extended statistics about this case to reveal more exact numbers how many method parameters are recommendable.

Figure 7.9. Amount of Method Parameters and Invokes



Every point in the coordinate systems represents a method that is calibrated by their amount of parameters and how often it gets invoked by other methods. We see that more methods with less than five parameters exist than methods with more parameters. Also, the maximum amount of calls is decreasing with the amount of parameters.

Figure 7.10. Average amount of Method Parameters and Invokes



The figure shows the medium call frequencies for a set of methods that share the same amount of parameters. Furthermore, a linear trend line shows that the call frequency in average is going down when the parameters increase.

7.8. Summary and Conclusions

We presented different reports of the Alfresco application. In detail, we explored different types of parent – children relations with drill downs, roll ups other OLAP operations. Investigations about the method variance of subclasses revealed that methods of subclasses exceed probably a pure relation to supertypes methods. In special, we recognized that methods of a supertypes subclasses seem to share method names of another supertype. Furthermore, we also demonstrated that Hypermodelling reports can be discriminated with dynamic factors, like metadata annotations, to enhance results.

Additionally, we showed that dependencies of an application are of interest for software development. A concrete example showed that Hypermodelling is suitable to investigate the dependencies of a programs packages. This can be used to investigate which packages are independent from others and to reveal if there are potential vendor lock ins in the software. Now, researchers can create specific tools on top of our examples that allow easy dependency investigations. Software project managers can use our approach to dig into the insights of an application and get details about vendor lock-ins.

In addition, we gave a brief introduction about the usage of Hypermodelling to investigate statistics about other quality indicators of a program. Thereby, we saw first support that methods with many parameters get less likely "reused" (called) than methods with fewer parameters. Therefore, we assume that reuse of method logic is less done, when a method has many parameters.

All together, we are confident that Hypermodelling is suitable for software analysis. Further research needs to reveal more advanced scenarios how it can be best utilized to help to manage dependencies and to investigate the most desired facts.

However, the main intention of this chapter is to present how Hypermodelling can be applied to analyze software. The diversity of the reports showed that we were capable to use the Hypermodelling approach. We revealed plenty of facts about a program that would be complex and expensive to be uncovered with another approach. Hence, we see the evaluation if Hypermodelling is suitable for software analysis as successful.

Nevertheless, we see further advancements of code analysis in standardized reports. In order to create those, we see the reports in this chapter as origin for further developments. Therefore, we show the general possibility of Hypermodelling to create standardized reports in a dashboard in the following chapter.

8. Hypermodelling Reporting

“All we have to decide is what to do with the time that is given to us.”
Gandalf. Lord of the rings - The Fellowship of the Ring. 2001

This chapter shares information with:
"T. Frey, M. Gräf. Hypermodelling Reporting: Towards Cockpits for Code Structure. In Proceedings of Theory and Practice of Computer Science - 39th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM). Springer. 2013."[103]

Abstract. Do we have a vendor lock in? How many classes of a framework do we extend in our code? Do we have strong cohesion in our packages? These questions may be asked by software development managers. To reveal such facts, usually a lot of effort is needed. Even when using Hypermodelling for software analysis, query effort is needed. Therefore, we present the use of Hypermodelling to create software cockpits. We show a cockpit for software variance that is reflecting facts about the inheritance of types and evaluate the effort to implement it. Additionally, we present first insights about advanced reporting solutions for cohesion, mining developer knowledge and Style Issue distribution. This way, important indicators can now be investigated at a central spot. Project managers can now use cockpits to investigate software more easily. This avoids costly, time-consuming and deep investigations in the first place. Further research can reveal additional cockpits. Furthermore, the reasonable effort to create such cockpits enables the possibility to create, evaluate and compare different cockpits.

8.1. Introduction

Before, we introduced Hypermodelling and applied it to analyze source code with queries. In this chapter, we extend the query based application by presenting reporting solutions that are internally based on queries.

The complexity of software development is undoubted. Project managers face the problem to control the development process. In order to overcome this complexity, project teams often use issue tracking systems. Issue tracking systems enable the specification of desired functionality or needed bug fixes for future versions [155]. Researchers come up with solutions, leveraging Data Warehouse technology to control the data of issue tracking systems [184]. This enables managers to gain an overview by dashboards. This project control dashboards, also called cockpits, contain key performance indicators and charts about the project process.

Beside the issue tracking control many challenges are faced within the development of a system, too. Often parts of systems or whole systems themselves are created by offshore development teams without having senior developers checking the code quality. In case, the foreign code base is merged into the own repository, managers want to know which frameworks the foreign code is depending on.

Currently, programs need to be parsed to extract information about framework usage. That consumes time and effort and makes information mining inflexible and static. Furthermore, if you want to see the dependencies in a clarified way (e.g. dependency counters, statistics), this is hardly possible. Therefore, the code is currently often merged into a companies repository without the needed investigations, because the effort is too high. If the code needs to be altered at a later time, the box of Pandora is opened: Developers do not have the necessary knowledge about the used frameworks. That missing knowledge is the main problem to react accordingly. If a manager knows about such issues, he can take actions to be ready for it. For instance, such actions can be to validate/check the contracted features of the desired system for training purposes of the developers for the used frameworks. Therefore, an easy way to control a minimum standard for taking over code developed by an outsourcing company is needed. Currently, the software cockpit cannot address this issue

since it is just based on issue data that is not available in this case. Furthermore, cockpits are not targeting at source code structure. Hence, software engineering faces the challenge to provide efficient overviews of code structure.

8.1.1. Contribution

In order to provide efficient overviews for code structure, we contribute with a new kind of cockpits in this chapter. We evaluate the feasibility and how Hypermodelling can be used to build a cockpit for parts of the code study we provided in the prior chapter. We present insights about the effort to build such a cockpit and we show how we derive this cockpit systematically. Additionally, we present the cockpit and reporting solutions that we built in videos.[1] With our cockpit it is possible to investigate the dependency on frameworks on various abstraction levels. As addition to this cockpit, we present further reporting scenarios and demonstrate advanced reporting possibilities. Out of the whole effort to build this specific cockpit and the other reports, we can conclude that Hypermodelling allows it to build cockpits for code structure in a reasonable time frame. Altogether, the contributions of this chapter are as follows:

- A schematic arrangement for a cockpit to investigate software variance

  We provide a schematic view over a cockpit to investigate software variance and discuss the indicators that should be shown therein.
- Implementation of the cockpit and effort evaluation

  We implement a concrete instance of the cockpit and evaluate the needed effort
- Advanced reports and their application scenario descriptions

  We provide and discuss further reporting scenarios and implementations for those. In special, we show first reports to the following topics: Cohesion analysis, mining developer knowledge, developer relations and developers Style Issue correlation.

All in all, the contributions of this chapter demonstrate the suitability to create code structure cockpits with Hypermodelling with reasonable effort. Furthermore, we depict concrete scenarios to apply the cockpits. Future researchers can now investigate our proposed scenarios further or have sufficient examples to derive their own applications with cockpits.

8.1.2. Reading Guide

First we discuss a concrete cockpit about software variance and evaluate its realization. This following, we focus on smaller reports about cohesion analysis, mining developer knowledge, developer relations and developer-Style Issue correlations. Finally, we draw conclusions and give an outlook to future work.

8.2. Software Variance Cockpit

First, we explain the variance of software with an UML class diagram. Then we go into detail and discuss main indicators to investigate dependencies based on variance indicators. Succeeding, we present a schematic view how a cockpit may look like. Lastly, we evaluate the effort to create the cockpit.

8.2.1. Recap of Software Variance

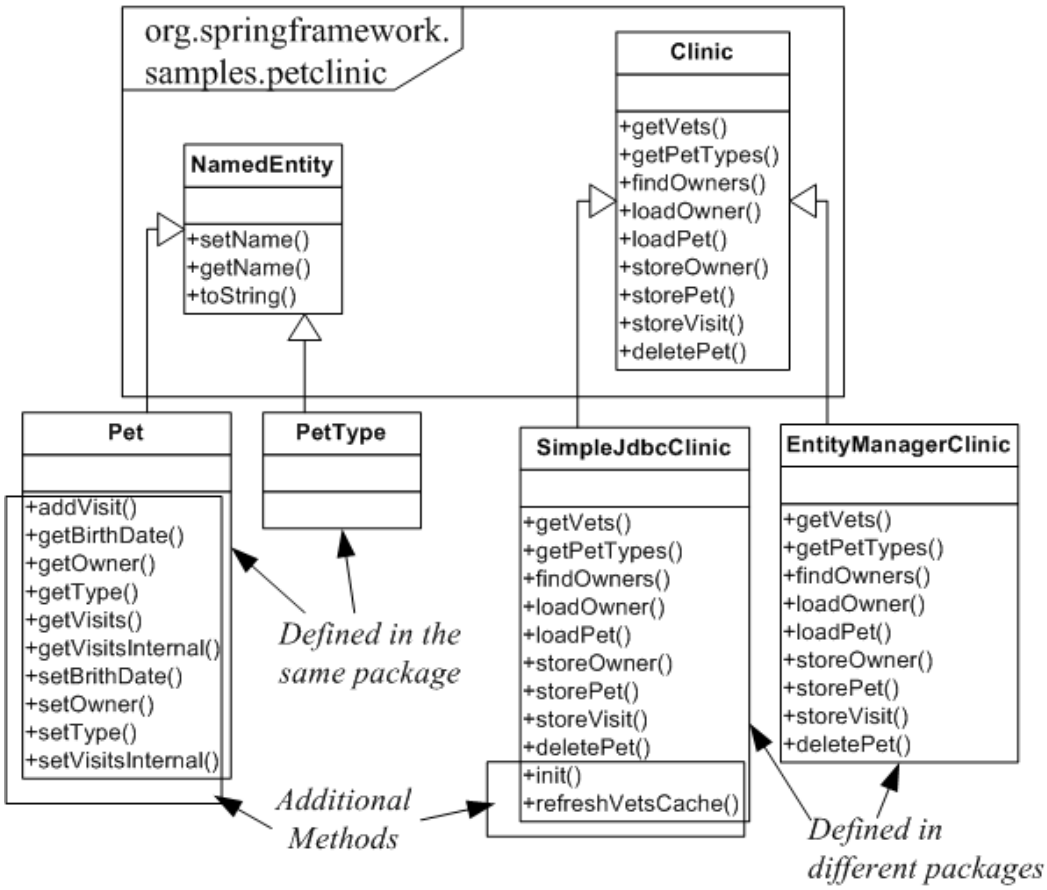
Different aspects of software variance and viewpoints about the dependency are discussed in the prior chapter. In the following, we also briefly present relevant facts that reflect properties of a software system that we use in the current chapter as indicators. We describe the variance and

[1] http://www.youtube.com/watch?v=aQ80JwmBuSQ - 7.11.2012
http://www.youtube.com/watch?v=6SSqpYRz_0w - 7.11.2012

indicators that are raised later, again. In order to reveal the diverse variance viewpoints, we explain software variance in the context of inheritance.

The class diagram in Figure 8.1 shows two types (NamedEntity class and the Clinic interface) that get inherited by classes and are defined in the same package. We call the types that are implemented or extended supertypes and their children inheritors. The inheritors occur in different packages for the interface and in the same package for the class. Methods get overridden and additional methods are contributed in their child types. Through overriding and addition, the defined standard by the supertypes get varied in the children by different means. In the following, we describe diverse indicators that can be computed out of this diagram.

Figure 8.1. Variance explained with an UML class diagram



Total inheritors

Investigations in the prior chapter described that the dependency on parent classes is problematic in cases of a software update. Therefore, one important indicator of software variance and the dependency on libraries is the total amount of inherited types. The more types are inherited the more is the software coupled to the super classes or interfaces. Typically, the types of libraries that are inherited are structured in a package hierarchy. This makes it possible to investigate the dependency on a library just by investigating the dependency on a package hierarchy. Therefore, the total amount of inherited types of a package depicts one degree of dependency on the package. If this package belongs to a library the package dependency is a library dependency. When more types get inherited of a package the dependency is higher. An example that can be depicted from Figure 8.1 is, that the total amount of inheritors of the NamedEnity class is two and the same goes for the clinic class. In case we refer to the total variance of the org.springframework.samples.petclinic package, the total amount is four. If we use the child package as discriminator, we can compute the amount for that perspective, too. For instance Pet and PetType is defined in the same package and the total amount

of inheritors that form the org.springframework. samples.petclinic package is two. Due to the fact that both other types (SimpleJdbcClinic and EntityManagerClinic) are defined in different packages the amount would be one for each package. We see this way, the amount of total inheritors and the variance differs from the viewpoint of the package wherein the children are defined. This is important, because the packages that are depending on types of another package can be depicted by numbers, reflecting the viewpoint. Hence, project managers can investigate the amount/degree of dependency from a package to the supertypes of another package.

Distinct inheritors

Similarly to measuring of total inheritance, but from a different perspective, is the amount of distinct inherited types. On the contrary to the total inherited types this number reflects the diversity of the inherited types. Thus, it answers the question: How many different types are inherited. If plenty of different types serve as parents the number grows. So, taken Figure 8.1 as example, the distinct inheritance of the org.springframework.samples.petclinic package is two, since only two different types are inherited. For the Pet and PetType package it is one what is likewise the same for the two other types in their own package. There, one type gets varied for each package.

Implementation ratio

However, when we want to have an indicator that reflects the “intensity” of used types we need another perspective. Thus, we introduce the indicator implementation ratio. With it, we enable a more concrete comparison of distinct supertypes to the amount of subclasses: The indicator enables to compare the amount of distinct types with the total inherited types in one number. Since classes or interfaces that implement or extend other types add commonly new functionality to the existing ones, also the original types get varied. Therefore, we see figures of implementation ratio as one indicator to measure the variance and dependency of supertypes. For all the shown types in Figure 8.1 the implementation indicator of the org.springframework.samples.petclinic package for all packages is computed as follows: The amount of inheritors of the package is four. The distinct implemented supertypes are two. So we can compute 4/2=2 and see that the indicator is two. Clearly, this can be, like all the other indicators before, computed for the diverse packages.

The measure of the implementation ratio is useful to depict a standard variation. A high or low value is a first indicator how intense supertypes are varied. When the indicator has at a high value, we can conclude that the defined standard of the supertype gets aligned a lot. This means that the same types are often implemented or extended. Every time a type is used, it gets adapted to the specific application needs. This way, developers have a starting point for further investigations to determine the types, responsible for high variance. This is especially useful, if varied supertypes are updated. If supertypes with a high ratio get updated, many children are dependent on them. Thus, it is recommendable to investigate how the children adjust the supertype to keep its future version compatible to the inheritors. Furthermore, framework manufacturers can use the ratio indicator to determine which types are mostly adapted. With that information, they can investigate how developers vary the types. It is possible to depict if there is a common use or functionality in the extending types. If so, this functionality can be encapsulated into a new version of the supertype.

Drill down to Top Inherited Types

When a project is investigated the desire to drill down to concrete types raises regularly. Investigators ask which specific types are inherited and how often. Therefore, this kind of drilldown is interesting for investigations. First an investigator focuses an abstract level of a package and then investigates the concrete inherited types. Therefore, a drill down of inheritance variance is to look into the specific types that get inherited. So when a parent package, like the org.springframework.samples.petclinic is correlated with an implementing package, like the one containing the Pet and PetType we can compute which types get most inherited. Thus, it is just the total amount of inheritors for a specific package. The most inherited types are then the top inherited types in that package.

Drill down to Top Overwritten methods

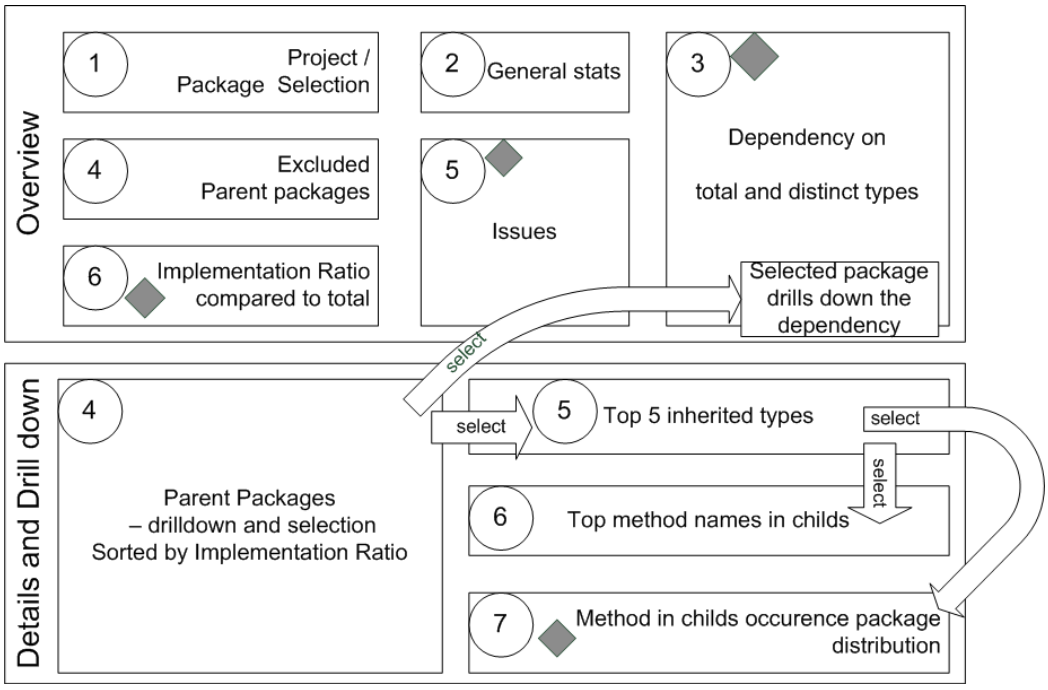
It is often interesting to determine the most common method names in inheritors. This knowledge is important to see first, if there are similarities and second to have the possibility to compare it with the original methods in the supertype. Through the inspection of the amount of used method names in children, conclusions can be drawn how often a method gets overridden. This way, estimations can be done about how much a change in the supertype method may affect the children.

The more types inherit a type the more it is important in the implementation. For instance, we can see that, logically, all methods defined in the Clinic superinterface get implemented in clients. For the Pet supertype we can see that the original defined methods do not get overridden in children and thus do not vary. However, we can depict which other methods are added in children. This gets quite handy, when looking at a larger number of children, to answer the question: Do many children override the same methods or add methods with the same name?

8.2.2. A Schematic Cockpit proposal

Out of the former described indicators, we formulate the arrangement of different elements for analysts. All the different variance stats are presented in Figure 8.2. We recommend the views with the grey diamonds to be graphical to provide an easier overview that is more eyes catching than pure tables. In general, we lean against principles of information dashboard design [86], as far as they can be applied for our special case.

Figure 8.2. A Schematic Variance Exploration Cockpit



The overview part

The overview part shows an abstract overview and graphical visualization of the currently investigated package or project. The following numbers refer to the circled numbers in Figure 8.2 and Figure 8.3 that is following in the section called “Cockpit usage”. We refer here already to Figure 8.3, because Figure 8.3 shows an actual implementation of Figure 8.2.

1. The package selection enables the user to select packages that he wants to explore. He can select packages from the hierarchy to have a known navigation. This selection serves the purpose to define the base that is investigated for variance. All other views show facts about the package that is selected here. This selection is the start of the exploration of a projects package hierarchy that slices/discriminates all the other elements of the report.

2. The general stats show a statistic about the current selected package. How many classes are in there, how many interfaces and how many method calls are occurring in a package. The main reason to show these stats is to give the investigator an easy overview to estimate the "size" of the investigated package.

3. Here we see a graphical representation about the dependency on different and total types of the current investigated package. There, the abstract level of packages is shown and how many types of a package are inherited. We propose to use a pie or spider-net chart to visualize the total and distinct dependency graphically.

4. Here the user can specify excluded packages. The reason that we see a necessity to offer this view lies in our investigations in the previous chapter. There, we recognized that standard packages are used often and the use density of one package covers the usage of others. Therefore, users can just exclude dominant packages that contain parents to gain a clearer view. All the different elements in the cockpit are filtered to exclude those packages.

5. In order to show also other indicators, we recommend showing them additionally to the variance issues that are related to the currently selected package. Such issues can be successful or unsuccessful unit tests of the classes in the package or indicators like code smells. Through the showing of the issues at the same time with the dependency on parent packages, managers can see correlations between issues and used parent packages at the same time. For example, when a certain package is used a certain Style Issue seems to be occurring with a high amount. When this thesis is formulated it can be the beginning of an in-detail analysis.

6. The implementation ratio of the current selected package (view 1) is shown with a graphical visualization. Through this, a direct excelling rate can be determined easily. So when the rate is low or high, is can be instantly perceived to decide if another package should be investigated.

The drill-down part

This part shows the details about the inheritance and presents specific types, packages and methods to the user. It is the part where the user can get into details.

7. We show the packages from where the supertypes origin. The user can drill down into the package hierarchy of the supertypes. When this drill down is done the depending measures shown, are updated. For instance the dependency on the distinct types (view 3) is now computed for the selected packages.

8. When a package is clicked (in view 7), also view 8 is adjusted. In this view we show the top parent types that are inherited from this package. This way, users can select a project and then explore which packages are the most original and then drill down into the parent packages hierarchy and into the concrete parent types.

9. Like in the case of packages and the drill down to types, this is also a drill down. In view 8 a type can be clicked and then the top method names in children get listed in this view. Thereby, the total occurrence of this method name is shown. This way, it is easily to see which method names are very common in the selected package (1).

10. However, in view 9 users have the possibility to see the method occurrence in the selected package. The visualization in 10 gives the users the opportunity to see the amount of methods of the currently selected supertype (selected in 8) in other packages. Users can easily see which other packages could be interesting for investigations about the same type. Because if other packages have a high amount of methods for the children of that type, the other packages share a huge variance, too.

8.2.3. Application Evaluation

In order to evaluate our approach to provide a dashboard for the inspection of software variance, we evaluate the feasibility through a demo application. We answer the question how much effort is needed to realize a cockpit with Hypermodelling. Furthermore, we overcome the necessity to create queries and provide a flexible real world solution and not to stick to a schematic ivory tower.

Our cockpit can serve additionally for further investigations of software variance. In the following, we first line out the necessary effort to build the cockpit based on the Hypermodelling approach. Then, we give a brief overview about the implementation that reflects the schematic overview in Figure 8.2.

Realization evaluation

For the realization, we used the following approach: First, a description, similar to the one in Section 8.2.2 was made. We used a schematic picture similar to Figure 8.2 and a detailed explanation what different elements should represent to provide a holistic goal. Then, we gave this description, plus a Data Warehouse that was already filled with source code and the associated queries for the diverse elements to a reporting specialist with the task to create the cockpit. The specialist was experienced with the used reporting technology and worked in that field for five years. Within 10 hours the specialist realized a first version of the cockpit that contained fixed queries. Fixed means that the queries could not be altered and the view was not dynamic and showed a report without selectable elements. Out of this first version, we adjusted and specified in detail requirements. Those requirements how the cockpit should look and behave were exactly and well formulated and given to the specialist. This iteration was done 4 times and every time new requirements about behavior were added. For instance, first we defined which elements should be parameterizable and change dynamically. Then, we defined different selectable elements and so on. In the last iteration, the loading time of the cockpit came in focus. It took five minutes with automatically generated queries on an Windows 7 computer with an Intel Core 2 duo with 4 GB RAM. In order to fix this performance issue the expert rewrote two queries of the cockpit manually, what reduced the loading of the cockpit down to acceptable 11 seconds. In order to achieve this performance increase only a small optimization to exclude empty values was applied. Therefore, even further tuning is possible. This way, we recognized that performance tunings are no real issues with a query based approach.

All together, including the first fixed version the specialist needed about 80 hours to create the demo cockpit. Additionally, it needs to be mentioned that the specialist is normally focused on more simple and business related reports. He needed to look up a lot of information for the complexity of the cockpit and its possibilities within that time. Additionally, the cockpit was built after his normal working hours. Thus, we can estimate that with a main working task and the complete knowledge about the used reporting technology the cockpit would even have been built much faster. Therefore, we see indications that cockpits for the internal code structure can be created within reasonable time and effort.

Cockpit usage

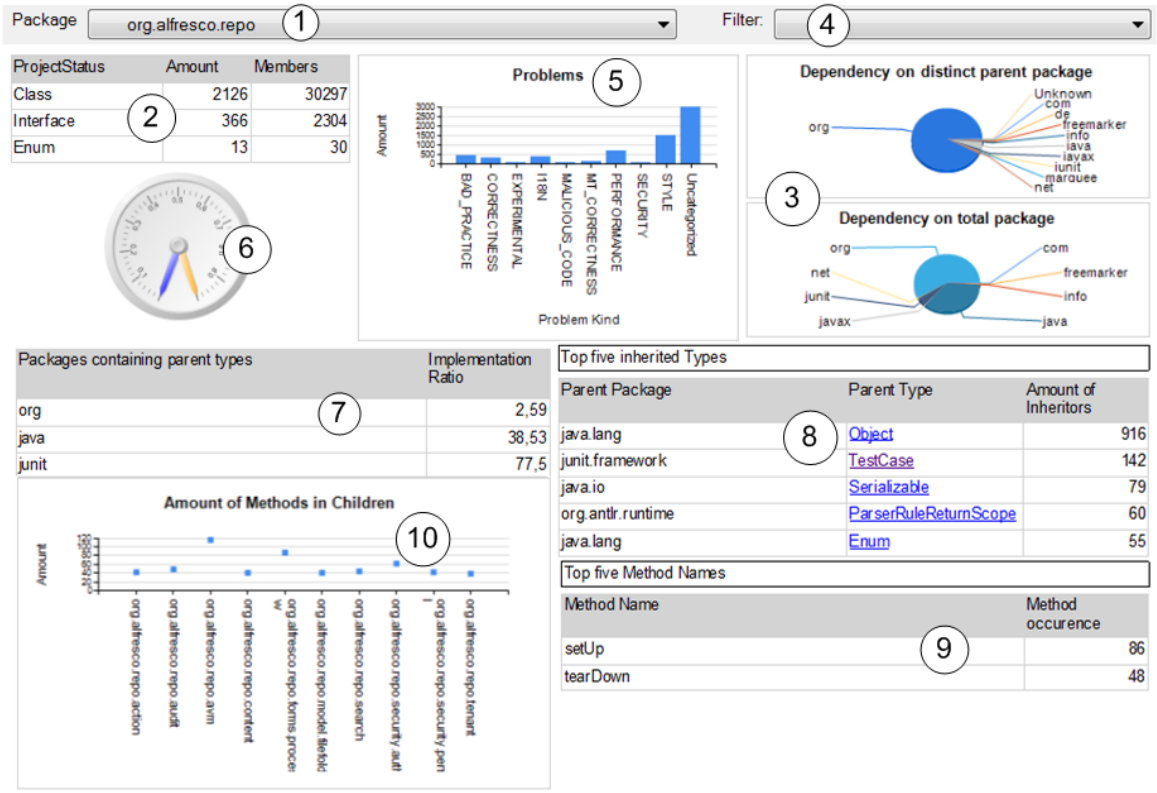
We show the cockpit in Figure 8.3. The numbering of the various elements corresponds to the schematic view in Figure 8.1. All the various parameters that are defined in the report can also be selected through URL parameters, when accessing the report. This has the advantage that when something of interest is discovered, the current looking report can be just linked and send to another person. Then, the other person has completely the same view and can go into further investigations. Also, when an interesting issue is revealed various export formats (excel, pdf) give the possibility to use the export for presentations or archival reasons.

We see that the user can select a package for which the various stats are displayed (1). This is done via a parameter that can as described before, also be specified via the URL. When the parameter is altered the whole report changes and shows the new specified package. The project stats (2) refer to the current package. Currently, it shows the amount of types (classes and interfaces) and their members (methods and fields). The different dependency types (total and distinct) are visualized via pie charts to give a quick graphical insight (3). In order to support estimations if there may be correlations of Style Issues and the dependency of packages, the problems bars (5) show the amount of detected issues. The speed indicator (6) allows gaining a quick overview of the variance of the selected package to the whole investigated project. All together, the whole charts can be sliced by excluding packages that contain parent types (4) to avoid a dominance of certain packages.

In (7) we offer a drill down into the parent packages. Depending on this selection the top parent types get shown in 8. Also, the dependency within 3 is adjusted. So, if the user starts to explore a specific package, containing parent types, he can explore this kind of hierarchy.

When a specific top inherited type is selected the most common method names are presented in 9. Additionally, the distribution chart in 10 shows the occurrence of methods in children in all the packages. Through this chart the user can quickly gain an insight if the current package that is investigated alters the selected type more than others.

Figure 8.3. Cockpit in Action



8.3. Advanced Reports

Before, we discussed inheritance dependency investigations within a project. Right now, we show additional possibilities to investigate other metrics with the Hypermodelling. All of those reports can be extended into solutions like the prior explained variance cockpit. However, the main focus is to show the possibilities what for Hypermodelling can be used for. Therefore, we scratch the scenarios only on the surface. The main purpose is to give indications in which direction Hypermodelling can be used so that concrete scenarios can be developed in the future.

First, we describe how Hypermodelling can be used to investigate cohesion metrics for methods. Afterwards, we follow up with some reports about developers and code.

8.3.1. Method Cohesion

It is described that the modules of a software system should have a single responsibility and not be dependent on other modules. They should be independent and serve as a ready to use unit. Within this context we introduce the indicator of cohesion that expresses to which degree the elements of a module belong together. In the following, we look at cohesion show different cohesion indicators that we computed by queries.

We note a relation of cohesion and the prior chapter about software analysis. There, we described reports about dependencies of packages that served as inspiration for the cohesion dashboard.

Our definition of cohesion is similar to literature [265]. Right here, we define cohesion to be the indicator that expresses the internal coupling of a module. The computation of those is for us the ratio of internal bindings (self calls) within the module compared to the bindings to other modules (external bindings, nonself calls). The scope of the module defines what is internal and external. For instance, the scope of a class contains all methods. The scope of the method contains only the internals of a method, but when we refer to a method to be a member of a class other modules can be considered as internal, depending on the vantage point. This way, different ways to compute the cohesion indicator are possible.

One may raise the argument that cohesion [265] differs from couplings [205] and there is just a correlation between loose coupling and high module cohesion, but is not the same at all. We agree to this argument, but induce that for our demo purposes of Hypermodelling it is rather important what can be done with the approach and not to argue about cohesion and coupling. Similar implementations to the one we show it in the following, can deal with such specifics in the future.

Therefore, depending on which kind of module we regard in a programming language, may it be a method, a package or a class, the indicator of cohesion is computed differently. Right here, we focus on the cohesion of methods. However, the cohesion of methods can also be expressed and aggregated in different ways. For instance, is a method depending on methods of the same class or other ones? Is a method depending on methods of the same package? Do we compute method cohesion at the package level and regard all methods together or do we regard single methods and aggregate their values together to the type and package level?

We see that the right and perfect way to compute cohesion is depending on the perspective and how it is defined as quality indicator for a project. Therefore, we show in the following just one example how Hypermodelling can be used to create a report about one kind of possible cohesion computation. Clearly, other computations about cohesion can be implemented likewise. In the following, we discuss and exemplary report about cohesion and describe the various shown indicators as example for other scenarios.

In Figure 8.4 a report and drill downs about cohesion indicators is shown. On the left side we visualize to key cohesion indicators. The "Nonself Calls/Type self Calls" indicator expresses if methods invokes only methods within the same type. Therefore, this indicator expresses the method cohesion within a type. The "External Calls/Internal Calls" expresses how many methods within the same package are invoked and how many of foreign packages get used. Both indicators result in numbers, which we classified in boundaries of good, medium and bad (red, yellow and green colors). The classification has been done by computing median ranges. Anyway, it is rather important that such boundary classifications can be done and not which specific kind of classification is applied. Right before the cohesion indicators, we see statistics about the method calls that are occurring within a project.

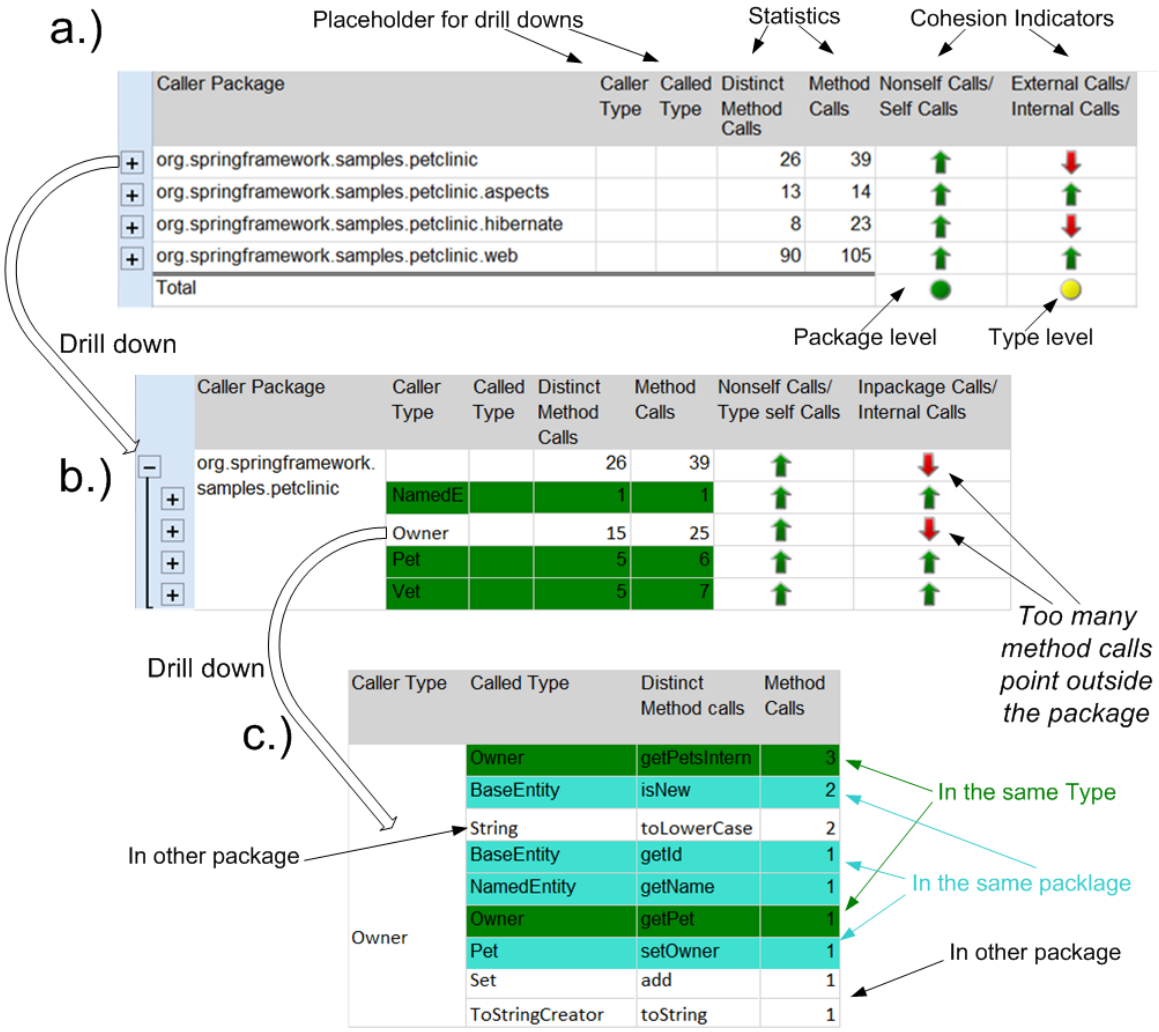
We note that the shown report is stripped down to the essential parts, needed for better comprehension. Therefore, if one recomputes numbers, he may end up in different totals.

Through this view with cohesion indicators, we recognize rapidly that the cohesion at the package level for methods seems to be in a "green and good" range. At the type level this seems not to be the case. In oder to reveal reasons for those bad indicators, a drill down just by clicking the specific package leads to further insights what we show in part b.) of the figure.

In b.) we see another supporting mean for this investigation. Types in the package that call mainly themselves get colored green. We see that the Owner type in the package is not colored green and calls 25 methods in total, from which 15 are going to different other methods. Also, the Owner type has a bad red indicator. Again, by clicking the type the insight drill down information of the type gets shown, what we show in c.).

The insights about the calls of a specific type in c.) define which other types are called. The colors help again reveal the destination of the calls. Ones go to methods of the type itself, other ones directly into the same package and other ones get outside the package. Here, we see that the Set collection (add) and string methods (toString) are used. Those are just helper utilities. Therefore, we can assume that even with the bad indicator the cohesion regarding packages is still acceptable.

Figure 8.4. Reporting Cohesion



We show excerpts of a report about cohesion. High level visualized and aggregated indicators at the package level give instantaneous insights about the cohesion metric (a). Starting form this, drill downs (b,c) into the specifics can be done and reveal why a high level rating is bad. We show that the report supports two levels of drill downs (b,c). The user is supported by coloring mechanisms that allow a fast overview which insights are responsible for the bad rating.

We note that in the original report more methods got shown, which were originally responsible for the bad rating. Here we gave the false information and accused the utility methods responsible to keep the example consistent, since details about specific investigated applications are not relevant for this example.

All together, we presented an exemplary report to investigate one kind of cohesion. We described how drill downs in a report can be done and that Hypermodelling can be used for this case. Now, concrete reports that respecting specifics can be created in a similar manner.

8.3.2. Mining Developer knowledge

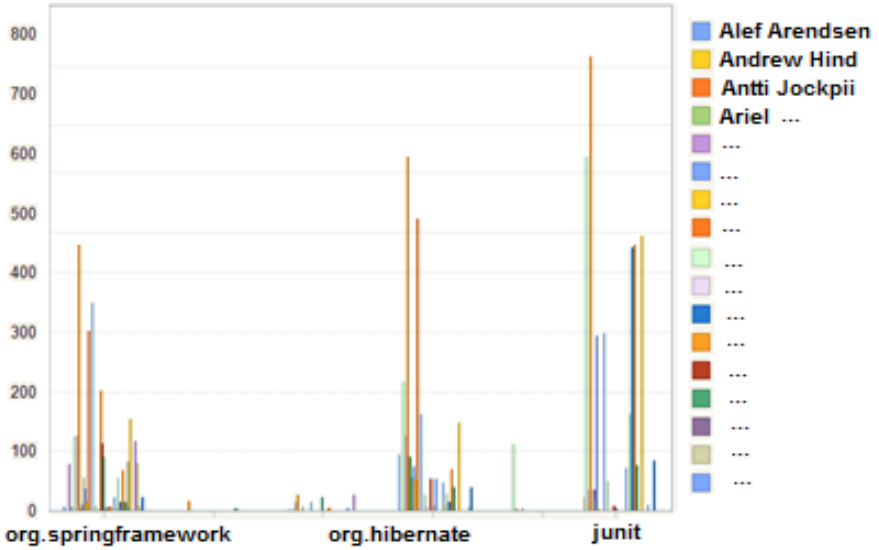
Before, we focused on internal code metrics. Right now, we use associated developer information to gain insights about a programmers work. Thereby, we refer to the prior chapter about software analysis, where we mentioned that project managers may investigate how dependent their project is on specific technologies of libraries (see Section 7.6.4). According to the use case to determine the required skills of developers for a project, it is similarly important to know the library usage experience of developers.

In order to gain first insights about developer's knowledge, we can use the information which source code was written by which developer. When a developer encodes logic that utilizes a library, he obviously knows how to use the functionality of this library. When he uses more parts of the library and does that often, we can assume that the knowledge about this library is higher. Clearly, this is not perfect, but even though: Still, it is an indicator for a developers skills.

Therefore, we use the information about code authors and their called libraries as indicator about their knowledge and present the result in Figure 8.5. There, we see the distribution of library usage and the different developers. The different top-level packages of libraries on the x-axes show the libraries. The y-axes shows the amount of method calls to this package. The different colors indicate the different developers. We see that most developers seem to use and have knowledge about the org.apache.common package structure. It is also interesting that knowledge about concurrency seems to be only in a few developers hands. Likewise, other interpretations can be made. Also, it is clearly possible to filter the graphic for certain developers.

However, one may rise that we just used method calls and no other means like inheritance. Therefore, we induce that this is only an exemplary and first idea to use written code of developers to infer from that about their potential knowledge. Further reports and cockpits can be created to reveal more detailed information. Right here, our report serves as just as indicator that similar scenarios are possible.

Figure 8.5. Developers Calls to Packages



We show a chart about mined developer knowledge out of source code. Developers are shown in different colors. The amount of their method calls is arranged on the y axes in the graphic. The destination of their method calls is shown in the x axis. The amount of method calls to a certain package can be used to do inferences about the potential knowledge of a developer about this package: Potentially developers know more about a package when they use it frequently.

8.3.3. Developer Relations

Before, we described the approach to mine developer knowledge from the codebase. However, in a development project not only code of external libraries gets used, also developers manufacture code that is used by other developers and documentation has to be made.

A scenario can be that one developer has to write an API documentation about his source code. Clearly, the right people to control or advance the API documentation are the ones that use the code of the developer. Therefore, we show the approach to use Hypermodelling to reveal which developers code is invoked by which other developers.

We show a crosstable how many methods of another developer are consumed in Figure 8.6 . There, we arrange developers horizontally and vertically. We see that Andy Hind uses plenty of his own methods. Furthermore, Britt Park makes heavy usage of Andy's code. They are followed by Derek Hulley and David Caruana. Therefore, if Andy wants to document his code, Derek, Britt and David should read and control his API documentation.

As a matter of fact, we look at the case of Brian Remmington. Brian consumes 31 methods of Andy's code. Lets consider that also Brian should read Andy's API documentation for the cases that he already used. In order to reveal the methods that Brian already used, we can drill into detail and reveal the methods that Brian consumed. We show this possibility in Figure 8.7. There, we see that we get even further information and which methods Brian invokes of Andy's API. A use case to utilize this information can be to show Brian his own methods and how he invoked the code of Andy while he reads the documentation.

Figure 8.6. Developer Relations

Developer		Andy Hind	Antti Jokipii	Ariel Backenroth	Arjen Poutsma
Alef Arendsen	-	-	-	-	-
Andy Hind	-	6882	-	-	-
Antti Jokipii	-	-	2	-	-
Ariel Backenroth	-	1	-	54	-
Arjen Poutsma	-	-	-	-	4
Arseny Kovalchuk	-	27	-	-	-
arsenyko	-	6	-	-	-
Brian	-	97	-	4	-
Brian Remmington	-	31	-	3	-
Britt Park	-	1484	-	11	-
CACEIS	-	6	-	-	-
David Caruana	-	317	-	-	-
David Ward	-	-	-	-	-
davidc	-	48	-	-	-
Derek Hulley	-	735	2	-	-

Developers are programming source code against method definitions of other developers. We show that this social information can be retrieved and arranged in a crossable with Hypermodelling. We see that the amount of method calls that are made to code of another developer.

Figure 8.7. Developer Relations - Drill-down

Filter	Selection
Called Developer	Andy Hind
Developer	Brian Remmington

Go Back one step

I Method	getChildRef	getNodeRef	getParentRef	getQName
buildModel			1	
checkFolder	1			
createFolderNode		1		
getTemplateNode		1		
makeNode	2			
moveNode	1			1
onSetUp		1		
onSetUpInTransaction		2		
parseXMLDocuments	2			
testCreatesFoldersForTemplatedLocation				1
testDelete	3	1		

A drill down for Brian and Andy show which methods of Brian invoke which methods of Andy and how often. The callers get vertically arranged and the callees, horizontally. We note, specific queries can also get more detailed if desired and show the types of the packages or further information.

However, this scenario shows just an example. Clearly, the slicing and abstraction capabilities of Hypermodelling are at hand to discriminate such results for specific packages and similar. Additionally, other parts of the code structure can be used to reveal similar information. All together, we see that some kind of social information already exists within the source code what can be used to help developers working together.

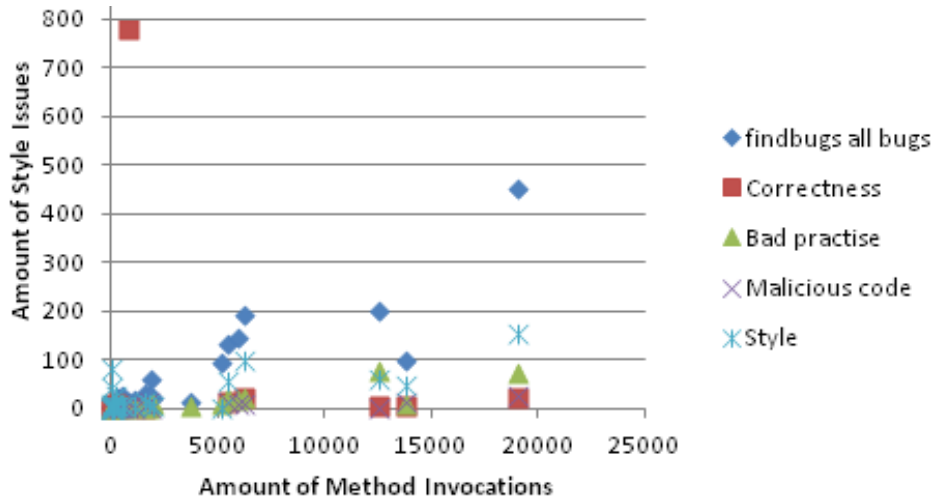
8.3.4. Developers and Style Issues

Another developer centric investigation can be made by utilizing that Style Issues as well as developers are associated with source code. Just imagine a company wants to lie down a minimum set of requirements for improving code quality by irradiating some Style Issues. But which developer does do which Style Issues regularly? - It would be time-consuming to communicate all Style Issues to all developers. It would make much more sense and spare broadcasting information to communicate the corresponding Style Issues to the developers who do them regularly. In the following, we show a first insight how Hypermodelling can be used within such scenarios.

In order to reveal the distribution of Style Issues and developers, it is necessary to consider that developers write different amounts of source code. When we look at someone that has written just a few 100s lines of code, he will have less impact to all occurring Style Issues, within a project, compared to someone that writes a few 1000nd. Additionally, different categories of Style Issues occur. We put all of this information together in Figure 8.8 and show all Style Issue categories, detected by the findbugs. The x-axis are the number of method calls by a developer and the y-axis are the number of detected problems. Every point in the coordinate system calibrates the amount of written method calls of a developer to the amount of Style Issues in the developers' source code. Therefore, we have to imagine that every point in the coordinate system is associated with a developer.

We see that many developers have written code between zero and 2500 method calls. We also recognize that as more lines of code get written, the amount of all Style Issues together rises (findbugs all bugs). Interestingly, we see that the different Style Issue categories are arranged differently for the ascending lines of code. At around 20.000 method calls the Style Issues occur the most; at around 13.000 the bad practices are preceding. Out of this, we see indications that the kind of Style Issues can vary for different developers.

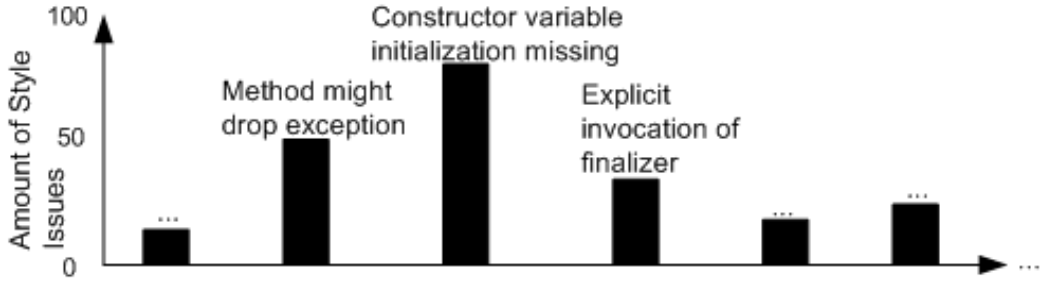
Figure 8.8. Style Issues according to Developers Lines of Code



Different kinds of Style Issues and the sum of those (findbugs all bugs) are shown in the legend. The amount of Style Issues is associated with developers. Their encoded amount of method invocations serves as x-axes indicator where the points for the amount of Style Issues are placed.

Out of the prior overview, we depict the developer with the most method calls (nearly 20.000) and go into the details of the occurring issues for him. Thus, another query determines the specific issues that were found for this author and reveals the result in Figure 8.9. The bars indicate how often a specific problem occurs. Such a problem can be for instance a non initialization of a field in a constructor or similar.

Figure 8.9. Style Issue Kinds of a specific Developer



The bars show the amount for different Style Issues kinds (x-axes) that occur in a developer's code. We cut the naming of the issues out, since their name is not discussed here. The y-axes measures how often the Style Issues occur. This helps to determine which Style Issues a developer violates most..

With this information, we contact the author of the code and discuss the different issue kinds with him. In this discussion, we recognize that the constructor variable initialization is missing, because the inversion of control design pattern is consequently used. Therefore it is possible to neglect this issue aside and focus on the other issues that are relevant for the specific project.

We approached the developer in real and did this discussion. Thereby, we also got the feedback that the flexible adjustment of Hypermodelling is especially important in projects, because different Style Issues matter differently to companies.

However, this scenario is a only first application to use Hypermodelling and Style Issues and developer information together in a report. Therefore, the main benefit in such a scenario is that the Style Issues can be specified in a query or excluded by demand. The developers see that as crucial and very important feature.

All in all, associating developers with malicious code is a difficult and challenging task that should not done in such a small discussion. We recognized that flexibility of the tool and to use accepted "bad issues" is crucial. Right here, we showed that it is generally possible. Now, future researchers can reveal which issues and processes are important to be looked at to implement our approach in specific scenarios.

8.4. Summary and Conclusions

We described the business case about the difficulties to investigate source code for dependencies on frameworks. A brief description of the Hypermodelling approach was given. The diverse indicators of software variance that express inheritance dependencies were described. We presented a schematic overview integrating these indicators and their relation together. We evaluated the feasibility through an implementation. Thereby, we figured out that this kind of cockpits can be built with reasonable effort and time range. Now, audits about the dependency and standard variation of software can be done easily.

Supplemental, we described how Hypermodelling can be used to create a report about cohesion. This following, we presented the idea to mine indications about a developers knowledge from a code base. Similarly, we gave insights how a code base can be used to uncover which developers use source code of other developers. Lastly, we showed the possibility to create reports about the most often done Style Issues by developers and their distribution in the code base.

Altogether, our investigations find that creating reports with Hypermodelling results in reasonable effort and there are a lot of application scenarios. Even the creation of complex cockpits is justifiable in time as well in effort. This enables further research about different cockpits for various roles about the internal structure of code. Researchers now have technology at hand that enables them to create structure reports. They can construct advanced usage scenarios with the shown possibilities of Hypermodelling or push our shown use cases forward.

However, in this chapter we presented a cockpit that describes the dependency mainly on inheritance mechanisms of object oriented programming. There are other mechanisms available that get used to couple artifacts in a program together. Therefore, we see the necessity to investigate which kind of other dependency mechanisms exist and integrate them in a future version of the cockpit.

Furthermore, our cockpit shows that it is generally possible to create a cockpit, visualizing relations in the code structure through “traditional” management charts. Further research should focus on studies about the application of cockpits in applied scenarios.

However, all the different shown capabilities demonstrated different ways how and for which advanced cases Hypermodelling can be used. The main purpose of those different scenarios was to give researchers impressions in which directions the technology can be applied. Additionally, the shown scenarios correspond to first work from which we got already interested developer feedback. Out of this reason, we saw the need to present these first results about advanced usage scenarios.

Another trail of research is to create further cockpits about the code structure to enable advanced project management support. This way, we see further research to identify further cockpits within diverse areas of development. Hereby, we see especially our advanced reporting scenarios of interest. For instance, developer or management centric dashboard about developer knowledge can also be developed. This way, we are certain that code cockpits can enable advanced software project management support in the future.

In summary, this chapter closes the evaluation of Hypermodelling's suitability for reporting and code investigations based on queries. All over, we saw that Hypermodelling is suitable to advance current research within these areas. All till here shown applications, are pure read only investigations of a code base. In the next chapter, we use those query investigation capabilities and apply those to predict and plan the migration of a program.

9. Hypermodelling the Future

"God, I love science fiction."
John Crichton. Farscape: Revenging Angel, Season 3, Episode 16. 2001

Abstract. Imagine a developer, who modularizes a traditional object-oriented program with annotations. Can we plan the migration and the adaption to the new mechanisms? Can we measure progress of our migration plan? We present a method that can be used to plan the realization of concerns with figures like it is done in enterprise planning scenarios. We evaluate our approach with an excerpt of a demo application that is upgraded against a new framework version. Our contribution enables further research about the planning of future program versions. Furthermore, we propose investigations about the necessary management support and the needed indicators to measure the progress of a concern encapsulation association movement.

9.1. Introduction

In the prior chapters, we introduced the Hypermodelling approach and showed scenarios how it can be used to do reporting about the source code structure. In this chapter we extend the idea of pure "read-based" reporting by focusing on the idea to "write and compare" indicators to plan the future. This extends the pure reporting based approach and shows a future use case of Hypermodelling.

Just imagine the current world. The Java community has done it again. A new mechanism to separate concerns has been introduced and is getting widely used to solve programming issues in a new fashion. Aspect oriented programming enables encapsulation of cross cutting concerns through pointcuts and Advices [141]. Additionally, widely applied frameworks offer ready-to-use cross-cutting concern functionality that can be weaved into programs by marking elements of a program with annotations [141]. This way, annotations allow encoding of issues directly in code, which have been carried out for years by means of configuration or inheritance. Thereby, using those annotations is quite similar to weaving in Aspect functionality within a program. In fact plenty of Aspects, like security or transactions, are today already realized with annotations. Therefore, we assume that learning how updates to annotations can be done better, will help to get first insights how to update to Aspects can be done in the future. Additionally, the approach of AOP is still controversial, therefore, we see the need to come up with a generic technique that is ready for the possible future of aspects and works also for today's problems. Out of this reason, we deal with the problems of updates from inheritance to annotations and present and present generic and adaptable method that can be transferred to other approaches.

Updated frameworks force programmers to adapt and respect the new structure of using annotations. Developers have to shift to this new paradigm to use annotations to refer to the logic defined in a new framework version. Conversion assistants help, but lots of custom code needs to be altered. Thus, different questions arise when upgrading a program: How can we plan systematically what and where to update? Can we see if we stick to our goals during the update process? How far are we?

Those questions can currently not be answered. Generally, a vision in software development is to model a program top-down. But, still, how concerns are encoded in a program cannot be planned. Currently, a project manager cannot plan the update for specific parts in the source code. A manager faces the challenge to delegate developer resources to parts of a program with pure experience. So, a manager needs a system that supports his decisions. For instance, such a system should help to estimate the effort to update the various packages in a program. This way, developers could be delegated to update the different packages.

In other areas, project managers plan the future based on reports. Figures like revenues are used as indicators to estimate their equivalent in the future. There, an economic plan is expressed as a figure in the future. The planned future figure can be compared with reports of the current figure. The comparison between the figures is then used to compute the distance of the desired future figure.

Such kind of functionality is commonly accomplished by planning with Data Warehouses [178]. We showed in the previous chapters that Data Warehouse technology can be used to compute figure for concern associations. In order to enable the planning of future figures, we introduce the method of planning indicators in a Data Warehouse to estimate future concern associations.

Therefore, the motivation of this chapter is to extend the current approaches in code analysis by introducing a future planning approach. This approach is widely used in Data Warehousing and its application within software engineering is new. Hence, the contribution of this chapter is as follows:

9.1.1. Contribution

We present the new idea of a concern association planning mechanism. We propose to compute figures for concern associations in a package before and after an update. These two figures can be compared to estimate the progress of an update process.

We define our future planning as method that contains the following features:

The computation of at least one indicator that describes at least one relation within source code, whereby:

- The indicator is measured before an update and used to specify the same or multiple other indicators for a future program version;
- The current and the future indicators are compared to determine the update progress of the program;
- Optimally, the specification of an aggregation of at least one indicator before and after the update.

Supplemental to the method, we provide a first evaluation to apply the method at a demo program that is migrated from inheritance to annotations. Managers can use this method to control the progress of updates and plan concern association at the package level. Now, it is possible to plan figures for annotations and object-oriented inheritance mechanisms in packages of a Java projects top-down.

9.1.2. Reading guide

First, we describe the approach how figures for concern associations can be computed. In order to evaluate the feasibility of our approach, we present an excerpt of a demo application that is updated and show the application. Succeeding, we address potential raised arguments against our approach and validate it in a discussion. Afterwards, we refer to related research and point out potential synergies. Finally, we draw conclusions and give an outlook on future work in this area.

9.2. Future Planning Approach

Here, we show how future concern association indicators can be computed on an abstract level. Therefore, we generalize the idea of an update to an annotation to a more generic view: A program is a set of vertices that are associated with each other. For instance, a class and its super class are two vertices that are connected. Another example are vertices and their relation is an annotation at a class. When a class is updated from inheritance, this changes the annotations associations between the vertices. First, the vertex of a class is connected with the superclass vertex and then it is connected with the annotation vertex.

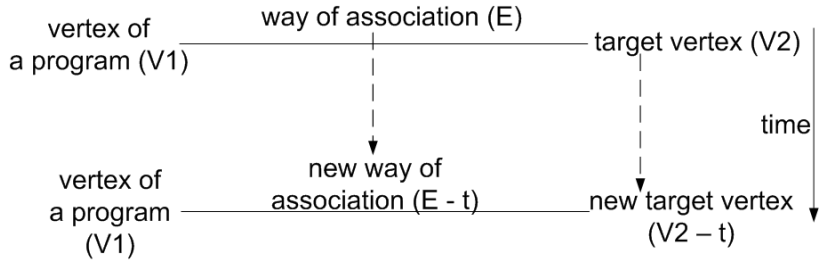
However, we present a general change of vertices and their associations in Figure 9.1. The main meaning of this figure is to show a vertex association that is altered on a generic level. We offer this abstraction to this generic level first, to enable the possibility to transfer our approach to other concern associations. This way, we show first the generic idea and then show afterwards a concrete movement in a program.

Figure 9.1 shows that the association of a programs vertex with another vertex changes over time. In the top region, a vertex (V1) is associated with another vertex (V2). This is the concern association before an update. After an update (bottom) V1 is associated with the vertex V2 – t. The node type

of V2 –t can be different to the node type of V2. Therefore, the edge (E - t) that associates V1 and V2 – t together can be a different one than the V1 – V2 association.

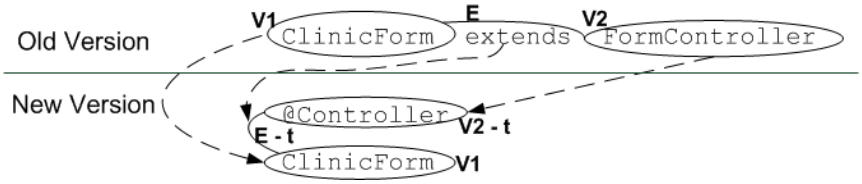
Sometimes programmers update a program and also alter the vertices (V1) to implement new logic. Here, we focus only on the case where the original node (V1) endures an update. We do this to concentrate on the case of a program update and assume that V1 endures most updates. This means that V1 stays V1 and is just associated with another vertex (V2 to V2 - t).

Figure 9.1. Generalized Association of Source code



We show a code example of Figure 9.1 in Figure 9.2. There, source code is updated. The vertices are rounded and the edges that connect them lines. A ClinicForm class extends a FormController. In a later version the FormControllers functionality is superseded by the @Controller annotation. This way the way of concern class associations changes from inheritance to annotations. Also the vertex of the controller annotation changes from a class type to an annotation type. The total number of concerns associated with a node can be expressed as distinct vertices connected by a specific vertex. Thus, the association figure count of FormConroller with ClinicForm is 1. In the updated version it is 0. Similarly, the association figure of ClinicForm with @Controller is 0 in the old version and one in the updated version. Generally, elements in programming are connected to multiple concerns. The association can be computed with an association figure.

Figure 9.2. Source code Evolution Example



The main idea is now to use the Hypermodelling approach to compute figures for concern associations in a package before an update. Then project managers plan the future concern association figures. The comparison between the current figures and the future figure expresses the progress of an update. This indicator comparison can be used by managers to measure the current state of an update.

Programmers update programs against new framework versions. Commonly, frameworks offer documentation on how concerns are accomplished and how the shift to a new version can be done.[1] Hence, we assume that a software project manager knows how to update a program to a new version. This means more specifically: A manager knows which vertices have to be updated. Thus, a manager can estimate the needed changes for the concern association of one vertex out of the upgrade instructions. For instance, a manager would know that the ClinicForm must be changed to annotations and he expresses the future figure out of the current one. If he can do it for one vertex, he can also leverage the knowledge to estimate future concern associations of multiple vertices that

[1]For instance, see a description for updates here: http://static.springsource.org/spring/docs/upgrade/spring3/pdf/spring-framework-upgrade.pdf - 7.11.2012

share the association. He can do the estimation for multiple vertices by multiplying the estimation of one vertex update with the amount of vertices that share the same concern association, e.g. all vertices that are associated with the FormController vertex. This can be performed, because the same associations need to be updated the same way. Additionally, this procedure can be abstracted and aggregated on to the package level, i.e. computing the number of vertices of a type in a package, which inherits from types that are defined in another package. Hence, a manager can also use a figure like the inheritance relation of one package to another to estimate the future relations.

Our method proposes to leverage this knowledge of the manager, to compute the indicators before an update and to specify the desired indicators after the update. The continuous comparison of indicators before the update, measured indicators during the update and the specified indicators by the manager gives insights about the progress of migration.

9.3. Application Evaluation

In order to preliminary evaluate the idea to plan future concern associations, we show an update of an application package in the following. We use a package of the spring petclinic demo application that is upgraded from inheritance to annotations to evaluate our approach. Thereby, we use the case that the demo application is updated from version 2 of the spring framework to version 3 of the spring framework. Our goal is to apply our planning approach to measure the update progress of the application until it applies annotations instead of inheritance, what is recommended in the new spring version. Clearly, we apply Hypermodelling to compute the necessary concern association figures and test if we can plan the update.

The demo application is available publicly in an old version with inheritance and in a new version with annotations.[2] Since, the source code of both application versions can be accessed publicly everybody can control our computations, estimations and the figures of the actual annotated version to justify our work. Furthermore, we used only an excerpt of the application to ensure that all named figures can be counted “by hand” and no special technology is needed to follow and verify our computations. Also, we decided to demonstrate our approach with a part of this publicly available application to ensure that we use valid program data that is updated. Another reason, we selected this application was to use an application that is not updated by ourselves. This way, we ensure the validity to real world updates of an industry like application. Lastly, we made the future figure plan of the new version without looking into the actually accomplished/implemented version. Through this, we were able to see if our planned figures were really representing the reality of figures in the new version. Additionally, we present all the artifacts that we used for our computation in various excerpt listings in this chapter.

First we show the figures of the application at the package level in the old version. Then, the top-down plan is shown and the new code based on the new version is presented and compared with the plan.

9.3.1. Code before the Migration

We compute indicators for the contents of a package before an update in the following. Then, we aggregate the indicators to the package level. If the package names are to long, we shortcut their prefix sometimes with a "*" and use a unique identification string as suffix.

Computing indicators for a package

In Example 9.1 we present associations of classes and their supertypes. The corresponding occurrence counts are presented in Table 9.1. The code in Example 9.1 shows classes in the org.springframework.samples.petclinic.web package of a demo application. We show that various classes inherit directly from the SimpleFormController class. This class represents a basic functionally that can be used to built controller classes that follow the corresponding controller design pattern. The last class, ClinicController, inherits from a more complex controller class and

[2]The sources can be found in the version repository: https://github.com/SpringSource/spring-framework - 7.11.2012

also implements an interface. Therefore the ClinicController is a member in two different slices at the same time. First, it belongs to the MultiActionController slice; and second it belongs to the InitializingBean slice. Table 9.1 shows the classes of the package and the corresponding occurrence counts for the type-slice associations. The total shows the complete amount of members that belong to a slice in the package.

Example 9.1. Class definitions in the *.petclinic.web package

```
abstract class AbstractClinicForm extends
    org.springframework.web.servlet.mvc.SimpleFormController

class AddOwnerForm extends
    org.springframework.samples.petclinic.web.AbstractClinicForm

class AddVisitForm extends
    org.springframework.samples.petclinic.web.AbstractClinicForm

class ClinicController extends
    org.springframework.web.servlet.mvc.multiaction.MultiActionController
    implements org.springframework.beans.factory.InitializingBean
```

Table 9.1. Example 9.1 expressed in a table

Parent classes	**SimpleForm Controller**	**Abstract ClinicForm**	**MultiAction Controller**	**InitializingBean**
Classes	***Inheritance count***			
Abstract ClinicForm	1	-	-	-
AddOwnerForm	-	1	-	-
AddVisitForm	-	1	-	-
ClinicController	-	-	1	1
Total	1	2	1	1

Package aggregation

We show the aggregation of the figures of the *.petclinic.web package (see Table 9.1) to the package level in Table 9.2. Thereby, we aggregate the packages of the supertypes to the package level.

We see in Table 9.2 that the supertypes MultiActionController and also the SimpleFormController are out of the *.mvc package. Thus, the total count at their package level is 2. InitializingBean is defined in the *.factory.InitializingBean package and AbstractClinicForm in *.petclinic.web. Therefore, we see the relation of the *.petclinic.web package to supertypes that come out of tree different packages.

However, we recognize that the dependency on the *.petclinic.web package is quite useless, when we want to plan an update against a new framework version, since it is from the application itself. Therefore, we apply the rule to use parent types of parent types, until a point in the inheritance hierarchy is reached, where the supertype is defined to be of a framework type. In our case, this rule means: AddVisitForm is inheriting AbstractClinicForm and AbstractClinicForm is inheriting from SimpleFormController. SimpleFormController is not of the application package anymore and belongs to the spring framework. Thus, an aggregation to the package of the SimpleFormController can be done. We use the inheritance hierarchy to count the total ancestors of SimpleFormController. We apply this rule and recompute the indicators and end up in Table 9.3.

Table 9.2. Aggregation at the package level of Table 9.1

Package of Parent-types	**org. springframework. web.servlet.mvc**	**org. springframework. samples.petclinic. web**	**org. springframework. beans.factory. InitializingBean**
Package	***Inheritance count***		
org.springframework. samples.petclinic.web	2	2	1

Table 9.3. Aggregation of Table 9.1 with ancestor relations

Package of Parent types	**org.springframework. web.servlet.mvc**	**org.springframework. beans.factory. InitializingBean**
Package	***Inheritance count***	
org.springframework. samples.petclinic.web	4	1

9.3.2. Migration and plan

Now, a software manager plans a migration to the new version. He knows that the new way of building controllers is done by annotations and not by inheritance any more. The manager plans the new measures of concern associations at the package level by looking at the current figures (Table 9.1, Table 9.3). He ends up with a plan in Table 9.4 that shows the aim to have inheritance anymore and uses annotations. Likewise, out of the upgrade instructions and out of the number of distinct classes in the package, the manager estimates that probably three classes will be in need of an annotation. The reason he estimates three can be explained quite simple; three non abstract classes are in the package and use inheritance.

Table 9.4. Plan of Future Package Concern Figures

Package of Parent-classes	**org.spring framework. web.servlet. mvc**	**org.spring framework. samples. petclinic. web**	**org.spring framework. beans.factory. InitializingBean**	**org.spring framework. stereotype**
Package				
Indicator:	*Inheritance count*			
org. springframework. samples. petclinic. web	0	0	0	0
Indicator:	*Annotation count*			
org. springframework. samples. petclinic. web	0	0	0	3

One could say that the estimation of the manager could be four. However, four is still in the right direction. The main principle that is demonstrated in this sample is: future concern associations can be estimated and planned with figures. The quality of the plan will finally, like in every business scenario, depend on the planning experts' knowledge.

After the manager plan is made, a programmer gets involved with the task to update the package to a new version. The programmer implements the classes as shown in Example 9.2. The whole coding time of the developer, the current concern associations can be computed an aggregated and compared with the plan. The developer encodes the new AddOwnerForm and replaces the superclass with an annotation. Therefore, we get the indicator of 4 times used inheritance and 1 annotation. These indicators can be compared with the goal figure of three annotations and zero inheritance and be used to compute an indicator for the progress. As a matter of fact, the developer goes on and deletes the unnecessary AbstractClinicForm in the new version and adds the rest of the annotations and finishes his task. All the time, while he encodes logic and alters the source code, the progress is measured and compared to the goal figures. Continuously, this comparison can show the distance to the goal figures. We imagine that a graphical representation can be done with a progress bar or similar visualizations in a cockpit.

Example 9.2. Realization of the plan from Table 9.4

```
@org.springframework.stereotype.Controller
class AddOwnerForm
@org.springframework.stereotype.Controller
class AddVisitForm
@org.springframework.stereotype.Controller
class ClinicController
```

We do not limit ourselves a specific algorithm that compares the figures and computes and indicator of the progress. We do not do this, because in this chapter, we describe the generic method and different algorithms are imaginable. Additionally, the comparison can be done at the package level or at the top level of all types. Therefore, a specific algorithm can work at different granularity levels. Therefore, we leave this to future researchers to come up with the most precise progress measurement computation.

We note that the manager can be wrong with the planned figures. But, still, the advantage exists that a progress in a plan can be measured by comparing the current figures with the plan. This way, it is possible to uncover easily if everything works according to the specified plan and progress is made. As a matter of fact, the realization of a plan and its adjustments will probably be a cyclic process, where figures need to be adjusted from time to time. Future researchers can dig into the specifics.

9.4. Discussion

One may bring up that the main reason for different concern encapsulation techniques is to avoid code clones and not realize new associations (1). Then, he may argue that our method is not generic (2). Additionally, it may be introduced that our demo application update is not an concern typical case, since we used controllers and the "controller" case is more about infrastructure capabilities, wherein the application is running (3).

We argue against these viewpoints: (1) First, we introduced our approach on a generic level of vertex associations. Such vertex associations can be computed for the different cases and change over time. Regardless why modularization techniques are introduced, we have to face reality, where this shift happens quite often. (2) Secondly, we mentioned that the update progress can be measured by comparing the vertex association figures of the current and future version. This is generic and the function that expresses the update progress is not limited to a specific case and can vary for different use cases. (3) Third, we believe that being a controller is clearly a concern of functionality. Similarly, Aspects, like security or transactions, are often also expressed as annotations nobody argues against it. Our method would also be application able for those.

Additionally, we argue for the validity for our approach by using a vertex based concern association measurement based on an informed argument. Others researchers express and understand concerns, their associations and a programs source code also as graph [220]. Our method advances this graph based approaches by using the concern graph associations to compute, plan and compare the different vertex association movements. Additionally, we presented first results of its application as preliminary evaluation. Those results underline that our method advances the current concern graph approach and further research is needed.

9.5. Potential Synergies

Concern graphs [220] show a formal model to describe concerns in a program. Planning of future concerns is not mentioned. However, other research reveals techniques to uncover non modularized (not directly perceptual) concerns in software via diverse algorithms [43]. Thus, it would be interesting to investigate the application of our approach to plan future concern implementations of concerns that are not directly perceptual and not modularized, yet. In special, not directly perceptual concerns can be uncovered with the algorithms of related research. Their planned modularization

can then be expressed as a future plan with our method. Finally, their manifestation progress into modules can be tracked by our method. Once all concerns are modularized, the plan is fulfilled.

ConcernMapper is a tool to associate concerns with code fragments manually [219]. It is applied within Eclipse and the concerns can be used to filter the development environment. No future concern planning is supported. Similarly to the prior mentioned research about uncovering concerns via algorithms, we see it as an interesting avenue to determine if ConcernMapper can be combined with our future planning method. Hence, a combination of manually described concerns in ConcernMapper combined with the possibility to plan future concerns could help developers to modularize concerns in the future.

Additionally, our work relates to the area of business computing, where enterprise reporting and planning is done with Data Warehouse Systems. In [178] a detailed description illuminates how economic indicator based planning scenarios are done. Our work here differs, because we do not focus on economic indicators. Rather we use the Hypermodelling attempt to utilize Data Warehouse technology for software engineering. Therefore, we advance economic planning scenarios by the possibility to plan indicators for source code.

Also, our work relates to research about program updates. For instance, research reveals the different update types that can occur [91]. In comparison to that, we presented only one update type within this chapter. The main difference is that we focused not on the update types and rather presented a possibility to express associations and their updates in numeric indicators. Those can be use to plan an update in a structured way. However, the different proposed update types and our approach can possibility be used together to compute additional indicators.

Other researchers focus on updates of application program interfaces and their usage [194]. Thereby, they reveal a method that compares two versions of an application program interface to determine changes. In contrast to that, our presented works focus is to plan updates and compute indicators for those. This is rather different, since our approach targets the case where a full automatic update is not possible. However, we are confident that the approaches combination can lead to hybrid approaches that allow doing both things at the same time; support users through tracking application program interfaces changes and controlling future indicators in the migration progress.

Similar to the prior mentioned update supporting approach is a method that focuses on the updates of libraries [62]. Again, the main intention of the approach is to track the main changes. The proposed tool compares the differences between old and new versions and recognizes changes in the library use. This information is used to propagate the specific changes over the program. Again, the scope of the proposed technique is differs from our approach. Our scope is to plan with indicators that can be used to track the progress in an update process and not to determine change sets. Therefore, we assume that future research can reveal potential synergies between the techniques.

9.6. Summary and Conclusions

We revealed the problem of concern association movements in a program update. We proposed to solve this problem by the method to adapt the planning mechanisms of Data Warehouses for concern association planning. First, we showed our approach and the motivation for future concern association planning. Thereby, we expressed the concern association movements in a program as vertex associations to make our approach transferable to other concern associations than annotations and inheritance. We performed a concrete example how figures of annotations and supertype associations in packages can be computed. We used and referred to the Hypermodelling approach that is capable of computing such figures in general. We depicted how future concern associations can be planned via figures. In order to do a preliminary evaluation, we presented an excerpt of a demo application that was updated against a new version and applied our method to it. The figures of our plan equaled the new version of the demo application and gave first indications that our approach is feasible. A discussion underlined the validity of our approach and the need for further research. Finally, synergies with related work were pointed out and showed that further research is needed. However, our approach can already be used to plan future concern associations and to measure the progress of updates from inheritance to annotations. Managers can now compare current and

future indicators to gain insights of an update process. Researchers can dig into concrete algorithms to figure out the best ways to compute progress indicators and investigate the application of our method in large scale projects for other kinds of concern associations.

In general, we introduced the new idea to enable top-down concern planning at the package level and to measure the update progress of and application. We are certain that systematic top-down concern planning has huge advantages for project management in software development. All together, our concept to model future concerns is new and can lead to advance project management in update scenarios. Further investigations need to be done to verify and evaluate our approach better. Furthermore, other concern associations than annotations and inheritance need to be considered. Additionally, we plan to work together with software project managers to apply our approach in real world scenarios. This will help to advance our approach for specific projects needs.

Lastly, our approach addresses practical needs in real world projects. Such projects are managed and they are not as perfect as the theory and resources, like developers, have to be scheduled. Hence, we argue that further investigations to help planning updates of a program and measuring the plans progress are needed. This is especially the case, when new mechanisms to separate concerns, like Aspects, are introduced. Managers want to estimate time and effort of an update, because they want to schedule their resources. We showed that concern associations in a program can be computed and the planning of the future concerns is possible. Through this it is possible to gain insight of the progress of an update. The contribution of this chapter will enable further research about supporting project management with the ability to plan future application versions and concern encapsulations.

We see the emerging need to identify further concern associations and indicators how updates can be measured and planned. We propose to identify further scenarios how concern updates are done and to specify the related indicators. Therefore, we see it as necessary to investigate in detail how the Hypermodelling approach can be utilized for often occurring scenarios.

Additionally, we address the necessity of research about tools and methods that enable easy to handle management support on various abstraction levels. We refer to the very main idea of this chapter to borrow techniques that are used in the area of Data Warehouses. We adapted a small piece of the planning idea from there and think a more detailed investigation of economic planning scenarios may hold clues to advance concern-association planning in general.

All together, we described the general approach to use Hypermodelling for future concern planning scenarios. With this, we leave the typical Data Warehousing application scenarios behind and focus, in the next chapter, on the application of Hypermodelling within the development environment.

10. Hypermodelling the IDE

“If by some miracle you manage to get the computer working again you've got to enter the code...”
Desmond. Lost Season 2, Episode 3: Orientation. 2005

This chapter shares information with:
"T. Frey. Hypermodelling for Drag and Drop Concern Queries. Proceedings of Software Engineering 2010 (SE2012). Springer. 2012"[97]

Abstract. Imagine a developer, who wants to alter the service layer of an application. Even though the principle of separation of concerns is widespread not all elements belonging to the service layer are clearly separated. Thus, a programmer faces the challenge to manually collect all classes that belong to the service layer. In this chapter, we use Hypermodelling to query for the necessary code fragments that belong to the service layer. Slicing and dicing the code in this new way overcomes the common limitation of having only one viewpoint on a program in the IDE. The service layer can now be uncovered just via a query. Additionally, we give a vision of future IDEs architecture that utilizes Data Warehouse components actively. In order to evaluate this architecture we present a search engine based on Hypermodelling. This shows that Hypermodelling can leverage new possibilities in the development environment. Now, researchers can come up with new ideas based on Data Warehousing components within the IDE.

10.1. Introduction

In the previous chapters, we focused on original Data Warehouse use cases for Hypermodelling. Right here, we transfer Hypermodelling to the development environment and enable concern queries therein.

Programmers spend most of their time reading and navigating source code [257, 219]. Often developers alter an application layer [93]. Usually the developer tediously gathers the artifacts that belong to the desired layer. This is a typical instance of the concern retrieval problem that programmers face commonly. Locating concerns in software is indicated to be a challenge for software developers [161,94, 96]. Probably the main reason why locating concerns in source code is so difficult is because their intertwining. Normally, not all concerns are packed into separate modules and therefore a module encodes multiple functionalities at the same time [198].

Modern development environments allow building working sets, consisting of one or more fragments corresponding to the various interests for a developer [145]. Such working sets need to be built manually by a developer and can be used to filter the navigation just for the elements belonging to the working set. Typical for working sets is to split the application into their distinct layers and organize those as different working sets. This concurrent view of elements belonging to a working set is beneficial for developers by allowing them to navigate between the various fragments. However, still, the challenge of code investigation to build the working sets exists. For instance, when new classes are added to a layer, they need to be associated with the corresponding set.

To support programmers in investigating source code beyond working sets several tools are available for concern analysis or query operations [219, 74, 77]. The query tools offer complex and advanced query languages. This leads to the fact that composing a query is quite complicated [74, 139]. It is indicated that a few query tools seem not to be enhancing productivity. This could result from the complexity of the tools [74].

In order to overcome the above mentioned limitations of manually creating working sets and complex queries, we present the Hypermodelling approach for the development environment. Hypermodelling enables developers to design concern queries without the need to learn a complex query language like it is used in other tools. Developers can use known views and elements via drag

and drop in their queries. The same view wherein the working sets are shown is used to present the result of a query. This way, we avoid dealing with a custom query result representation view, like other tools use it. Developers can now do queries for the various layers of an application without the need to organize them into working sets. Additionally, videos are available, showing our implementation in action.[1]

10.1.1. Contribution

Thus, this chapter contributes to this our work about Hypermodelling, by showing a new application scenario of Hypermodelling for the IDE. In special, the contributions are:

- Scenario for the IDE

  We present the scenario to apply Hypermodelling within the development environment for concern queries.
- IDE Implementation

  We provide a custom implementation of Hypermodelling for the IDE. Thereby, we utilize further research, containing which views are most used by developers, and combine it with Hypermodelling to create the implementation. Our implementation enables developers to drag and drop concern queries from well-known views. The query result is presented through a filtering mechanism of the most used view.
- Use case based evaluation

  We evaluate the benefit of the tool by showing use cases. The use cases indicate a benefit of the tool.
- A future Data Warehouse technology IDE architecture and its first realization-evaluation in form of a Hypermodelling powered code search engine

  We show a future IDE architecture and how Data Warehouse technology can be integrated into it. Furthermore, we evaluate the suitability of Data Warehouse technology for code search, by providing a code search engine implementation based on Hypermodelling.

All over, we address the complex query composition problem of other IDE tools by utilizing the Hypermodelling approach for the development environment. This allows developers to design queries for code slices and their combinations, easily. Furthermore, we present insights how future IDEs can utilize concrete Data Warehouse technology and evaluate the feasibility through an Hypermodelling based implementation of a code search engine. Now, researchers can implement Data Warehouse technology in the IDE and come up with more application scenarios.

10.1.2. Reading Guide

The chapter is organized as follows. First, we describe an IDE implementation. Afterwards, we present a preliminary evaluation. Then, we present a future IDE architecture with Data Warehouse components. Afterwards, we evaluate the application of OLAP components in the IDE with a first code search engine implementation. Finally, we draw conclusions and explain future work paths.

10.2. Hypermodelling the IDE

In the following, we describe the Hypermodelling implementation for the development environment (Eclipse). The Hypermodelling tool (plug-in) for the IDE is implemented by a custom query processing engine. This was done to show the application of Hypermodelling in the IDE before implementing Data Warehouse techniques therein. With the current implementation, we inspect source code files in real time. This has the advantage that new concerns in altered source code files are detected when a new query is executed.

[1] See the project homepage at http://hypermodelling.com - 7.11.2012

Next, we describe the user interface and way of operation of our tool. Then, we give some technological insights about the implementation.

10.2.1. Use of the Implementation

In the following, we describe our tool and its usage. We start by describing the views in the IDE on a high level and get afterwards into details about the way of operation and advanced capabilities.

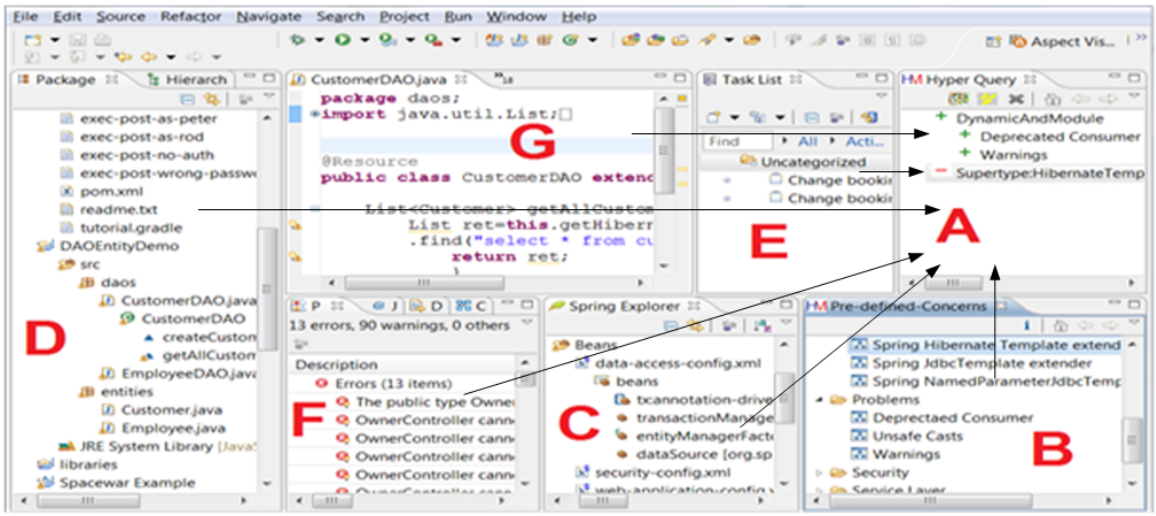
User Interface

In general, our tool supports well known views as drag sources to compose a query for concerns. This avoids that developers are forced to learn new views. We present a visualization of the Eclipse IDE with our tool (A, B) extension in Figure 10.1. We indicate the drag and drop capabilities by arrows to the "A" part of the graphic. We see that a developer can compose a query by dragging elements from known views into the query view. This way, the concerns in the views, showing source code, libraries and metadata (B-G) can be added to a query.

We filter the package explorer (D) to present the results of a query. This avoids that programmers need to learn a new view. We use the package explorer, because it is one of the main tools that developers use for code exploration. Furthermore, we use a filter, because information reduction is a central element to better the overview of a developer.

In the case of the Eclipse IDE the most used (clicked) view of developers is the package explorer [189], which shows a hierarchical structure of projects, packages, classes and methods. Hence, we assume that it is the package explorer is the most used for source code exploration. Additionally, researchers found indications that structured code investigations of developers are effective [218]. Therefore, we credit the way how developers explore source code in a structured way and use the package explorer as result presenter.

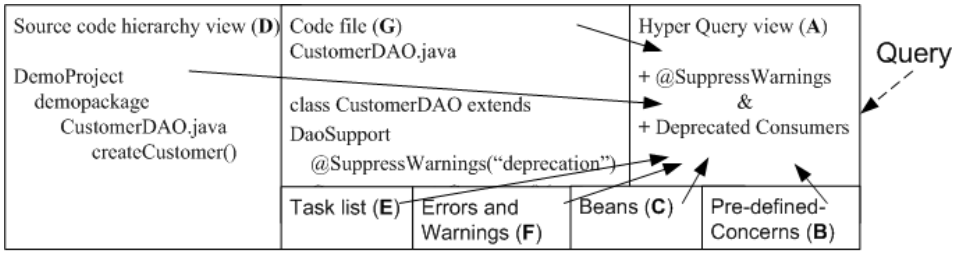
Figure 10.1. Hypermodelling IDE Implementation - Screenshot



A user can compose queries out of the different views of the IDE (B-G). He drags and drops elements from them into the query (A). The query results get shown in the package explorer (D)

In Figure 10.2, we present a schematic of the IDE. The letters correspond to the ones in the screenshot (Figure 10.1) and the arrangement is similar. This schematic view shows the main idea of our approach: Different views of an IDE serve as drag sources and developer can compose a query via drag and drop. Then, the result of a query is presented in one or multiple well known views, by a filter mechanism. In the following, we describe the views that we use in our prototype as drag sources.

Figure 10.2. Hypermodelling IDE Implementation - Schematic



We see a schematic view of an IDE that presents the main idea. The arrows visualize the drag and drop of concerns from well-known views into the query. This way, we indicate that there can be even further views that are capable to be drag sources. The general schema is that developers can use views as drag sources that are employed during programming. Once a Hypermodelling query has been executed, it is possible to filter all the views of the IDE (B-G) for the results. This enables to study query results in well known views.

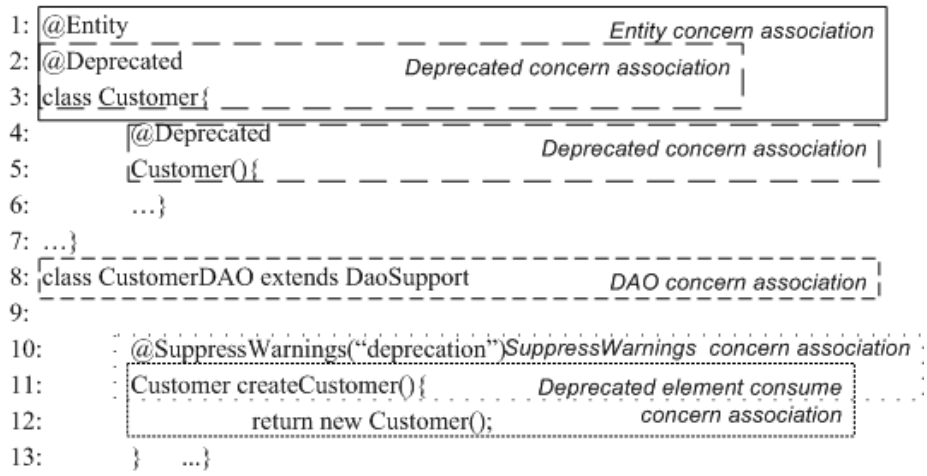
In addition to the ordinary views of the IDE, we realize a view where well known concerns are listed (B). Those well-known concerns are elements like annotations, different superclasses or meta-information. For instance, we offer a generic template for inheritance to define a slice of superclasses children. Similarly, meta-information like unsafe type casts can be added to a query. Also, a template for a string in a class or method name can be added to a query. Aside from generic templates, we also offer ready-to-use concerns (e.g. children of a DaoTemplate class). Generally, the main idea of the pre-defined concerns view is to enable a fast access to often used concerns. This is out of the intention to enable a systematic query creation and to offer a sets of concerns that a developer would have to search effort full in the normal views otherwise. Additionally, we offer combinations of concerns, like different concerns that belong to one layer as concern units in this view. Therefore, we call this view the pre-defined concerns view, because standard concerns can be offered this way.
The other views (C,D,E,F,G) show normal views of the IDE from which concerns can directly be dragged and dropped to a query. In general, we follow the Hypermodelling idea to use code structure (D,G) and associated data (C,E,F) as part of a query.
We support various code elements on a generic base. This means that new annotations, classes, aspects, packages and similar things can be dragged and dropped out of a code fragment into a query (G). This way, we enable the addition of elements that are defined in programming libraries and in the code itself to a query. This is possible, because the different elements in Java are defined through language mechanisms. This has the advantage to present a subsidiary possibility to add concerns to a query by using familiar views. This way, it is possible to populate concern data of an actual investigated program into queries.
View C shows a bean view of the IDE where classes that are defined as beans are visualized. E shows Mylyn Tasks with which different fragments may be associated.
D shows the normal package explorer. From that, Java Elements or groups of Java elements can be added to the query. This makes it very easy to include or exclude elements in the view from a query.
F shows the error and warnings view as drag source. From G, the editor, annotations can be added to a query.

Way of Operation

In the following, we describe the query capabilities. The query view (A, Figure 10.2) shows the Concern slices that are included ("+") and excluded("-") in a query. For example, from the following Figure 10.3, the @Entity slice would consist of the class Customer. The @Deprecated slice contains createCustomer(). Another result for the query of "+" DeprecatedConsumer is createCustomer(). So, if "-" SuppressWarnings is added, an empty result would be the outcome. For this reason it is possible to tie slices with an "AND" together, which we visualize with the "&" in the prior IDE schema (see Figure 10.2). A query +DeprecatedConsumer & + SuppressWarnings determines all consumers of deprecated marked elements that suppress compiler warnings at the same time.

This way, developers can reveal deprecated element consuming code fragments, showing originally compiler warnings, before their suppression. This is quite useful, to determine not updated spots in a program that do not excel at compile time or in the errors and warnings view.

Figure 10.3. Exemplary Code Slices

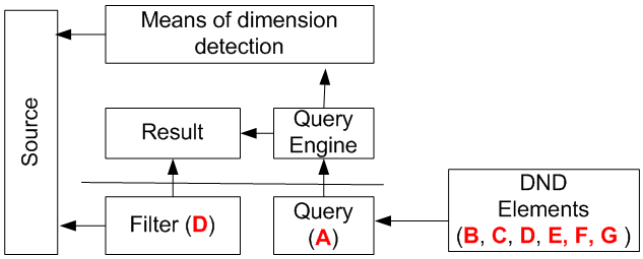


We recap Figure 5.4 here again, to reuse its contents here for explanations.

We visualize our implementations the way of operation in Figure 10.4. There, the letters (A-G) correspond to the ones in Figure 10.2 and Figure 10.1. Concerns are selected via dragging and dropping them from the various views (B,C,D,E,F,G), into the query view (A). Then, the internals of the tool process the query and the user and gets the result with a filter presented.

At execution time, the query engine uses different means of dimension detection to determine fragments in the source code, belonging to the query specified dimensions. When all fragments that correspond to the query are found, they are stored in a result. This result is used to apply a filter to the package explorer (D). Thereby elements are excluded that are not in the result set. Thus, users can benefit from their previous knowledge about the mode of operation of the package explorer.

Figure 10.4. Tool Operation



We visualize the tools internal way of operation of operation. Elements are dragged and dropped from a view into the query. The query gets executed in a query engine and uses means of dimension detection to reveal the code elements that correspond to the query. Out of this, a result is generated and used to filter the view of the source code until only the elements of the results remain visualized.

Advanced Capabilities

Plenty of concerns for drag and drop are supported in queries. We give an abstract overview about the kinds of concerns that we support via drag and drop in Table 10.1. We see the concerns grouped by their kinds. Furthermore, we see that not only code structure is supported, but also associated information, like compiler output and beans. A specialty is also the possibility to use conventions with strings in a query. This makes it possible to define slices of classes that contain the string "dao" or similar things in their name. This can be used to query for all extenders of a "DaoSupport" class that do not carry the string "dao" in their name. The result of this query can provide insights about violated coding conventions.

Table 10.1. Supported Concern Kinds

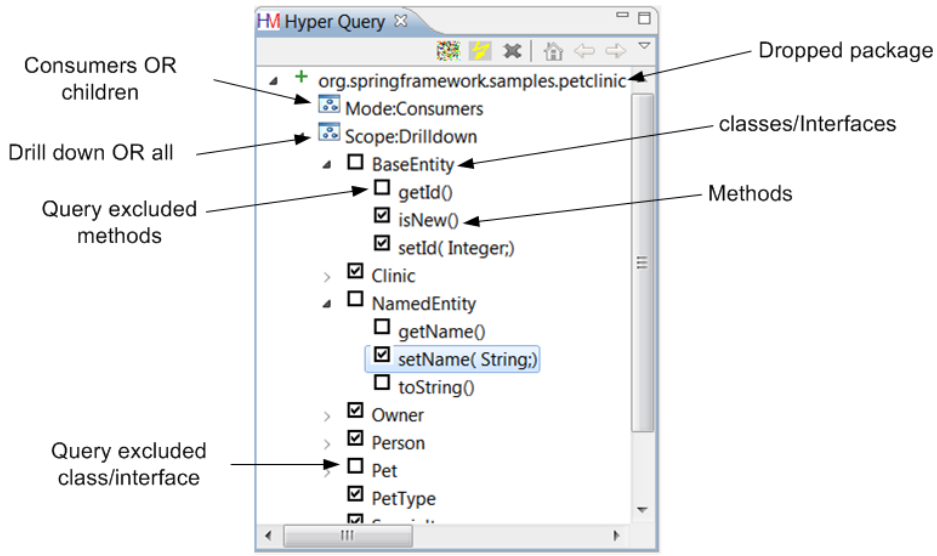
Concern kind	Support in the Implementation
Inheritance	Extend and implement
Compiler Output	Unsafe casts, Problem markers
Domain specific	Bean definition, Annotations
Current Code of interest	Drag and Drop Java Elements, Search Results, Tasks
Coding Conventions	Generic class name contains a string
Aspects	Advice affected Elements
Consumers of Elements	Method, Type (Class, Interface), Package

As a matter of fact, aside of plenty of supported concerns, the tool supports even more advanced operations. A specialty that we note here is the support for drill downs. A package can be dragged and dropped into a query (view A) and be the source for a drill down. We show a dropped packaged in Figure 10.5. When such a drop is done, the user can select if the children (packages lower in the hierarchy or contained types) of this package are desired to be the result of the query.

Additionally, developers can specify if the result should contain the consumers or the lower ranked elements of the queried package. Lower ranked elements are the elements sub packages or types within a package. Consumers are elements that refer to a package. Such references can be method calls into a package, type instantiations and similar programming constructs. Consequently, both options support drill downs.

Right now, we go exemplary in the case of consumers. We imagine all consumers of the package a revealed and it gets of interest which specific types of the consumed packages are used. Therefore, the all members can be expanded with a drill down to the concrete types. This was done in Figure 10.5, where we see that different types are marked and some have been deselected. This means that only consumers of marked types get revealed. Hence, a drill down to types is done. However, users can even go further, as we show, and drill down into the concrete methods. Like in case of the types, consumed methods can be excluded and included. This way, systematic top-down investigation is possible.

Figure 10.5. Drill down (Query View)



We show how a package can be used for a drill down. A package can be dropped into a query. A selection specifies if its children in the form of classes and interfaces are desired to be query results or if the focus is set on consumers of the package. In any case, the contents of the package can opened via a drill down to its contents (classes, interfaces, members). All expanded elements can then be included or excluded from a query.

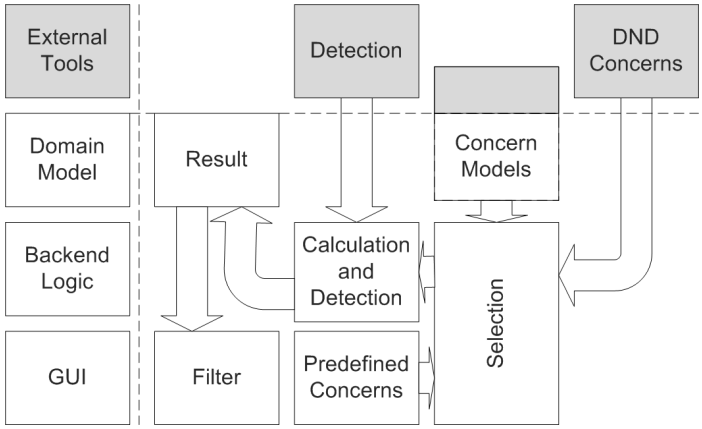
10.2.2. Implementation Insights

In the following, we give quick overview about implementation insights. First, we describe the different internal units. Then, we illuminate which other Eclipse plug-ins functionality is used and extended by our plug-in. Succeeding, we illustrate how the internal query sequence of our tool works and get, finally, to a schematic cube view that visualizes our internal model.

Internal Units

Figure 10.6 shows how the different units work together. On the left side, we see the different logical units, which indicate that the boxes in the same horizontal layer belong to them. Additionally, we see the External Tools are marked grey to indicate where we use third party elements. Beneath those, we see the Domain model which refers to the internal data model of our application. Our internal processing logic is defined as Backend Logic. Finally, we offer our contribution to the user interface in the graphical user interface (GUI) axes. In the following, we describe the different units in more detail.

Figure 10.6. Internal Units



We show the classification of the different components of our tool on the left side of the figure. Furthermore, we see that our plug-in works together with third party plug-ins, what we indicate with different grey colored elements. The arrows between the different components indicate the information flow.

We see that the usage of External Tools as drag and drop sources (DND Concerns) are the various views of different Eclipse plug-ins. We hook drag and drop functionality into plug-ins for Aspects, Java Elements, Beans, Problem visualizations and Task context. Therefore, the DND Concerns functionality is arranged in grey at the level of External Tools, because it is dependent on third party plug-ins.

Additionally the pre-defined concerns are offered as drag source. Since the pre-defined concerns come purely out of our development efforts, we arrange them at the GUI level. The arrow to the Selection indicates that the pre-defined concerns can, like the DND concerns, selected to be part of a query.

Once concerns are dropped to the query, a concern object from the Concern Models is instantiated. This means that classes are instantiated that reflect the dropped concern. Additionally, the concern objects are referencing the elements of external tools, where they are originally defined. Thus, the model of those concerns consists partly of elements that are originally from the external tools and is, therefore, shown half grey (Concern Models).

Once the query is executed, the concerns defined in the query are detected (Detection) within the source code and the Result is calculated (Calculation). Currently, the multi dimensional model used for calculation and detection is implemented in an imperative way, by scanning the elements within the IDE. Scanning utilizes different third party plug-ins for the detection, what we show through the

grey color. Every scanned element is compared with the concerns of the query and corresponding elements are put into the query Result, considering the computation directions, like "and", "or" and "without".

Finally, we use the Result of the query together with the filter component. The Filter docks into the GUI and restricts the view to only to the elements of the Result. Thereby, the Filter is intelligent and follows the logic of the view. This means that if just a class is in the Result, the Filter includes path (packages) to the class. This has the reason to let the filtered view look like the original view, just with less elements.

Used Extensions

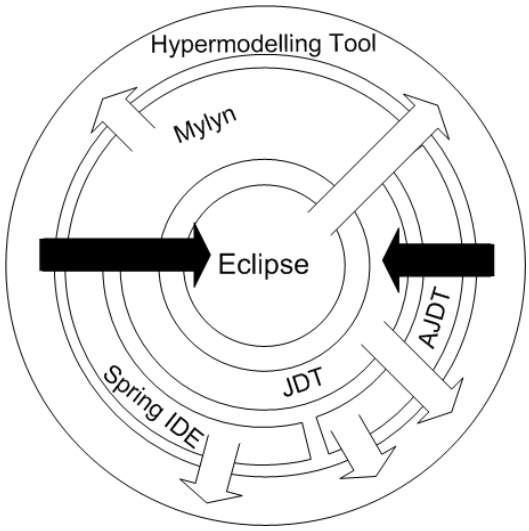
Right before, we described that the tools internals way of operation depends on other third party plug-ins in the IDE. Those are needed for concern detection and to allow dragging and dropping elements into a query from the various tools.

Therefore Hypermodelling integrates into Eclipse by leveraging capabilities of various other third party tools that also integrate in the development environment. We give an overview in the following. Figure 10.7 shows the Eclipse platform on which the Java Development Tools (JDT [136, 23]) are built. The circles are used to symbolize that the Spring IDE [25] and the AspectJ Development Tools (AJDT [24, 59, 27, 69]). Those tools provide enhancements for Bean definitions and AOP development. Mylyn [26, 139] enables to work task focused. Elements out of the Spring IDE, AJDT and JDT can be associated with the tasks.

We indicate that the different plug-ins work together, by arranging them from the inside to the outside. We see that all tools are founded on the Eclipse IDE and AJDT and the Spring IDE are dependent on the JDT plug-in. Mylyn is arranged on the same level as JDT, the Spring IDE and ADJT to indicate it works together with all of them and it is not depending on those.

The Hypermodelling tool uses the distinct tools to realize the drag and drop and query capabilities and is therefore arranged in the outermost circle. Concerns from the various tools can be dragged and dropped into the Hypermodelling tool as shown by the white arrows. These are used to compute results for a query and to filter the package explorer view of the JDT. Hypermodellings enhancement of JDT and the Eclipse platform itself is visualized via the black arrows to show the contribution of the Hypermodelling tool. In detail, the Eclipse IDE is extended with new views and the JDT is extended with a filter capability of the package explorer.

Figure 10.7. Big Picture



We show that the Hypermodelling Tool for the IDE uses different plug-ins of Eclipse (white arrows). It combines functionality of those and then contributes itself to the Eclipse IDE and to the Java development tools (JDT) (black arrows).

We use Extension Points [122] and other internal functionalities (e.g. the internal data models) in order to integrate our Hypermodelling logic. Right here, we do not get into implementation details and focus on the big picture. For general details about extending the Eclipse IDE and the plug-ins, we refer to related literature [122, 14, 13, 12, 231, 232, 138, 262, 66].

Query Sequence

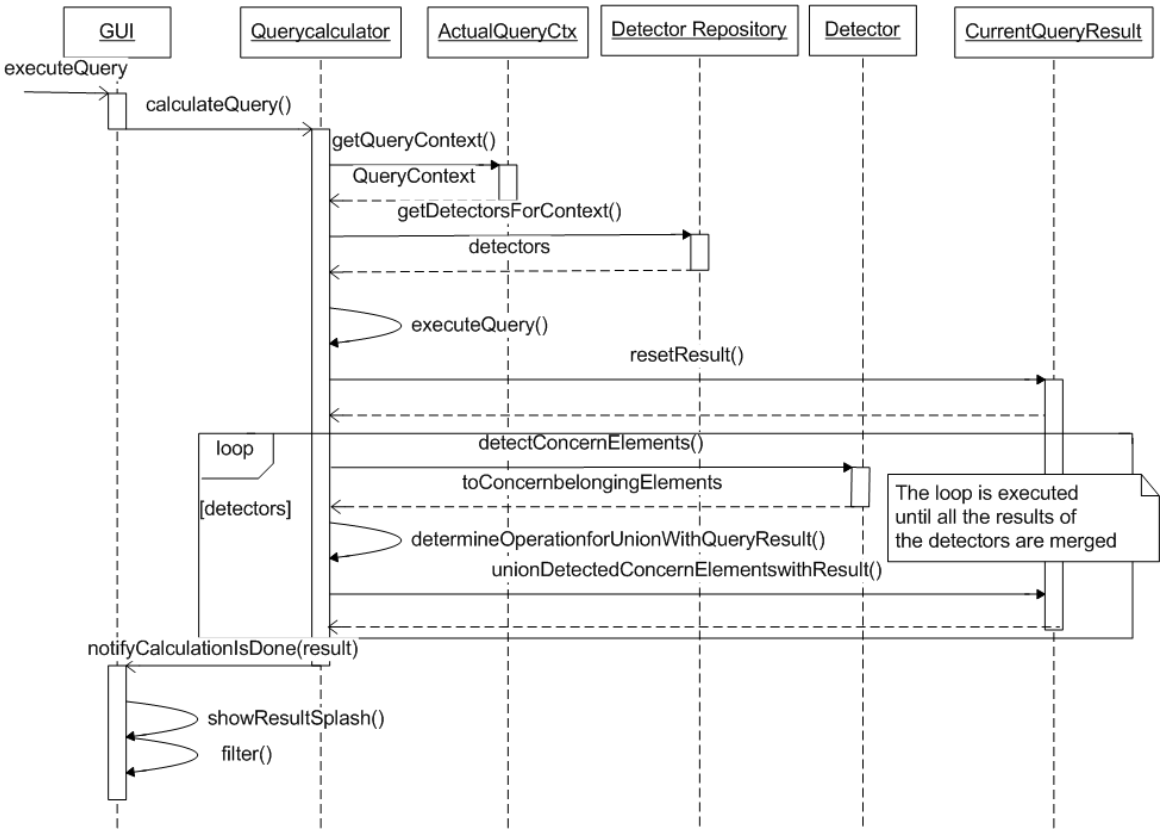
Like described before, the tool is implemented via a custom query processor. The internal processor, called internally QueryCalculator, uses so called Detectors that reveal the code fragments that belong to concerns. In detail, the query sequence works as follows:

Figure 10.8 shows a sequence diagram of the query processing. When a query is fired in the IDE, the dropped concerns (ActualQueryCtx) get retrieved from the QueryCalculator. Then, the selected concerns are used to determine the corresponding Detectors that are necessary to uncover their appearance in code from the DetectorRepository. Before the query execution starts the set the result empty (CurrentQueryResult).

Then the loop, responsible for computing the results of a query, gets started. Within this loop code parsing and applying the Detectors is used to identify the concerns in source code. In detail, every code file gets inspected and the different Detectors get applied and determine if there are code fragments within a file that belong to a certain concern. Since a query can be composed with inclusion and exclusion of concerns, every time a Detector reveals code belonging to a concern, the selected operation get determined (determineOperationforUnionWithQueryResult). Then, the determined operation gets used to integrate it into the CurrentQueryResult ("and", "or", "without"). This process is executed until all code files are parsed and all concerns are detected.

Once the query computation is done, the user gets notified via a splash screen (showResultSplash). Lastly, the user can apply the filter for the IDE that is based on the query result and study its contents.

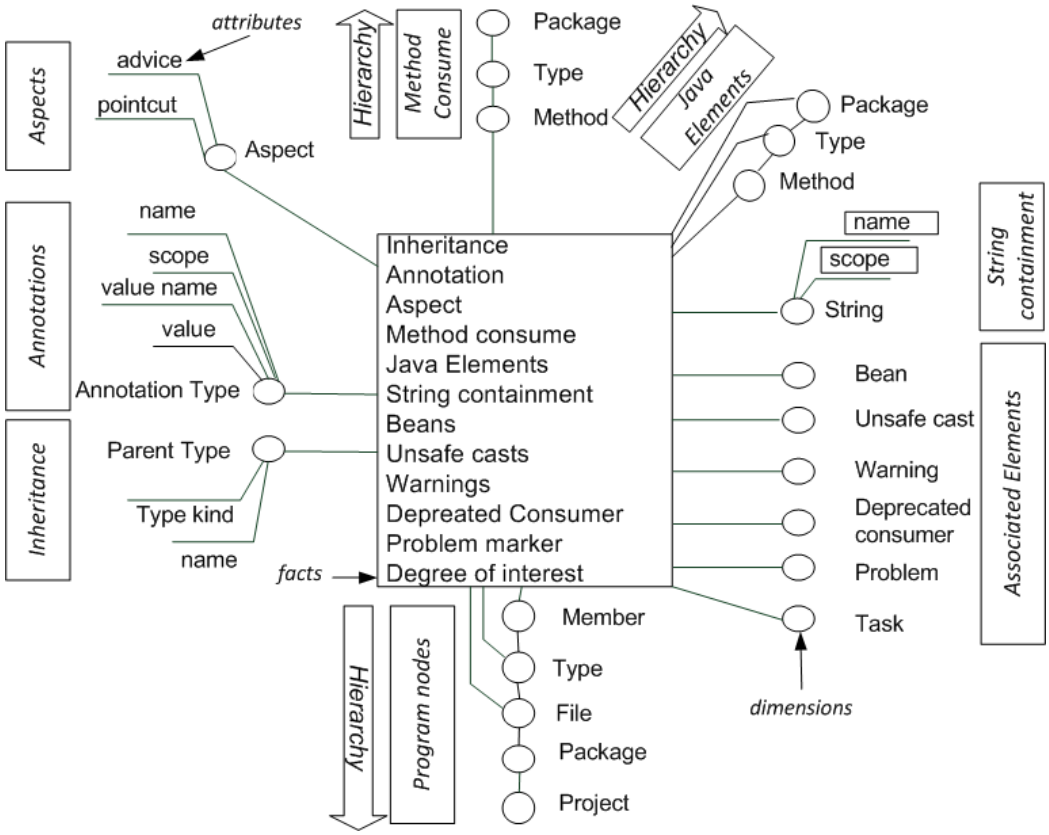
Figure 10.8. Query Sequence



Schematic internal Concern Model

Right before, we described the query sequence and how the computation loop of detected concerns is done. In spite of the manifold supported concerns, we developed the custom computation engine that integrates detected concerns into the result along the lines of the prior discussed Hypermodelling Data Warehouse implementation. Therefore, we present a multi-dimensional model of the concerns that are supported within the IDE tool in Figure 10.9. However, internally the model is not realized in a multi-dimensional way and a set of imperative rules out of the results of concern detection. In plain English: We use an ordinary Java program to compute the results. However, the cube model was created out of the main reason to have an implementation blueprint for programming the Hypermodelling tool for the IDE. Additionally, we derived the model as documentation to be capable to use Data Warehouse technology in later versions of the IDE tool. With the model, there is already the cube specified that is needed to adjust the tool to Data Warehouse technology. In the following, we describe, briefly, the different parts of the model.

Figure 10.9. IDE Cube



In the center, we see the facts arranged, indicating a relation to code elements. We see that different facts are used to describe the different relations. We recognize that inheritance relations are specifying the superclass-children associations in code. Anyhow, likewise the rest of the facts can be described and we refer to Chapter 6 for detailed explanations about relations in code. As a matter of fact, here we see additional facts for classes that are defined as Beans and Unsafe Casts that are associated with source code. Similarly, Deprecated Consumer data, Problems Markers and the Degree of Interest is available as fact. Hence, the facts connect the different dimensions that are shown in the model together and realize their relation to each other. In the following, we describe the different regions of dimensions in the model.

At the bottom, the program nodes, responsible result representation, are shown. They are ordered in a hierarchic manner. The direct connection of the dimensions with the facts shows that various facts can have a relation at diverse levels of the hierarchy. For instance, can facts be associated

with members or with types, like classes or interfaces and files. Members are elements like fields of methods that are a part of a type. Generally, the facts can be aggregated via the hierarchy.

The inheritance fact and dimension connects the type with its parent types. Parent types are classes or interfaces that get extended or implemented. The type kind attribute indicates if the parent type is an interface or a class. The name specifies the name of the parent type.

The Annotation fact connects members and types that are annotated. Thus, the scope attribute is used to determine if an Annotation occurs at a type, field or a method. The name obviously is the Annotations name. Likewise, the distinct parameters that an Annotation can have are defined via their name and value.

Aspects affect various source code fragments via their Advices, where the advised elements are commonly defined via Pointcuts. This is realized as facts that connect Aspects to the diverse program nodes together.

The association of method with its called methods is realized through the Method consume facts. Like shown, the consumed methods are member of a Type what is itself defined in a package. This way, queries, like "Which packages consume methods of other packages", can be answered.

Java Elements are any element within source code. This can be used to exclude or include any method, class or package from a result set. The Java Elements are structured hierarchically what makes it possible to compute the higher dimensional levels. For instance, method can be included in a query as result the containing package and type is computed. Furthermore, all types of a package can be excluded this way. This enables dragging and dropping elements form normal views, like search results, for inclusion or exclusion, into query.

String containments are used to define if a method or a type contains a certain string. The choice between methods or types containing the string is done by the scope.

Associated elements are a collection of dimensions that are associated with the source code. Beans indicate if a class is defined as a Bean in the configuration. Unsafe casts identify methods or types, where casts to generic, non parameterized types happen. Warnings mark elements in the source, where compiler warnings appear. Deprecated consumers define methods or types that use marked deprecated elements. Similar is the Problem marker that realizes an element and the corresponding source code of the problem view. Lastly, the Degree of interest fact connects a task with source elements. Tasks can have associated elements, where the importance of the elements is measured via a degree of interest.

10.3. Evaluative Use Cases

Here, we present example applications for the tool to evaluate possible use cases.

10.3.1. Resolving the Service Layer

A possible application scenario of our tool is to uncover code that belongs to a certain application layer. A developer can compose a query to uncover elements that are belonging to the service layer. Therefore, he composes code slices that belong to this layer together in a query. When he executes the query, he can see results like working sets in the package explorer. For example, the slices in the query can be:

- Classes that are annotated by typically in the service layer appearing annotations, like @Service;
- Classes that have parent classes or interfaces that belong to the service layer, like "extending controller" classes;
- Classes that contain strings that are typical for service layer, like “service” or “controller”.

This way, a query would reveal all source code that belongs to the different concern slices. Additionally, concern slices that belong to a layer can be offered as a group in the pre-defined concerns view. A developer can just drag and drop a layer query from the pre-defined concerns view without having the need to assemble the query himself. Generally, the query for a layer, by using code structure patterns, avoids building working sets manually.

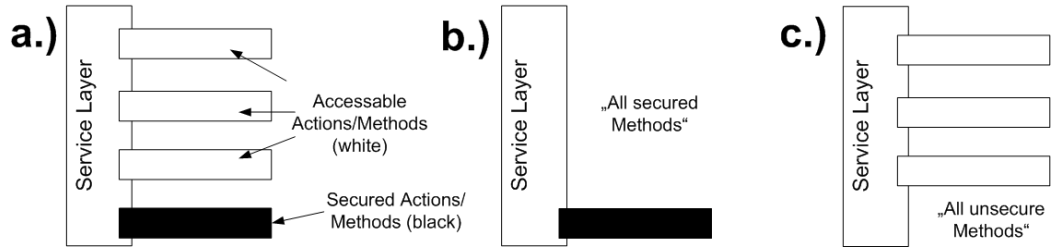
In order to use another tool, developers need to learn the complex query language of that tool, e.g., JQuery [77], first. The Hypermodelling idea of code fragments that belong to multiple concerns is simpler. Furthermore, all the elements that belong to the service layer have to be defined by hand in a query in other tools. The Hypermodelling tool in contrast offers pre-defined concerns and offers the possibility to drag and drop concerns into a query. Lastly, the other query tools present the result of their queries within new views, where our tool enables to work with the package explorer.

10.3.2. Unsecured Methods

Another challenge for developers is to uncover unsecured publicly accessible methods to identify potential security leaks. A developer can compose a query for those with the Hypermodelling tool. For instance, a developer can query for classes that inherit a framework class which is responsible to make the methods of the class publicly available (for instance, a controller superclass). Additionally, he can exclude the secured classes in the query. Excluding secured classes is possible, because applications are normally secured via Aspects or annotations (for instance, @RolesAllowed). Hence, all publicly accessible methods that are not secured can be uncovered easily. With other tools a complex query building process and result representation in standard views is not possible.

We visualize the approach in Figure 10.10. There, we see the service layer that contains various methods (a). One method is secured, the others not. The secured method can be identified with a query (b). Another query for all accessible methods without the ones that are secured reveals c. There, we see all the methods that are accessible from the outside and no security policy is applied and there maybe fixing is needed. Now, developers do not need to search such spots in a program manually any more.

Figure 10.10. Slicing "Unsecured" Method Calls



10.3.3. Deprecated Update

When software evolves, often elements are deprecated. A typical task for developers is to upgrade code that uses deprecated elements into code that does not use deprecated elements any more.

Currently, the warnings and error view lists all elements that consume deprecated elements. A programmer has to click through a whole list to get to the desired elements that need to be upgraded. With the Hypermodelling tool, a developer can add the slice of deprecated consumers to a query. This way, all the methods and classes consuming deprecated elements are revealed.

However, after resolving and inspecting the first deprecated element, a developer recognizes that it is extending a deprecated class. He checks the documentation of the superclass to determine how the code should be updated. In this moment, the developer gets to know that an annotation is used in the new version instead of inheritance. The developer applies the new knowledge and updates the code. In that very moment, the knowledge how to update this kind of deprecated class use is at hand. Hence, the developer has the desire to update all elements of the same kind, to avoid checking the documentation again. It would take quite a while to find the elements with the same problem in the errors and warnings view. Luckily, he uses the Hypermodelling instead and adds just the slice of extenders of the same deprecated class to the query. The result of the query shows all elements using the same deprecated class instantly. Now, the developer can update all classes and add annotations, without having the need to use the errors and warnings view. This is a huge advantage, because right in the moment where the knowledge about an update procedure is at hand, the corresponding code fragments can be easily resolved, what spares time and effort.

As a matter of fact, the developer could also use another code query tool. The main difference to Hypermodelling in this example is that Hypermodelling is built to query for plenty of code and other associated facts, where other tools are mainly created just for code queries. Therefore, the main advantage of Hypermodelling is that concerns like deprecated consumers can be used. Also, any kind of code slices can be combined with the error and warnings concern. Other tools are not that flexible.

10.4. OLAP enabled IDEs

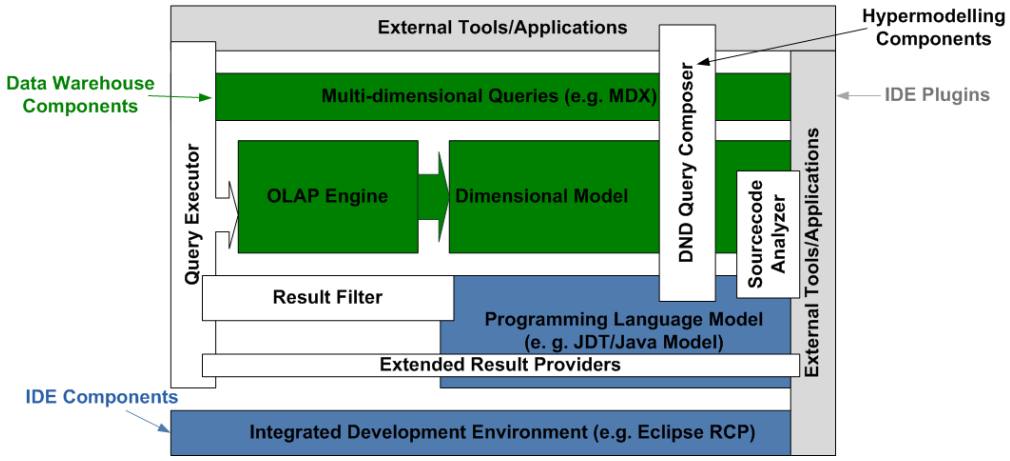
In the sections before, we described a custom IDE plug-in that was following the Hypermodelling approach and showed different evaluative use cases of its application. The IDE version of Hypermodelling is a lightweight and custom implementation that does currently not use Data Warehouse components.

We created a custom solution for the IDE out of several reasons. In general, we consider a full blown Data Warehouse based solution for the IDE as overkill and problematic. The main reasons that we see a Data Warehouse based solution for the IDE as problematic are:

- Files and the code structure within the IDE changes quite often. This would probably mean that we need a full/delta Data Warehouse loading process each time something is changed.
- The often happening changes within the code in the IDE would lead to rapid changing dimensions, what would violate the principles of a good Data Warehouse design.
- A local Data Warehouse installation with a relational Database would be needed, what we see as too heavy just for one application.

Nevertheless, using Data Warehouse components in the IDE would be beneficial out of speed ups and advanced ready-to-use functionalities, like creating reports. Furthermore, Data Warehouse technology offers plenty of speedup mechanisms [152]. Thus, all in all, a customized approach to use Data Warehouse components in the IDE is needed. Hence, in order to advance our current and custom implementation, we propose a lightweight utilization Data Warehouse components in the IDE. We give a schematic overview of this approach in Figure 10.11 and describe it in the following.

Figure 10.11. Future IDE with OLAP Components



We see a schematic arrangement of components to integrate Data Warehouse technology into the IDE. We see that an OLAP Engine, a Dimensional Model and Multi-dimensional Queries are supposed to be a part of future IDEs. Supplemental components, like a Result Filter, a query computer, a Query Executor and Result Providers serve as addition for a future IDE.

In Figure 10.11, we present components of an OLAP enabled IDE. The different colors correspond to different areas of the tools. The white components are the custom components that are needed in addition to the Data Warehouse components. The green elements are Data Warehouse technology and the blue elements are the ones that are part of the IDE. The grey elements visualize the openness

for third party tools and extensions. In the following, we give an overview of the different areas of the graphic.

The IDE Components describe the runtime environment wherein the specialized programming tools are embedded. For the prior case of the Hypermodelling plug-in for Java, we used the Java Development Tools (JDT). Hence, we see that the IDE core as well as the JDT for Java is the foundation to embed further tools in the IDE.

However, modern IDEs are open to integrate new tools on top. The prior referred plug-ins of Eclipse are here visualized as 'External Tools/Applications'. For instance, the AspectJ development tools use and extend the Java Model and contribute new IDE functions. Therefore, the 'External Tools/ Applications' are arranged around all the other components to visualize that there can be interactions.

Anyhow, what is new, compared to current IDEs, is that Data Warehouse Components are embedded. We see MDX Queries, a Dimensional model and an OLAP Engine.

The Dimensional model corresponds to the prior explained cube and dimension schema. It represents the multi-dimensional and computational model (cube) of source code. We arrange it next to the 'External Tools/Applications' to credit that dimensions can refer directly to the data of other plug-ins.

The OLAP Engine utilizes the dimensional model and does it computations on top of it, what we indicate with the directed arrow.[2]

Lastly, we credit that Multi-dimensional Queries can be composed in a Data Warehouse query language (MDX). We show that 'External Tools/Applications' can use this query language, by the arrangement of the box. It is thinkable that not only Hypermodelling uses this query language and additional applications use the multi-dimensional query capabilities.

We tie the arrangement together and integrate Data Warehouse components in the development environment, by introducing additional components. The Sourcecode Analyzer is responsible to parse the source code and to update the contexts of the multi-dimensional model. Therefore, it directly updates the OLAP cubes. This component supersedes the traditional loading process from a relational database. This is important, since the prior mentioned issues of the heavyweight Data Warehouse components, like a relational database, stays limited to a Dimensional Model. Furthermore, the direct cube update overcomes the problem of rapid changing dimensions, since the Sourcecode Analyzer is responsible to keep the model consistent to the current reality in source code. Thus, this spares to credit the time changing dimensions within the IDE, since the multi-dimensional model focuses on the snapshot of current reality. Supplemental, the Source code Analyzer is responsible to integrate Data of External Tools into the dimensional model. Hence, the source code Analyzer is the shortcut to reduce the use of Data Warehouse technology to the pure multi-dimensional units and to overcome the prior named overkill by a complete Data Warehouse. Therefore, we arranged the Sourcecode Analyzer as connector between the Dimensional Model, the Programming Language Model and the External Tools.

In order to create a Multi-dimensional Query, we propose a Drag and Drop Query composer that is similar to our query view. The Query composer allows to Drag and Drop concerns form the various components together and composes and executes the OLAP query internally on demand. Therefore, we arranged it over External Tools and the Programming Language Model to indicate that concerns can be dragged and dropped from everywhere. Internally, the query composer uses the Dimensional model and arranges an MDX Multi dimensional query of the dragged and dropped concerns.

The Query Executor uses the composed query and pushes it into the OLAP Engine. The OLAP Engine uses the Dimensional Model and computes the result. The Query Executer get a handle to this result.

The Result Filter gets the handle for the current result from the Query Executor and allows filtering the view of the IDE. Thereby, it uses the Programming Language Model and the associated tools to filter the IDE for the result.

[2]An lightweight open source OLAP engine, written in Java that can probably be integrated in the IDE is Mondrian. See http://mondrian.pentaho.com/ - 7.11.2012

Lastly, we propose the component of supplemental result providers, what we call "Extended Result providers". Those work similar to the Result filter and get the result handles from the Query Executor. We introduce them to indicate further and different result representations, besides filtering, in the future. Such can be carts or similar things.

10.5. Evaluation of OLAP Code Search

Right before, we proposed an OLAP Engine integration for code search queries in the IDE. Such queries correspond to code structure search. The whole Hypermodelling implementation for the IDE is currently based on a custom implementation. Therefore, one may wonder if Data Warehouse technology can really be applied in the IDE. We credit this doubts and verify that Data Warehouse technology and OLAP queries are suitable for code search in the following. Additionally, we report about the needed effort to create a code search engine implementation.

We evaluate Hypermodelling's suitability as search engine, by creating and testing an implementation as follows:

1. Definition of an parameterized OLAP query to reveal code fragments

   An OLAP query for the complete source code structure is needed, where each query parameter value of the query acts as placeholder variable for code search.
2. A program that composes, executes and presents the results of a query needs to be manufactured.

   Such a program uses input variables and put them into the prior defined query. Furthermore, it is responsible to cut off the empty variables form the query and to execute the query one demand. It is also responsible for handling the result.
3. Let the implementation run and see if it works, by using demo code.

   This test shows then actually if Hypermodelling is suitable as code search engine.

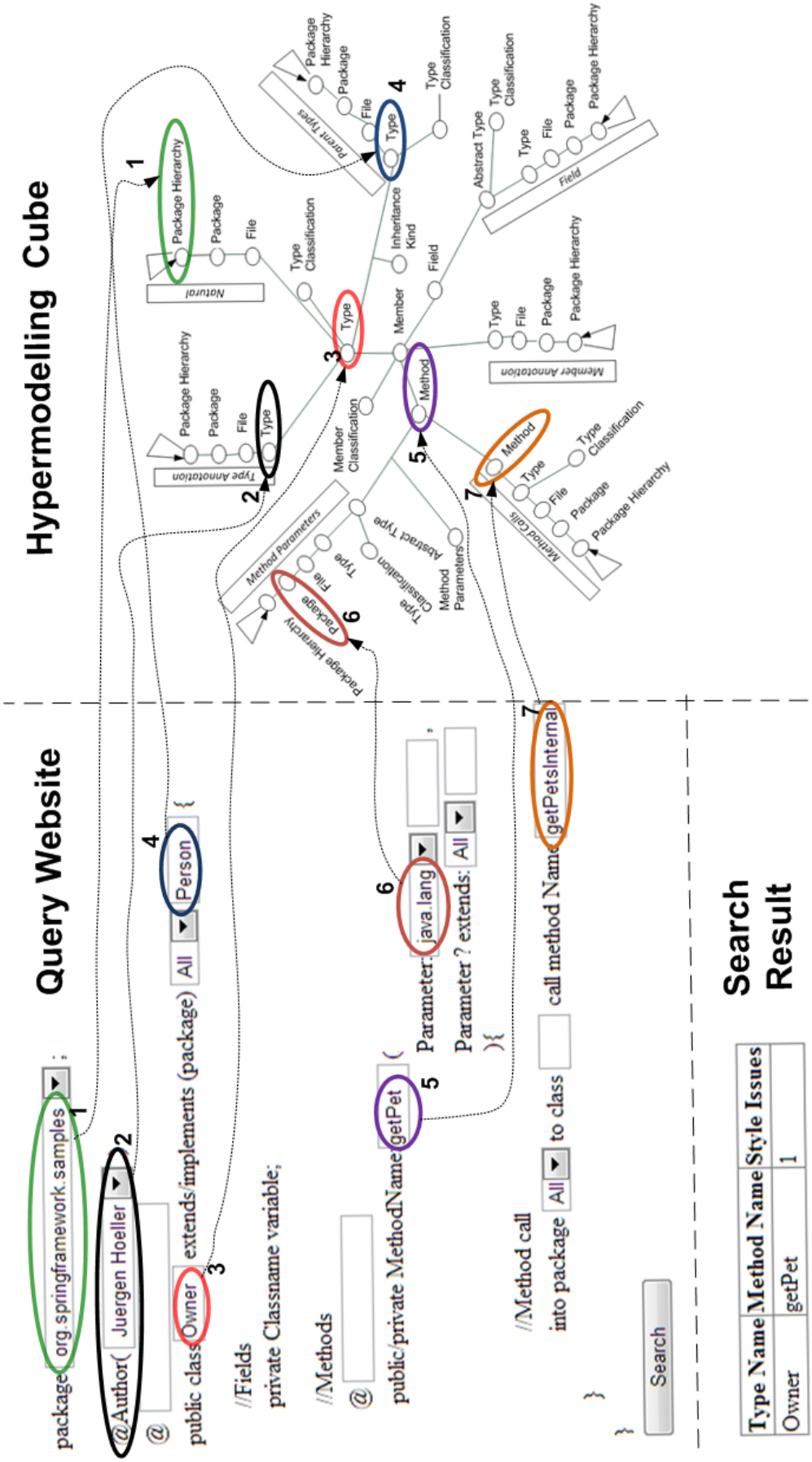
In the following, we report our experiences and the effort to do these steps in more detail:

1.) About half a day was needed two compose an OLAP query for code together with all the parameters. Thereby, we derived a general rule how to concatenate the different parts of the query string for all code elements. The query was tested and resulted in search code elements.

2.) For the second part, a program that composes the query, we created a website that executes the query and presents the result. With bug fixing, about three days were needed to create the user interface. The main reason for that delay of three days was to find out how to use and bind an OLAP server to a website. Also, the configuration of the OLAP server and the http server had to be secured, what takes a while. Therefore, we can justify the effort to implement a search like functionality on top of an OLAP engine as reasonable.

3.) The last part, to run and test the search engine was done easily and resulted in: "It works". In case one may wonder how the engine looks or how we use the OLAP query in special, we provide a screenshot of the search engine in Figure 10.12. There, we show the search engine and the used internal data cube. In the following, we provide further explanations.

Figure 10.12. Code Search based on Hypermodelling



We complement Figure 10.12 with Example 10.1, where we show the searched Owner class that is the result of the search process in the screenshot.

Example 10.1. Example Searched Code

```
/** Simple JavaBean domain object representing an owner.
* @author Juergen Hoeller   */
    public class Owner extends Person {
        -...
        public Pet getPet(String name, boolean ignoreNew) {
            name = name.toLowerCase(-)-;
            for (Iterator it = getPetsInternal(-)-.
                        iterator(-)-; it.hasNext(-)-;-) {
                -...
            -}...
```

Figure 10.12 shows the search engine and sets an additional focus how the search engine interface relates to the multi-dimensional concern model.[3] On the left side, we see the search engine. In general, the search engine user interface is made to look similar to code block structures within the development environment. This visual structure was created to ease comprehension for developers. On the right side, we see the multi-dimensional model. At the bottom, we see the query result. Different nodes in the multi-dimensional model are numbered and equal their numbered counterparts in the search engine. Supplemental, directed arrows show the association. The arrows are directed to indicate that the entered values get searched with the OLAP query. However, only the entered values are linked to the multi-dimensional model, to keep an overview possible. Clearly, all values that can be entered can be associated with the model. In the following, we discuss the input fields of the search engine from top to bottom, briefly.

Right on top, we see the selected and filled field of the package, wherein source code is searched. In our implementation, a drop down menu allows to select the package. Anyway, the input method can be altered into a string box that does auto completion or any other fancy thing.

Right beneath the package, the source codes author can be selected with a drop down menu. Currently, only one Author is supported, but this can be overcome, easily, and extended to multiple authors. Supplemental to the author annotation, another annotation can be specified, what is currently not done.

That following, the class name can be entered as string and the parent type of Person gets specified.

Beneath class fields are arranged. Fields are nodes in the multi-dimensional model and can be integrated in the search engine. However, we do not show this in the current query engine GUI, because we recognized that the "empty to search values" are easier to understand, when we show less options in the query interface.

Following, contained strings of a method name can be specified. Additionally, we see that one method parameter has to come out of the java.lang package. The second method parameter is currently set empty. However, we see that a parameter can also be specified to extend a certain supertype. Lastly, we see that method calls can be specified by the called methods type, package and name.

When a user hits the search button, he gets a result set, like shown at the bottom of the figure. We see also that the result also contains the amount of Style Issues that occur in the method.

Clearly, it is possible to extend such a search engine by restrictions. For instance, a selection box can be used to let the user exclude code with certain Style Issues from a query. We mention this here, because we see further research avenues in this direction.

All in all, this section shows that Hypermodelling and especially OLAP queries are usable for code search. The utilization of Hypermodelling OLAP cubes for code search can be done with

[3]See Section 6.5.2 for details about the multi-dimensional Hypermodelling code cubes.

minimal effort. Thus, the prior IDE vision of integrating Data Warehouse components in the IDE will enable similar search operations therein. Therefore, we see first and strong indications that Data Warehouse components in the IDE can leverage advanced search capabilities. Hence, our future IDE architecture is likely realizable.

10.6. Summary and Conclusions

Source code fragments belong to multiple concerns. The Hypermodelling approach utilizes this and allows Data Warehouse like queries for concerns. We leveraged and transferred the Hypermodelling approach to the IDE. As a result, we presented a tool that enables developers to query code in an OLAP similar manner. The tool takes advantage of the fact that the package explorer is the most used view in Eclipse, by offering a filter for the result representation of a query. Various presented usage scenarios indicate already beneficial applications. Supplemental, we presented software architecture for future IDEs to integrate Data Warehouse technology into them. We verified the suitability of the architecture for the search use case with a Hypermodelling based code search engine.

Altogether, we introduced the approach to use well-known information by a programmer, to enable multi-dimensional queries on source code. Now, the structural information in the code and of frameworks can be used to reveal the code of an application layer. The tool demonstrates that OLAP like queries in the IDE are possible. Now, other researchers can investigate developer productivity increases with the approach.

However, although our presented tool already supports plenty of concerns we have the desire for more. Software tests and their results can be an interesting candidate to be supported for queries. Additionally, it might be interesting to rank the importance of concerns with algorithms similar to CodeRank [191]. A usage scenario can be to suppose the most influential concerns on top in the pre-defined concerns view.

Generally, our method is only supporting clear and well defined concerns. Thus, it should to be investigated how fuzzy concerns can be supported within the tool. For further development of the tool, evaluations and discussions should be carried out, to decide which concerns should be supported in the next version. Additionally, the actual application of Data Warehouses based on our future IDEs architecture should be used in the next revisions of the tool.

The package explorer as the only view for result representation needs to be reconsidered. Definitely, the current application has the advantage of providing the possibility of summoning concerns from other views this way, but only one kind of hierarchy can be shown. Maybe, there should be other result views offered to enable further investigation capabilities with other hierarchies. Likewise, programmers often work with models, when they investigate a program. Therefore, further research is needed to investigate how models can be used to represent query results.

Furthermore, more application scenarios need to be considered to show what is possible with the Hypermodelling approach. For instance, we believe the application of the tool has benefits in the area of composite business applications (mashups) [259]. These wire various business domains together to derive a new functionality. We imagine that these various business domains can be recognized as concerns and used for queries. We believe that investigations in this area can lead to further benefits.

However, the proposed future architecture of IDE indicates there can be further use cases of Hypermodelling within the IDE. In the following chapter, we present such an additional use case.

11. Hypermodelling Live

OLAP for Code Clone Recommendation

"Wrong. Get into trouble. Make mistakes. Fight, love, live."
D'Artagnan's Father. Three Musketeers. 2011

This chapter shares information with:
"T. Frey, V. Köppen. Hypermodelling Live - OLAP for Code Clone Recommendation. Proceedings of Baltic DB & IS 2012. Tenth International Baltic Conference on Databases and Information Systems. CEUR-WS. 2012"[98]

Abstract. Source code repositories contain often millions lines of code. Code recommendation systems ease programming by proposing developers mined and extracted use cases of a code base. Currently, recommender systems are based on hardcoded sets of use cases what makes them inflexible. Another research area is adaptable live detection of code clones. We advance clone detection and code recommender systems by presenting utilization of our Hypermodelling approach to realize an alternative technique. This approach uses Data Warehousing technology that scales for big data and allows for flexible and adaptable queries of source code. We present the generic idea to advance recommendation and clone detection based on queries and to evaluate our application with industry source code. Consequently, recommender systems and clone detection can be customized with flexible queries via Hypermodelling. This enables further research about more complex clone detection and context sensitive code recommendation.

11.1. Introduction

In the prior chapter, we described the general visions to use an OLAP engine within the IDE. In this chapter, we go one step further an describe an additional scenario that uses an OLAP engine for code recommendation and clone search.

Code recommender systems advance integrated development environments. They are based on the idea to extract and mine information from code bases to generate recommendations. Recommendation data contains information, which method calls occur commonly together or which methods of a super class get overwritten [54]. These data is compared to current coding of a developer and proposals are offered. The extraction and mining process limits recommendation to be easily adjusted to specific requirements. For instance, it is desirable to have recommendation information available for diverse APIs and also for an own project. Furthermore, project requirements often differ. In one project, it is required that the recommendation code base just comes out of a specific project and in another setting all available code should be used for recommendation. However, this type of flexible recommendation is currently not available. One main reason for this may be the immense size of modern code bases, resulting in difficulties to adjust the extraction process: It is necessary to scan the code base for different recommendation information every time.

Another challenge is the detection of code clones [223, 263]. Thereby, code bases are scanned for duplicates. One main challenge is the different type of code bases and clone detection methods. Sometimes, a certain package should be excluded because replicas are allowed. Furthermore, clones may be exact duplicates of a code fragment or similar pieces of code. Hence, clone detection faces the challenge to provide an easy adjustable infrastructure that allows detecting different kinds of clones and different code base configurations.

Note, clone detection methods can be configured and live detection is possible [263]. However, it still is a challenge. Thus, we present alternative approach as complementary subsidiary to code recommendation.

Altogether, recommender systems and clone detection face the challenge to provide an adaptable infrastructure for large code bases that allow by flexible means to detect code clones and recommendations. In order to overcome these current limitations, we propose to use our Hypermodelling approach for flexible code recommendation and clone detection. Through this approach both, techniques can be covered with Data Warehouse and OLAP technology [128, 146] that scales well for large data sets and uses easily adjustable queries.

11.1.1. Contribution

The usage of Data Warehouse technology represents a new application scenario for Hypermodelling and the contribution of this chapter is as follows:

- We present the method how to use Hypermodelling for clone detection and code recommendation. In detail, we describe the new application scenario for Hypermodelling to use it for code recommendation and live clone detection.
- The advancement of clone detection and recommendation system through flexible queries
- The application evaluation through an exemplary, step by step, use case

Therefore, our contribution is to describe, how Hypermodelling can be used to advance recommender systems and clone detection at the same time. We also provide an evaluation, in which we demonstrate the application of this chapters method on a real world source code excerpt.

11.1.2. Reading Guide

The remainder of the chapter is structured as follows: First, we describe the general approach, how Hypermodelling can be used to detect clones and recommend code fragments. Afterwards, we describe the approach with a concrete example to evaluate its practicability. Finally, we do a summary and draw conclusions.

11.2. Clone Recommendation with Hypermodelling

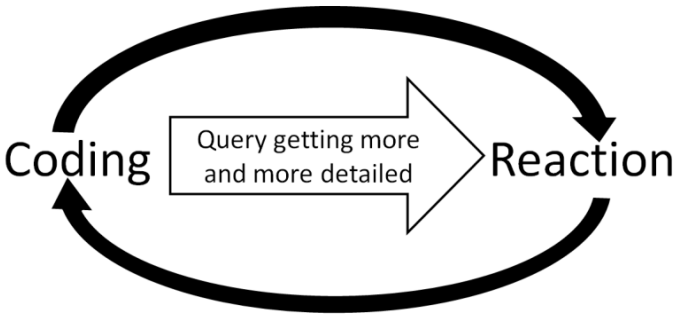
First, we describe the overall idea to use OLAP queries for clone detection and recommendation. Afterwards, we refer to recommendation and clone detection.

11.2.1. Overall Approach

We propose a query based approach to advance recommendation and clone detection. We use OLAP queries and our Data Warehouse approach, out of the size of modern code bases. Additionally, Data Warehouse technology has many best practices and tuning methods at hand. Queries allow flexibility, adaptability, and Data Warehouse technology scales for big data. Thus, our live query approach should help to avoid clones and give recommendations at the same time.

We propose to use Hypermodelling like in Figure 11.1. There, we indicate that a developer encodes functionality (coding) and in the background the written code is used to execute an OLAP query against the Code-Data Warehouse that contains coded structure and elements. For instance, such code can be parameters of methods. Then, the query result gets presented to the developer. This may be in the form of recommendation or in a notification of a code clone. Hence, this query result influences ongoing coding process. From the result, the programmer goes on and encodes more functionality. With this additional functionality, again, a query can be executed that contains more information than the first. Every time, as more and more code exists, the query is more and more detailed. If the programmer finishes his current coding, the query is quite detailed with all code fragments that are belonging to a method. If another method shares enough similarities, the corresponding code is presented to the developer and she has to decide if a code clone exists.

Figure 11.1. Query Refinement Throughout the Coding Process



We see the cycle of a query that is fired in the background while programming. This query is getting more and more refined by continuous entered source code of a developer.

Currently, recommendation is not done live and neither is based on a query approach. Even more, code recommenders are inflexible and do not support slicing and dicing data for specific needs. This is often likewise the same for clone detection. So, with our Hypermodelling approach to load code into a Data Warehouse, every query could be customized to meet specific requirements or just be sliced by a specific viewpoint.

11.2.2. Recommendation

Code recommendation systems mine facts that plenty of programmers have done out of a code base. Imagine, a developer overrides a method. The written method is compared with the recommendation data and she gets proposed which methods were called by others that did an override of this method. Exactly the same information can be revealed with an OLAP query. This information can be presented to a developer to recommend him what others did.

What we describe is a typical application of a code recommender system. Therefore, we propose a to adopt the query process from Figure 11.1 for recommender systems. A developer encodes functionality and queries containing this encoded functionality are done. These queries reveal similar code artifacts and most things that others have done in those are proposed to a developer. The developer continues to encode functionality. This encoded functionality can be added as refinement to the query to get more detailed information on what most others did in a similar situation. With these queries, the recommendation data can be generated live and adapted easily. The Hypermodelling approach ensures that different code bases can be loaded and queried with a Data Warehouse. This allows us to reuse infrastructure that is designed for big data and for dynamic slicing.

11.2.3. Clone Detection

Similarly to recommendation, we also realize live clone detection. Imagine a developer encodes a method, sharing a high similarity or equality to another method. Today, he continues his work. However, in this moment, he has knowledge at hand what he has programmed and can easily compare it with potential clones. This step fits perfectly into workflow and potential clones can be avoided easily. We propose to use our Hypermodelling approach to detect code clones live though a query based approach as described in Section 11.2.2. Through queries the detection process can be easily adjusted to project specific requirements.

For clone detection, imagine a developer is encoding functionality in a method. He creates the method declaration with all method parameters and then he encodes logic into the method. Methods of objects are called and other constructs are realized. Like for recommender systems, regularly queries are executed to determine duplicates. If similarity is too high, code is presented to the developer so that he can decide if he has produced a clone. We show this process in Figure 11.1, that also describes a clone detection cycle.

11.3. Evaluation with an example

In order to evaluate the application ability of recommendation and clone detection based on queries, we select a method of the alfresco project. Data in our Data Warehouse and queries on a real project demonstrate that our approach enables recommendation not only with prepared data. We depict a class (AVMShareMapper) that implements two interfaces that are implemented by other classes of the application. Thereby, we query if others also implement these interfaces to ensure a valid example that shares similarities with other classes. We select the afterpropertiesSet method for investigation and divide the method in four different parts. For every part, we execute queries to simulate how a developer would encode this method and queries would be executed in the background. Our scenario is mainly based on live clone detection. However, the same approach of queries can be used for recommendation.

Figure 11.2 shows the extended coding process that can be supported by queries. We imagine the coding process there as follows: In the first step a developer encodes the class body. Then, he goes on and encodes step by step a method. We split the afterpropertiesSet method and arranged it above, corresponding to the process of a developer. We describe exemplary queries in natural language beneath the process. Those queries can be executed with the Data Warehouse query languages from the development environment in the background, while the developer is coding. Behind queries is their result or at least an excerpt. At the bottom, possible ensuing actions are described. In the following, we go through the process of Figure 11.2:

First (1), a developer starts encoding the class and implements the ShareMapper and the InitializingBean interface. These two interfaces are used to create a query for the most common method names of the classes that implement one of the interfaces.

Note, a query can be sliced by a project or a specific package. Plenty of customizations are possible.

The result shows the amount of children types with the same method name. The result is sorted following the occurrence of method names of ShareMapper implementers. It would be possible to sort the result after the occurrence in InitializingBean or to merge most common method names in both interface implementers. Anyway, the result information can be used to show developers which methods other developers implemented by extending a certain interface. For our scenario, we imagine that developers see plenty of times the afterPropertiesSet method is implemented and start encoding this. Thereby, it can be recognized that a developer uses a method name based on the InitializingBean and the following queries can be specialized on this interface. To give a better impression about the technique, we present the query in Example 11.1. The parent class or interface is named ParentType and the CodeStructure is the OLAP cube. The query is based on multi-dimensional expressions standard [1] and shows that the amount of methods for children is computed for extenders of the ShareMapper and InitializingBean.

Example 11.1. Query for method names of interfaces children

```
SELECT { [ParentType].[Name].&[ShareMapper],
         [Parent].[Name].&[InitializingBean] -} ON COLUMNS -,
       { [Method].[Name].[All].CHILDREN -} ON ROWS
FROM [Code-Structure]  WHERE ( [Measures].[Method-Count] -)
```

In the second step (2), the developer starts encoding logic of the afterPropertiesSet method. He calls a method of what is used to refine the former query. It is enriched with information which methods are called to reveal which types also obtain same methods, implement the interface and have the

[1]For further information about the MDX language, see Section 4.4.

same method name. Such kind of similarity can be an indicator that the developer produces a clone. Furthermore, maybe the developer is putting effort and thoughts into implementing a method that is already implemented. Therefore, a developer gets presented other methods that share a high similarity. If a user rejects the proposals and wants to encode further functionality, the information about similar methods can also be used to present methods that are called in the similars. We show the corresponding query in Example 11.2. The result types of this query that share a similarity (MultiTenantShareMapper, HomeShareMapper) can also be used in another query to generate recommendation information. Exemplarily, such a recommender query is shown in Example 11.3. The called methods of two similar types (MultiTenantShareMapper, HomeShareMapper) are computed based on a query. This can be used to propose a user which other methods are called by other developers in a similar situation.

Example 11.2. Determining similar code

```
SELECT { [CalledMethod].[Name].&[getConfigSection]} ON COLUMNS,
       {[Type].[Name].[All].CHILDREN} ON ROWS
FROM [Code-Structure]
WHERE ( [Measures].[Method Calls],
        [ParentType].[Name].&[InitializingBean],
        [Method].[Name].&[afterPropertiesSet] -)
```

Example 11.3. Determining what similar code did

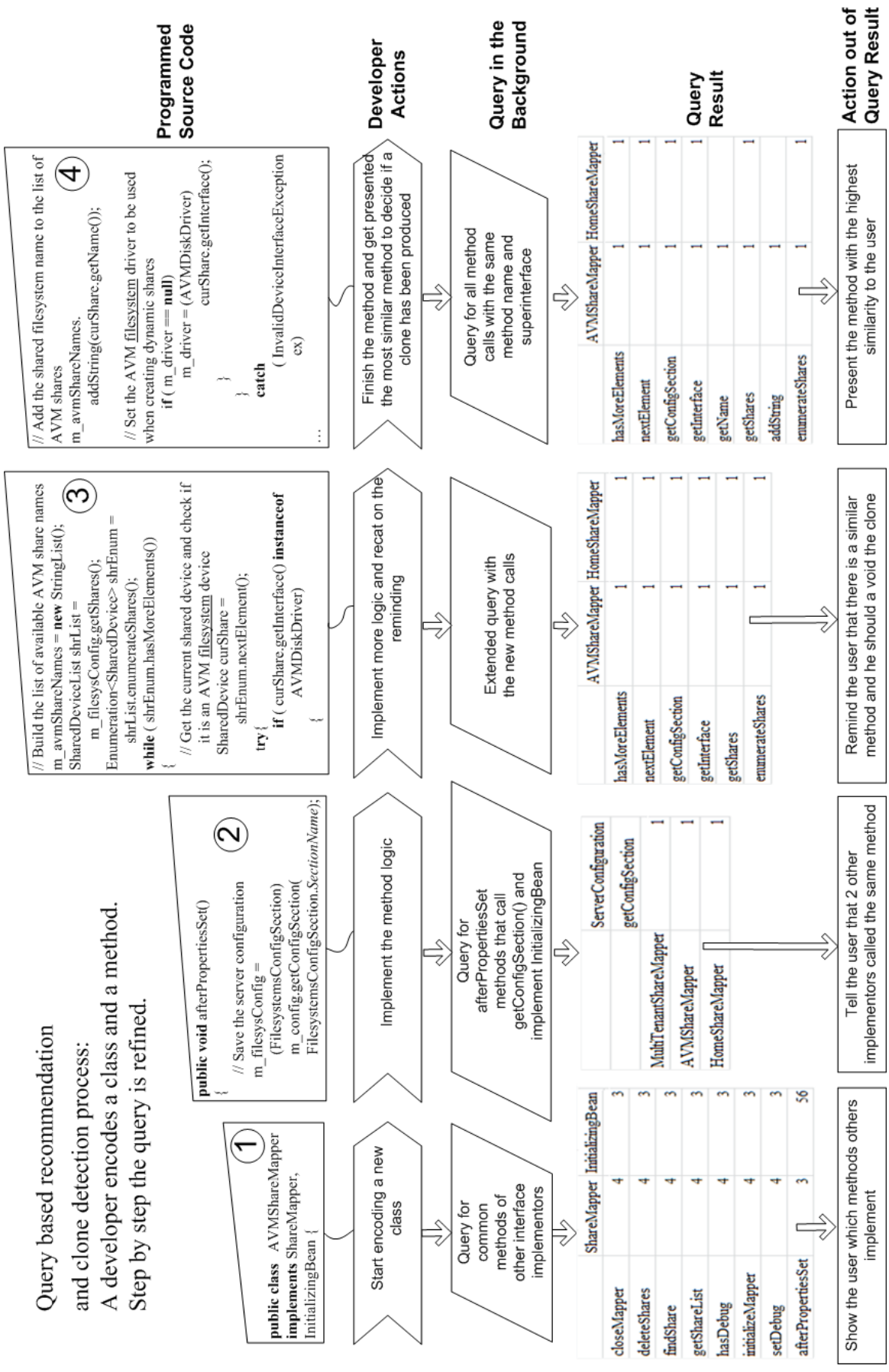
```
SELECT NON EMPTY { [Measures].[Method Calls  Count] -} ON COLUMNS -,
                 { [Called Method].[Name].[All].CHILDREN -} ON ROWS
FROM [Code-Structure]
WHERE ( [Type].[Name].&[HomeShareMapper],
        [Parent -- Type].[Name].&[InitializingBean],
        [Method].[Name].&[afterPropertiesSet] -)
```

Succeeding (3), a developer encodes more functionality. In the background the query is extended to compare method calls of the current method with the ones of other extenders. It is like a fail save detector that tries to uncover if there is a clone produced. With every new method call, the query is extended and executed again. After six shared method calls and the super interface, enough indicators are collected to remind the user with a traffic light or a pop up about similarity.

Note, at this moment such kind of queries can also be used to propose method calls, like in step 2.

Finally, in the forth step (4), the user finishes the method. This is recognized in the background and a complete query for all method calls is executed. Out of the high similarity of the method calls, it is assumed that the method logic is maybe “externalizable” and customizable through parameters. However, the easiest way is to present methods with a high similarity to the user and let him decide if this is a clone.

Figure 11.2. Query based Recommendation and Clone Detection Process



11.4. Summary and Conclusions

We described the problem of large code bases and the inflexibility of current code recommendation and challenges of clone detection systems. We proposed to overcome these limitations through a query based approach that uses Data Warehouse technology. Our approach is evaluated and its application ability is shown by queries to a real code base. So, the next generation recommendation and clone detection techniques can be based on Data Warehouse technology.

In general, we see the need to describe our method in more details. Additionally, the use of Hypermodelling technology enables further research on Data Warehouse based clone detection. We showed the capability to find “full method clones”. However, clones can also be copied code fragments within different methods. Therefore, further investigations can focus on identification of queries for these clones. Thereby, we see ways to compute similarity indicators based on queries and the clone granularity level (method or fragment based) as important questions. The precision of our technique is fixed to method calls, discarding method parameters. Hence, the same called methods that take different parameters and are considered as clones. For that reason, the possibility to enhance or adjust the precision of our queries through additional facts should be considered. Therefore, we see the need to work together with industry developers to evaluate which level of precision and granularity is desired in practice.

Furthermore, we connect code recommendation with clone detection. Therefore, our work makes it possible to regard both areas together and investigate possible synergies in the future. This connection is an additional difference to previous research. Generally, our approach shows a rudimentary and first scenario with primitive queries. More complex queries, scenarios, and areas are of interest for further investigations. For instance, the area of refactoring is also near clone detection and relations to it can be investigated. We also see an advanced trail by integrating the context (e.g., the package, the prior studied code, or the task) wherein a developer encodes functionality to advance recommendation systems. Currently, other code recommenders [2] are dull and based on the same rule set. Our dynamic query based approach enables further research, how recommendations need to be altered and adjusted to different contexts of a developer to respect his current programming tasks within the recommendation.

However, we close the application scenarios of Hypermodelling right here and leave additional applications for future research.

Part IV. Related Work and Conclusion

12. Related Work and Discussion

> *"Always be willing to fight for what you believe in. It doesn't matter if a thousand people agree with you or one person agrees with you. It doesn't matter if you stand completely alone. Fight for what you believe."*
>
> *President John Sheridan. Babylon 5. Season 5 Episode 21: Objects at Rest. 1998*

Abstract. One may wonder: Is Hypermodelling really new? How does it differ from the motivating techniques? Are there potential synergies with related work existing? In this chapter, we provide a comparison to related work and refer to work from related areas. We show the potential synergies exist and future work needs to inspects those further. Additionally, we give a discussion and reveal a few down sides of Hypermodelling and point out in which cases they matter.

12.1. Introduction

Hypermodelling contributes to diverse areas in software engineering. The prior chapters contained information about different application scenarios. As a matter of fact, the application scenarios of Hypermodelling cover a huge area and it is a challenge to gain a holistic picture of all related work. Furthermore, crediting related work for the individual scenarios does not cover all approaches that can be considered as related.

Additionally, like every piece of information technology, there are down sides of Hypermodelling. Those need to be raised, discussed and credited in special. For instance, the areas of motivating work to derive Hypermodelling need to be compared to it. Therefore, we provide this chapter and compare Hypermodelling to related work. Supplemental, we discuss the known down sides of Hypermodelling.

12.1.1. Contribution

The contribution of this chapter is to study Hypermodelling in comparison with related work. We use informed arguments and illuminate the down sides and the advantages of Hypermodelling in comparison to related work. The contribution of this chapter is threefold:

- We present a comparison to motivating work and show that no motivating approach credits all the contributions of Hypermodelling at once.
- We describe approaches from other areas and point out the differences and potential synergies with Hypermodelling.
- We provide a discussion about the downsides of Hypermodelling.

All in all, this chapter underlines that Hypermodelling is a beneficial contribution.

12.1.2. Reading guide

First, we compare Hypermodelling to its motivating research. Then, we present further related work to Hypermodelling. Afterwards, we discuss the downsides of Hypermodelling and give, finally, a brief summary about the chapter.

12.2. Comparison to Motivating Work

Hypermodelling was derived by regarding different contributions to the area of software engineering. In the following, we first give a comparison about the main differences to related work. Then, we present a table with different capabilities of Hypermodelling and discuss how the motivating techniques differ in those points. This underlines that Hypermodelling is a step forward compared to its motivating work.

12.2.1. Motivating work

Hypermodelling was derived by crediting contributions of prior art. In the following, we briefly recap the corresponding techniques and compare them to Hypermodelling. Then, we give an overview of the main differences in a wrap up table.

Multi-dimensional programming approaches

Multi-dimensional programming approaches target to develop new ways of concern encapsulation. The multi-dimensional approaches share that they see the concern space of a program as a multi-dimensional space, where the concerns influence each other. In order to advance the object-oriented programming, these approaches introduce new means of modularization and composition for a better suitable concern separation [87].

In contrast to this, Hypermodelling is as additional tool for programming and no language extension. Therefore, it has the advantage that it is not a programming paradigm and it can be used together with commonly applied programming languages and no extension of those is needed. However, aside the obvious differences, like reporting tools and the different application scenarios, the Hypermodelling paradigm differs in certain aspects from the multi-dimensional separation of concerns. Hypermodelling offers computation directives for concern associations and the possibility to aggregate those. This is a huge difference, because multi-dimensional paradigms have no focus in abstractions and to express the module interweaving in a multi-dimensional model. However, Hypermodelling is also a multi-dimensional approach and future research should investigate further synergies between both areas.

Virtual separation of concerns

Virtual separation of concerns focuses on the possibility to do a concern centric projection of source code. Normal object-oriented source code and compiler meta data is analyzed and used to reveal concerns in source code. This information is then used to present non separated concerns in source code separated in the IDE [135]. Complementary tools advance the approach and allow zooming into the concern (feature) hierarchy with different abstraction mechanisms [243].

Hypermodelling differs, because its main purpose is to provide a multi-dimensional model of the source code structure and associated artifacts. In fact, the multi-dimensional model of Hypermodelling can be compared with the internal concern model of virtual separation of concerns. Even though, the models still differ in their very main meaning, because Hypermodelling consists of a computational model as foundation to compute aggregates and perspectives. In contrast to this, virtual separation of concerns does not target Data Warehouse computation models. Additionally, Hypermodelling enables plenty of applications, where virtual separation of concerns merely focuses on the development environment. Reporting, recommendation and drag and drop of concern queries are not possible with virtual separation of concerns.

Besides all this differences, we see the need to investigate how and if the virtual separation of concerns feature concern model can be realized with Hypermodelling. We assume that there are synergies between these two approaches and further research needs to investigate how Hypermodelling can be applied to software product lines.

Orthographic Software Modeling

Orthographic Software Modeling (OSM) is a multi-dimensional viewpoint and software navigation approach. It aims to adapt the ideas from computer aided manufacturing (CAD), where the different models are arranged orthogonal to another and have little overlapping data. In order to keep the different models in CAD consistent, CAD uses a central model as foundation for different projections. Logically, one goal of Orthographic Software Modeling is to develop a holistic model central model for software systems. This holistic model is then used to do real time transformations into specific domain specific projections, like it is done with CAD models [39].

The main difference of OSM and Hypermodelling is the kind of the used model. The internal Hypermodelling model defines and allows computations and OSM focuses purely on a data centric

model. In contrast to Hypermodelling, neither queries for reporting nor code analysis are part of the OSM technique. Hence, OSM targets a fixed set of models, where Hypermodelling tries to introduce flexibility with queries. Additionally, Hypermodelling uses a code model as central point to allow the association with different quality indicators and management data like developers. OSM does not target to integrate this data.

Generally, OSM targets models where Hypermodelling focuses mainly on source code and the associated elements. Therefore, we see the approaches neither as competitive but rather as complementary. Future research needs to reveal potential synergies with OSM and how and if model data can be integrated with the Hypermodelling approach.

Source code Mining infrastructures

Another motivation to create Hypermodelling is source code mining. An exemplary source code search infrastructure [42] is based on a relational model with four tables. The relations within code are realized through a relational table, connecting different rows of an entity table with each other. An entity represents a class method or a package. Hypermodelling uses in contrast various tables to represent the diverse relations that occur in source code. This has the advantage that when the relational model of Hypermodelling is queried no self joins of the relations are need like it is the case for the code search infrastructure. Additionally, the main difference is that the whole Data Warehouse toolbox can be used with Hypermodelling. Therefore, Hypermodelling comes with reports and application scenarios that are not part of prior mining infrastructures.

Additionally, researchers mention Data Warehouse techniques as a potential enabler for advanced source code mining applications. Software Intelligence (SI) [118] describes this idea on an abstract level. In spite of the relation to SI, Hypermodelling is still unique. SI neither proposes multi-dimensional models nor takes the fact into account that concerns play a central role in software development. Neither are our application scenarios mentioned. Therefore, SI, can just be seen as a research proposal that we see partly fulfilled with this work.

Software Project Cockpits

Storing source code in databases [113] describes code queries with a logic programming language. The source code is stored in a relational database and queries in the logic language are translated into SQL queries. This can be seen related because relational databases can be queried, too. Hypermodelling uses Data Warehouse technology to do the query and the relational model is just used to be the source for the cubes. Therefore, it was very efficiently possible to create a report for the software dependency, because a whole Data Warehouse toolbox could be used. With normal databases the report would have had to be programmed by hand.

Soft-Pits focus on software cockpits for issue tracking systems [184, 64]. Hypermodelling differs, since it is not only based on data out of issue tracking systems and current cockpits ignore the multi-dimensional program structure. Logically, code cockpits research presents no multi-dimensional models for source code. Likewise, most of our application scenarios differ to the code cockpit based ones. Additionally, Hypermodelling advances code cockpits by code structure investigations. Furthermore, the cockpits and reports in this work enable a direct viewpoint on code structure what is not possible with prior cockpits.

Therefore, we see the current code cockpits about issue tracking data as complementing element to the other use cases of Hypermodelling. Within our work, no data of issue tracking systems was loaded into a Data Warehouse and associated with the code structure. Therefore, future research can focus how best merge the research results of soft-pits and Hypermodelling.

12.2.2. Differences, Similarities, potential Synergies

All together, we show the main differences of Hypermodelling to prior and motivating work in an overview in Table 12.1. In the following subsections, we describe each row step by step and refer to potential future work by synergies.

Table 12.1. Comparison to Motivating Work

Support / Approach	Hypermodelling	Multi-dimensional programming approaches	Virtual separation of concerns	Orthographic Software Modeling	Source Code Mining Infrastructures	Software Project Cockpits
Big Data Code Bases	Yes, Data Warehouse technology	-	-	-	Partly, relational self referencing tables not perfectly suitable	Indirect, usage of Data Warehouse technology, no source code integration
Different Viewpoints	Yes, Different data cubes, multi-dimensional navigation, queries, dynamic	Yes, through modules, fixed	Partly, through a feature model	Yes, different models, fixed	Yes, Through queries	-
Respect concern interweaving	Yes, Composition - Fact - mapping	Yes, separation and weaving through new modules and composition techniques	Yes, View based separation	Yes, hidden, Central composition model	-	-
Multi-dimensional concern space	Yes	Yes	Partly, feature hierarchy	Yes, different models, Multi-dimensional navigation	-	-
Respect usage of frameworks	Yes, possible within queries	Yes, encapsulation though new weaving techniques	-	-	Yes, API usage Mining	-
Abstraction	Yes, hierarchies and relations can be aggregated	-	Yes, Concern/Feature Hierarchy Zoom	Yes, through models	-	-
Analysis/Integration/ Mining	Analysis with OLAP queries, possible Data Warehouse mining extensions	-	Limited, feature analysis through parsing	-	Basic, used for extraction and apply mining afterwards	Partly, a few pre- defined cockpits
Cockpits	Toolset of Data Warehouse technology can be used to create them	-	-	-	Theoretically, could be done with coding effort and queries	Yes, but limited data is available
Main difference to Hypermodelling	-	No abstraction, no analysis	No analysis, no queries	No analysis, no queries	No multi-dimensional models, data integration not addressed	No source code structure analysis

Big Data Code Bases

We see that merely Hypermodelling focuses explicitly on the current big data size of code bases. Source Code Mining Infrastructures target large scale code bases by the internal usage of a relational database. However, in the current implementation the relational table design is self referencing what results in self joins in queries what slows the computation of their result down. Out of this, we refer to the current infrastructures as only partly supporting big data code bases. Software Project Cockpits, support big data indirectly by using Data Warehouse technology. Still, no source code is part of the current code cockpits.

Different Viewpoints

Hypermodelling respects different viewpoints towards source code. It supports different data cubes, multi-dimensional navigation and the execution dynamic queries. Thus, we refer to Hypermodelling as a projection based technique to enable different viewpoints. Therefore, the support for different viewpoints is realized in a complete other way then in multi-dimensional programming paradigms. There, modules are used to enable different viewpoints. Logically, this makes their viewpoints rather fixed in comparison to Hypermodelling, where concerns do not need to be realized in their own physical modules.

This non physical concern encapsulation shares a similarity to virtual separation of concerns. However, the feature concern model of virtual separation of concerns is mainly hierarchic and does not serve as origin for flexible as queries. Out of this we mark virtual separation of concerns only as partly supporting multiple viewpoints.

Orthographic software modeling is rather similar to Hypermodelling in this point and supports multiple views. Since Hypermodelling is query based, we consider OSM more fixed technique with predefined view angles.

Lastly, we can imagine that source code mining infrastructures support multiple viewpoints, when we credit queries and assume that query execution on a repository results in a viewpoint.

Respect Concern Interweaving

Hypermodelling respects tangling and scattering by using its composition-fact mapping. Thereby, each composition is realized as a fact within the data cubes. This enables to query for the different fragments that belong to a concern, even if those are intertwined within modules. Other researchers use separation through new kinds of modules. This differs from Hypermodelling and future research needs to focus how those approaches can be combined.

More similar to Hypermodelling is again virtual separation of concerns that targets also a view based separation. OSM uses in contrast to Hypermodelling another approach and tries to hide composition complexity within a central model.

Multi-dimensional Concern Space

Natively, Hypermodelling supports the multi-dimensionality of the concern space through the usage of the multi-dimensional Data Warehouse data structures. Likewise, the multi-dimensionality of the concern space is by multi-dimensional programming paradigms addressed since those were especially developed for this structure. Virtual separation of concerns differs here a bit since the feature model within the approach is mostly hierarchic what probably origins the limitations within the presentation in the traditional views of the development environment. But still, virtual separation of concerns also addresses the multi-dimensional concern space, partly. Again, OSM is near to Hypermodelling, since it offers multi-dimensional navigation and models for the different viewpoints.

Respect usage of Frameworks

The reality that frameworks are commonly used within software development is respected in Hypermodelling with the capability that source code artifacts of a framework can be part of a

query. Multi-dimensional programming approaches take another approach to advance the usage of frameworks: They allow weaving functionality of a framework in a program by using new composition and modularization techniques. Furthermore, some source code miners also investigate the API usage of frameworks [54], what we credit in the table.

Abstraction

Nevertheless, one key point in deriving Hypermodelling was to provide new kinds of abstraction for large scale code bases. Hypermodelling focuses hereby mainly on a indicator based approach that allows the aggregation of hierarchies. Similar to this, virtual separation of concerns offers the non indicator based feature/concern hierarchy that can be browsed and zoomed in. This differs mainly in the point that Hypermodelling allows to use multiple different hierarchies simultaneously and offers indicator aggregations on top. Therefore, future research can reveal how the feature hierarchies can be extended with the Hypermodelling approach. However, again it is notably to mention the OSM approach that targets abstraction also, but uses another way to achieve this. Here, the abstraction is model based, whereby different models can correlate to different levels of aggregation. Hypermodelling uses computational abstraction. Again, we see the need to study potential synergies in future research.

Analysis/Integration/Mining

Hypermodelling leverages the usage of the whole Data Warehouse infrastructure with an OLAP processor, visualization capabilities and reporting tools. Furthermore, data mining suites can be "plugged" to the data cubes in the Warehouse. In contrast to those advanced capabilities, the feature determination of virtual separation of concerns seems rather limited. Here, parsing determines which code fragments belong to which feature. This is not comparable to OLAP or data mining algorithms. However, the feature determination of virtual separation of concerns should be investigated further. Hypermodelling gives guidelines howto associate further code classifications with code structure. Virtual separation of concerns feature associations can be realized as such classifications. Hence, possibly, the feature determination of virtual separation of concerns can be integrated into Hypermodelling.

Furthermore, traditional source code mining approaches offer the foundation to apply mining. They focus on the extraction of information with SQL queries. This extracted information is then used to apply mining algorithms. Hence, the main difference to Hypermodelling is that Hypermodelling offers a complete OLAP engine and mining can be applied in the Warehouse itself, what spares the extraction step. This way, we see the potential to apply and integrate already done mining scenarios with less effort in Hypermodelling. Further research needs to reveal more insights about the specific scenarios.

Lastly, we classify prior developed cockpit approaches as partly supporting analysis, because they support the direct analysis of one specific field of interest.

Cockpits

A common usage scenario of Data Warehouses is to create dashboards. Hypermodelling has the whole tools to create those at hand. This results in reasonable effort to built code cockpits for different parts of the code structure. Theoretically, since source code mining infrastructures are query based, it is possible to create cockpits "by hand". However, it is worthwhile mentioning that the needed effort would exceed the one needed with Hypermodelling, severe. Data Warehouses offer plenty of support here that would all be lost by creating cockpits manually.

Since cockpits already exist for issue tracking systems we credit the usage of Data Warehouse technology. The main differences in the already existing approaches is, like mentioned before, the limitation that current cockpit approaches do not represent the code structure. Hence, it cannot be queried and used within a cockpit. Future research can extend code cockpits with the Hypermodelling code Data Warehouse model.

Conclusion

All together, multi-dimensional programming approaches differ mainly out of their main purpose to provide advanced concern encapsulation to Hypermodelling. Therefore, they provide no abstraction mechanism in the way Hypermodelling does. Furthermore, they are not built for analysis of code. This is logical, since it is not the purpose of programming paradigms to provide analysis. Hence, we future research can reveal possible synergies of code analysis with Hypermodelling and multi-dimensional programming approaches.

Similarly, virtual separation of concerns follows different goals than Hypermodelling. Therefore, the analysis methods of the approach are limited. Likewise is this the same for OSM. In OSM no query based approach is supported. For both approaches, future research can reveal how to combine the approaches with Hypermodelling.

However, source code mining infrastructures do not address integration capabilities of the Data Warehouse tool chain. Current issue tracking cockpits do not address source code structure investigations. Hypermodelling addresses the integration of both areas. Therefore, we see Hypermodelling as key technique to consolidate those two areas in future research together.

Finally, this comparison shows that related work targets partly the same areas like Hypermodelling, but no technique is equal nor does it address all areas like Hypermodelling does. Hypermodelling is a compared to prior research a holistic approach and future research needs to reveal further synergies with the motivating techniques.

12.3. Further related techniques

Before, we described the main differences to motivating research. Now, we describe related areas to Hypermodelling and its application scenarios. Thereby, we point out the main differences and advantages of Hypermodelling. We divide related research into logical subsections and discuss therein the corresponding technologies.

12.3.1. Source code Query tools

Storing source code in databases [113] describes code queries with a logic programming language. The source code is stored in a relational database and queries in the logic language are translated into SQL queries. Hypermodelling uses Data Warehouse technology to do the query and the relational model is just used as data source for the cubes. Hence, Hypermodelling has the advantage that aggregations of the data in the cubes can be done. We believe that this aggregations can be used to speed up the approach with the logic language. Future research needs to investigate this in more detail.

Semmle code[1] is a source analysis and investigation tool. It offers a query interface to investigate source code and its associated elements within the development environment. It supports queries and result presentations like heat maps and traditional pie charts. Similar to this are code structure cockpits with Hypermodelling. However, our cockpits differ from semmle, because they are based on the Data Warehouse technology and focus on the efficient investigation of certain use cases. Semmle, on the contrary, offers multiple possible representations for queries, but not the possibility to create a cockpit for a specific use case. Furthermore, Hypermodelling for the IDE differs from semmle and allows to compose queries with drag and drop. In semmle, developers need to learn a query language. Hypermodelling uses a well-kown view for result representation, where semmle uses a special result view. Finally, the explicit multi-dimensionality of source code is respected by Hypermodelling as first class citizen. Semmle is just an ordinary query tool and does not target investigations of the multi-dimensional concern space.

ConcernMapper is another IDE query tool for manually associating source code with concerns [219]. The associated concerns can be used to filter the development environment to focus on elements belonging to the selected concerns. Hypermodelling, on the contrary, has persuasive query

[1] http://semmle.com - 7.11.2012

capabilities that unleash queries with combinations of concerns far beyond the simple filtering. It can be interesting to extend the Hypermodelling tool with the capability to add associated concerns of ConcernMapper to a query.

Flat^3 [227] offers the possibility to search for code fragments, in the IDE, via text strings and presents the search results in a result view. Also, concerns can be discovered via the execution of a program. Flat^3 offers rudimentary visualizations. Compared to Hypermodelling the multi-dimensional concern structure is ignored and just one concern kind is supported. Again, the detected concerns could be used to extend the Hypermodelling tool and be used within queries.

Mylyn is a tool that can be used to track down elements that are relevant for a task [139]. Hypermodelling for the IDE, is in contrast to Mylyn not a tool to track down concerns automatically, but tasks of Mylyn can be used within the queries. Mylyn itself lacks the possibility to calculate elements that belong to one task and not to another. Through the support of Hypermodelling this limitation can be overcome. Now it is possible to compute elements that belong to one and (not) another task or to combine queries for tasks with other concerns.

JQuery is a source code query tool with a predicate logic language [21, 166, 77, 167, 74]. The main issue is the complexity and the custom language to do the queries. Additionally, no result filter for the known views is offered. With Hypermodelling for the IDE, developers have a tool that is integrating into their well known views. There is no necessity to learn a new complex query language and new views. However, detailed investigations can possibly reveal synergies of the approaches in the future.

Ferret is a similar tool to JQuery that allows building queries within the IDE [76, 75]. Ferret adds a new view to the IDE and targets mainly to investigate a program and its associated elements in a query result view. In contrast to other query tools, Ferret supports code associated artifacts in queries, like results of the eclipse testing and performance tools. Therefore, it can be compared with the integrative approach of Hypermodelling. However, Hypermodelling differs, because it uses Data Warehouse technology and introduces filtering and additional use cases like code analysis, code cockpits code structure planning and aggregation. Nevertheless, the research about Ferret reveals which conceptual queries are of interest for developers. Future research can investigate these queries and determine which additional data can be of benefit to be integrated with Hypermodelling and if there are further potential synergies between the approaches.

The **Concern Modeling Environment (CME)** [63] allows multiple concerns to be associated with a code fragment. CME offers a query interface to query the concern associations and program structures and show them in a result view. The main difference to Hypermodelling is that no inspiration from Data Warehouse technology is taken. Additionally, the current result representation in the package explorer and the drag and drop capabilities make it simpler to compose queries in Hypermodelling and inspect their results.

The **AspectJ Development Tools (AJDT)** offer support for programming with Aspects [24, , 59, 27, 69]. In contrast to Hypermodelling, AJDT is based on the dominant viewpoint of Aspects. Our presented Hypermodelling tool supports various kinds of concerns like annotations and unsafe type casts. In our approach Aspects are just representing one concern. Hypermodelling can be used to extend the concern kinds of ADJT and be used to treat all concerns equally by a query based approach. Thereby, would it be interesting to adapt the Aspect visualizing view to show not only show Aspects and their files or packages together, but to use other concerns, like annotations or Beans, and see the Aspect distribution for those.

NDepend and JavaDepend are source code investigation tools for the integrated development environment [30, 31]. They are providing a query language to query for source code structure and associated metrics. The outcomes of queries are the source code artifacts that correspond to the query. For instance, queries can specify things like all classes that have more than 30 lines of code, having test coverage and that extend a certain super class. Then, the results get shown in a result view. Also, different visualizations to investigate dependencies are provided. First, there is a dependency graph. Secondly, there is a dependency matrix and lastly, there is a heat map like view offered for visualization. Furthermore, metrics, visualizations and queries can be generated

together into a static report file. Hypermodelling on the contrary uses Data Warehouse technology and is based on the explicit idea of a multi-dimensional classification of a fragment. This can be seen especially in the navigation with Hypermodelling where the whole multi-dimensional structure of code can be used for browsing and aggregations for the different viewpoints can be done. NDepend shows the results of queries in the normal structure of projects and packages down to classes and their methods, where Hypermodelling can use different viewpoints. For instance, it is possible to in Hypermodelling to revert and alter the hierarchy and use the inheritance hierarchy of types as result order. NDepend is here just a normal query tool that supports various views, but no real multi-dimensional viewpoint. Furthermore, Hypermodelling can be used to create code dashboards that are interactive and not static like it is the case in NDepend. Additionally, Hypermodelling for the Eclipse IDE allows drag and drop capabilities and to present the results of a query with a filter, where in with NDepend a new view has to be learned.

Supplemental to all the prior approaches, Hypermodelling utilizes the whole Data Warehouse tool set and we presented advanced applications like reporting, recommendation and the usage in the development environment, what is not credited by prior art.

12.3.2. Code Search

In the following, we credit different code search and analysis techniques, similar to Hypermodelling.

Traditional Code Search Engines

Well known, and similar to normal text based search engines, are code search engines. A popular code search engine is **koders.com**.[2] With this engine developers can search for strings in source code files. Those strings can be specified as necessary to appear in a document, concatenated with an logical 'and' or used as exclusion criteria. Furthermore, the stings can be specified to appear in method, class and interface definitions.

Krugle[3] is a source code search engine similar to koders.com, but more search functionality is supported. Whole code snippets can be searched for their occurrence. Likewise, the search for method definitions and method calls is possible. As addition to this, a fuzzy precision be activated set to reveal source code that corresponds to the searches only partly.

Similarly, **JExamples**[4] offers a code search engine where class names, function invocations and supertypes can be specified. Additionally, one complexity metric can be specified in queries to determine only code that corresponds to this metric.

Codase[5]offers supplemental query capabilities compared to the prior mentioned search engines. There, code can be entered to search for program structures that equal the entered code.

In general, all of those search engines follow the goal to reveal text in documents and work similar to ordinary search engines. Therefore, the capabilities to query for the code structure are limited, since every structural element is realized as an extension of the normal word index. Furthermore, only one search engine supports to use metrics as additional indicators for code search.

Hypermodelling differs from the mentioned code search engines in the internal usage of a multi-dimensional model that realizes all the connections in code. The code search engines are not, or at least not known, based on an internal multi-dimensional computation model. Therefore, the results that come up in a code search process cannot be ranked dynamically by indicators. For instance, a query in Hypermodelling for methods can be ranked by the amount of invokes of a method. The discussed search engines do not offer such dynamic ranking capabilities.

Additionally, Hypermodelling can use code associated information, like style issues in queries. This is a huge advantage over the current search engines, because when code is searched not only revealing fitting elements is important, it is equally important to reuse high quality code. The current

[2]http://koders.com - API description - 7.11.1012 http://corp.koders.com/support/getting-started#search - 7.11.1012
[3]http://www.krugle.org/document/advanced_search/ - 7.11.1012
[4]http://jexamples.com/ - 7.11.1012
[5]http://www.codase.com/ - 7.11.1012

engines do not address this need. They do not allow searching code that does not suffer certain style issues, like Hypermodelling does.

Furthermore, Hypermodelling can use aggregation along the different hierarchies in source code. This way, queries can reveal: "All methods with a parameter of any type x that is defined in the javax.persistence package". Ordinary code search engines do not support all the hierarchies in code and do therefore not allow similar queries.

Test Driven Code Search

More advanced code search scenarios are based on source code mining infrastructures. A search engine on top of sourcerer [42, 41] [6] allows to search code much more precisely than the prior mentioned ordinary code search engines. For instance, private and public modifiers or method callers can be specified in a search. The search engine translates a query into sql and queries the relational sourcerer database. Out of this, the search engine inherits the prior described limitations of sourcerer. Hence, the key limitation for search process is that sourcerer focuses only on code structure and does not credit quality indicators that can be associated with code. Furthermore, sourcerer has not the aggregation and tuning mechanisms of Data Warehouses at hand.

However, on top of sourcerer the application of test driven code search is possible. Test driven code search uses test cases to reveal code elements, suitable to fulfill a test [157, 158]. Thereby, a test case is compared to other test cases that contain the same code. Out of this, the code elements are determined, suitable to fulfill the searched test. Then, an engine resolves dependencies of the test fulfilling elements and offers to possibility to extract those.

Also other research groups focus on test driven code search and provide similar infrastructures and tools for this kind of scenario [125, 126] . The needed code analysis engine to reveal the necessary artifacts is similar to sourcerer and called Merobase.[7] Therefore, Merobase offers a code search engine with advanced capabilities where code snippets can be entered and code that contains those snippets is revealed as addition to the test driven search. Internally, Merobase works with a relational database structure similar to sourcerer and is based on relational queries.

The main difference of test driven code search and Hypermodelling is that the test driven code search infrastructures are not indicator and Data Warehouse based. Furthermore, no quality indicators, like style issues, can be used to rank or filter the search results. This is a problem, because when search and found code gets reused, it is desirable that it complies to a certain quality. Thus, we imagine a benefit in extending test driven code search approach with the capability to limit search results by style issues and other quality indicators. Hypermodelling has such query capabilities, but cannot do test driven code search. Therefore, we see research needed how test driven code search can be combined with Hypermodelling.

Code Similarities Search

Other related work can be found in the area of code recommender systems and source code mining [54, 107]. Code recommender systems advance programming within the development environment [54]. They provide methods to mine data out of code bases and generate recommendations out of it by extracting cases what most developers did. Such recommendations are cases like: "Developers that called method a also called method b in 80% of the cases." When a developer encodes logic, the recommender system scans the recommendation database and proposes the developer what other developers did in similar situations. Therefore, the recommender systems are similar to a code search engine that computes usage indicators. Currently, the whole recommendation generation is hard wired and fixed on a specific set of algorithms. When the code base is altered and new patterns emerge the recommendation sets need to be generated again [2].

Our analysis of software (Chapter 7) is near the idea to mine recommendation sets from existing code [54]. Compared to the fixed analysis algorithms of code recommenders, our code analysis

[6] http://sourcerer.ics.uci.edu/code-search-engine.html - 7.11.1012
[7] http://www.merobase.com - 7.11.1012

with Hypermodelling is more flexible. Hypermodelling reveals facts with queries from from a Data Warehouse. For example, code recommenders can mine subclassing directives and generate recommendation sets that descibe which methods of a class or interface get often overridden in children. Our analysis (Chapter 7) shows that Hypermodelling can reveal more complex relations in source code. For instance, we uncover that children of the Serializable interface often override three different methods of the object class. The method presented in the mining subclasses paper [54] is not capable to uncover this information sticks purely to ordinary subclassing and method overriding directives.

However, we also described the case to use Hypermodelling for code recommendation in the IDE (Chapter 11). The main difference here is that recommendations with Hypermodelling are query based. This makes code recommendation more flexible. Adjusting queries and excluding parts of the code base by indicators is possible by altering a query. Code recommenders are hard wired and such ajustments are not easily possible. Furthermore, we propose live recommendations with a client server infrastructure to advance the approach even further. Therefore, we see the emerging need to investigate Hypermodellings use for code recommender systems further.

We also credit the work about clone detection (Chapter 11). Clone detection is about revealing duplicates or near duplicates of source code. Researchers describe different clone detection methods [223]. One clone detection method [263] proposes live clone detection with a client-server architecture. In comparison to our approach to use Hypermodelling and queries for clone detection, the described detection [263] is not based on customizable queries like Hypermodelling.

Another approach, CloneDetective [131], offers an advanced framework and tool chain for clone detection, which is especially geared towards flexibility of clone detection research. Hypermodelling, on the contrary, targets to utilize Data Warehouse technology. In general, we see the use of Data Warehouses as an addition to known clone detection mechanisms and not as competitor. Data Warehouses often already exist in enterprises where they are used for business applications. Therefore, our approach makes a reuse of Data Warehouse technology for clone detection possible. Hence, further research should determine which clone detection approaches can be realized with Data Warehouse technology.

All together, this section showed that the area of code recommendation systems and clone detection are near Hypermodelling and future research needs to investigate potential synergies in more detail.

In-memory Code Search

Other researchers dig into utilizing "in-memory" technology to create new code search engines. Thereby, a search engine stores the abstract-syntax-tree (AST) of program files in database table and utilizes the colum-row index of an in-memory database. Query tests of the stored information indicate that in-memory technology can significantly accelerate the process of source code search [201, 202].

Additionally, the researchers propose source code analytics as additional application on top of the in-memory database. Thereby, they reveal a simple scenario and create a Data Warehouse cube on top of the AST in the database. The cube consists out of seven different dimensions and focuses on the structure of one code file and does not contain "file crossing" information, like call hierarchies. However, the authors show indications that already the file centric information can help to reveal new facts about source code. In fact, the authors even mention that more advanced cubes can be useful for dependency investigations, by realizing call hierarchies and similar things in a cube model [203].

All together, this research is a strong indication that Hypermodelling is a step in the right direction. Often, reviewers of publications about Hypermodelling raised questions about its performance and if it is fast enough. The research and performance measurement of in memory code search shows clear and straightforward indications that today's database technology is performing well enough for such kind of applications as Hypermodelling is.

However, our work about Hypermodelling is differs, because of its holistic schema for code. The the in-memory case focuses only on seven dimensions. Aside this richness, the data model

of Hypermodelling also differs, because we used several advanced computation rules, where the in-memory data model does not address different computations of relations in source code. Hypermodelling focuses on the complete multi-dimensional hierarchies of source code, where in-memory code search just focuses on a primary AST hierarchy, without resolving the inter-file relations. Additionally, Hypermodelling proposes advanced scenarios, like code recommendation and code structure planning.

Finally, we see in-memory code search and analysis research as complementing technique to our research. Currently, the in-memory technology offers high-performance code search, but has limits out of the relational data model and the missing cube structures. Hypermodelling offers an extended relational and multi-dimensional model and adds supplemental application scenarios on top. Therefore, we see the potential to combine the two research projects.

12.3.3. Source Code Investigation and Management

Sonar[8] is a source code investigation tool that provides several reports and dashboard about code metrics. In general, source code is analyzed out of the version repositories with several tools. For instance, test results and other quality data, like code duplicate are computed and then stored in the sonar database. The primary hierarchy (projects down the package hierarchy to the classes and the methods) of source code is also stored in the database. Out of this, then different visualizations and reports are generated. Hence, sonar just makes a mapping of metrics to code and allows an analysis of this. The multi-dimensional structure of the code itself is ignored. Additionally sonar provides a dependency matrix of source code elements that can be considered as similar to the viewpoint of Hypermodelling. The huge difference is that the dependency matrix cannot be filtered for a specific dependency type like inheritance of methods or interface, like it is possible with Hypermodelling. However, also no OLAP technology is used in Sonar. Therefore, it can be interesting to investigate if and how Hypermodelling can be docked onto it.

Lattix[9] is a tool that provides different dependency matrices of source code projects. Users can explore those decencies and see which package and types are dependent on which other. A special feature of Lattix is that rules can be specified where no dependencies should be. For instance, a data–access layer should not be dependent on the service layer. Lattix makes it possible to specify that there should be no relation. Then, the maintaining of this rule can be checked in the further development process. However, in contrast to Hypermodelling, Lattix is limited to pure dependencies and does not target the use cases of code structure cockpits, code recommendation and detailed code structure exploration. Furthermore, Hypermodelling supports different kinds of dependencies through its multi-dimensional architecture. For instance, dependencies on classes and interfaces can be distinguished what is not possible with Lattix. All together, Lattix gives detailed insights for what and how dependency matrices are useful. Therefore, it can be interesting to study all the use cases for Lattix and relate them with possible enhancements of Hypermodelling.

Blackduck[10] is a company that offers several products for code investigation. One product is a code search engine that allows selecting different parameters in a code search cycle and to specify the license of the source code. However, the search engines capabilities for parameter search are limited and in contrast to this the whole different hierarchies and the multi-dimensionality can be respected with a code search engine based on Hypermodelling. For instance, methods can be searched that have a parameter out of a certain package and a parameter with a specified superclass. This is not possible with the code search engine of Blackduck. Additionally, Hypermodelling offers the possibility to built cockpits for code structure and to use the whole Data Warehouse toolbox, what is also not supported in Blackduck products. The Blackduck suite only offers here only pre-defined cockpits without the possibility to create custom cockpits for code structure. However, the products of Blackduck are interesting and a various aspects of the development cycle are covered with them. Future research can investigate synergies between the approaches.

[8] http://www.sonarsource.org/ - 7.11.1012
[9] http://www.lattix.com/ - 7.11.1012
[10] http://www.blackducksoftware.com/ - 7.11.1012

Understand[11] is a source code metrics analysis tool. It supports to investigate the source code for different metrics. Furthermore, dependencies within the source code can be investigated. Therefore, the software offers various visualization graphs. All together, Unterstand differs from Hypermodelling in not using a multi-dimensional viewpoint. The offered graphs are there from the perspective that is considered by the tool. For instance, the dependency browser explores to explore the codes relations. With Hypermodelling the perspective could be switched, but understand uses a fixed point of exploration. However, in general the graphs of Understand are complimentary to Hypermodelling and illuminating possible synergies can be interesting.

Klocwork offers various tools for the development process. One product is for static code analysis and offers various visualizations to explore code. Again, the tool differs from Hypermodelling, because it is just a traditional code analysis tool that does not focus on a multi-dimensional perspective. However, like in the other cases in can be interesting for future research to do an in detailed comparison to reveal synergies.

The **IBM Rational Team Concert (RTC)**[12] tools provide dashboards for projects with a set of predefined widgets, allowing monitoring of the 'health' of a project. The main difference to our proposed cockpits is that we focus on queries for the source code structure and RTC cockpits show mostly project statistics out of issue tracking. Therefore, we see possible synergies for further research in extending the RTC dashboards with Hypermodelling based ones.

A book [204] refers to a special Data Warehouse for software metrics and describes a method to control a software development project with those. Since the measures calculated in this dissertation represent somehow metrics of source code this seems first to be related. However, the metrics described in the book are economic project figures and differ from the code relation indicators and computations of Hypermodelling. Nevertheless, future research can investigate the relation of code structure metrics and the project management metrics mentioned in the book.

12.3.4. Semantic Concern Representations

Hypermodelling uses a multi-dimensional data and computation model. Therefore, we see research about modeling the concern space and software ontologies as related. We present this research briefly here and point out the differences to Hypermodelling.

Software Evolution ONtologies (SEON)

Software Evolution ONtologies describe the domain of software evolution analysis and mining software repositories. Thereby, the research targets to cover the holistic cycle of software development including source code, revisioning, issue tracking, tools, evolution movements, stakeholders and other associated artifacts and technology [1, 261]. A Java ontology describes relations, occurring in Java source code [1].

However, the ontologies follow a different approach than Hypermodelling. Hypermodelling has its main focus on aggregation and a multi-dimensional viewpoint. Ontologies focus on knowledge representation. Today, the ontologies cannot be used to generate the reports like it can be done with Hypermodelling, easily.

We consider the ontologies modeled knowledge as opportunity to advance Hypermodelling. The ontologies already describe how the data of different tools relates to code. Investigations can dig into this, use the information about Hypermodelling and reveal how and if it is possible to generate the corresponding data cubes and relational model out of the ontologies. All together, we see further investigations in this area needed.

Concern Modeling

In this work, we focus mainly on code, but see the area of multi-dimensional graphical concern space modeling as related. Thoughts about enhancing the Unified Modelling Language with a multi-

[11] http://www.scitools.com/ - 7.11.1012
[12] http://www-01.ibm.com/software/rational/products/rtc/ https://jazz.net/products/rational-team-concert/- 7.11.1012

dimensional viewpoint are described in [92]. Other researchers describe the concern space of a program in other ways [245, 246, 179, 245, 247].

All together, the multi-dimensional graphical representation of concerns differs from the Hypermodelling approach, since Hypermodelling focuses mainly on code. However, it is imaginable that Hypermodelling is used to work together with graphical models. Therefore, we see the need for a detailed review about graphical concern representations in future research.

12.4. Discussion and Down Sides

In the following, we discuss different known down sides of Hypermodelling. Thereby, we address potential raised critique points in different subsections.

12.4.1. Yet Another Tool

There was an effort in developing and implementing Hypermodelling that needs to be justified. One may wonder and ask why we need a new approach and yet another tool and do not use the different existing approaches of related work. The answer is quite simple: Because the benefits of integration on one platform exceed the efforts of many scattered tools. Furthermore, integration, like it is done with Hypermodelling, ensures consistency and a central spot of analysis. In addition, synergies between the different, currently scattered, approaches can be unleashed.

Nevertheless, Hypermodelling adds the complexity of integration to the current research and tools. Our current implementation solved this concrete challenge for the Java programming language. Therefore, this supplemental effort was already done by us, in the case of Java. However, for other programming languages and Data Warehouses, we provide a concrete guideline and framework. Additionally, the concrete implementation can be adapted and taken as reference, what reduces the effort for future implementations.

We argue that the shown use cases of Hypermodelling also justify the effort that was done. For instance, the different analysis scenarios of this dissertation would have taken a lot of more effort, if they would have been done with current tools. Furthermore, if additional analysis scenarios are done, they will exceed the effort that we needed in order to create Hypermodelling within a short time. Hence, the down side of higher effort is just one that occurs for a single investigation. When multiple investigations are done, this down side can be neglected. In special, we argue that often it is complicated to transfer research results to practice, because the investigations that are made in research are made with specific projects. If a transfer of results for a concrete enterprise solution is wanted, the same studies need to be done again for the specific projects of an enterprise. With Hypermodelling the investigations of such different projects can be done with little effort, since the complexity and effort is shifted into the Hypermodelling implementation.

Thus, we are certain that the effort to derive and implement Hypermodelling is justifiable in cases where multiple investigations are needed.

12.4.2. Data Warehouses are Heavyweight

One of the most often raised argument against holistic solutions, like Hypermodelling, is that they are not lightweight. We induce that Hypermodelling is created for big scale code bases. No solution that handles big data is lightweight. Thus, we have to compare the like with like. Clearly, Hypermodelling is heavyweight for a single application, but the main intention is to derive a technique that handles big data. There, it is suitable.

Nonetheless, we also introduced a future IDE architecture that strips down the used Data Warehouse components to an OLAP processor within the IDE, what we can call lightweight (see Chapter 10). But even in that case, one may rise that a custom implementation for a specific use case, like for instance code recommenders, is even more "more" lightweight. Nevertheless, Hypermodelling is generic and targets multiple use cases. If just one case is demanded, clearly a customized solution for a specific case is always more lightweight. We cannot argue against this, because not beeing lightweight is the general price of integration and a generic technique for multiple use cases.

12.4.3. Relational Model is an Overhead

Another down side of Hypermodelling is its dependency on a relational model to be compatible with most Data Warehouse solutions. Direct loading of source code into the data cubes would have spared to develop the relational model. Furthermore, the processing overhead to transform code first relational and then multi-dimensional would not exist. However, related research (see the section called “In-memory Code Search”) also describes solutions to store source code in in-memory databases. Those in-memory databases are still founded on a relational model and offer a huge performance increase, compared to traditional databases. Currently, our approach is compatible with these databases, because Hypermodelling still has a relational layer. Therefore, we see a benefit in the relational model for Hypermodelling.

Thus, the light in this current approach of using the relational model is that it makes the technique compatible with most Data Warehouses. The down side is overhead. Its the trade of being compatible or creating a special solution. Currently, we credit Data Warehouse reality with a relational data store. Nevertheless, we agree about the overhead and are confident that this down side can be overcome in the future by innovative solutions. Such a solution can be to create a virtual database access layer/solution for source code that emulates a relational database. This way, a relational source code access of the Data Warehouse could be realized without having the necessity to load code into the relational schema and avoid the overhead.

12.4.4. No Real Time Data

Hypermodelling is currently dependent on regular data loads into the Data Warehouse, what means that there is no real-time data available. In general, this is a normal down side in Data Warehousing. However, loading can be done in short cycles, what leads to near real-time data. Furthermore, continuous integration servers generate builds and tests every time a check-in of source code into a repository is done. Similar mechanisms can be introduced for Data Warehouse updates. Aside of this, our IDE implementation overcame the non real-time limitation, by providing a live concern detection in queries (see Chapter 10). Nevertheless, this approach in the IDE slows down the query execution time. Therefore, we see a trade off between loading data and on-demand parsing.

Additionally, research in Data Warehousing investigates how in-memory databases allow data analysis in, or at least near, real time to overcome the corresponding trade offs with normal business data [252]. Therefore, we are confident that research in this area can be transferred to the area of Hypermodelling and help there to overcome the current issues. Nonetheless, future researchers need to investigate the possibilities and opportunities of new Data Warehousing technology for Hypermodelling further.

12.5. Summary

We showed the main differences of motivating and other related work to Hypermodelling. We recognized that Hypermodelling has advantages to the techniques of prior art and that future research can reveal additional synergies. Then, we described related research and compared it to Hypermodelling. Again, we recognized that Hypermodelling is unique and future research can reveal further advancements in the diverse areas. Lastly, we did a discussion and invalidated potential raised arguments against Hypermodelling.

13. Summary, Conclusions, Vision

"Save the Cheerleader, Save the World."
Hiro Nakamura. Heroes. Season 1 Episode 4: Collision. 2006

Abstract. Hypermodelling is our new multi-dimensional and big data handling technique for program analysis, code reporting, code search, IDE enhancements, and code structure planning. One may wonder what the remainder of this work about Hypermodelling is. Which scenarios did we show? What are the contributions of Hypermodelling? What are the avenues for future research? What is our future vision? This wraps up the content of this thesis. We answer the questions and draw conclusions. Thereby, we reveal insights about different avenues for further research and present our general future vision.

13.1. Introduction

We derived Hypermodelling and studied it in various application scenarios. We presented related work and showed the differences. Like it is in every research project, achievements were made, but more questions stay open. In order to give an overview about the achievements, the open questions and future work, we provide this chapter. Thus, this chapter contributes to our work as follows:

13.1.1. Contribution

We provide a wrap up and outlook of the main remainder of this thesis. In special, we provide:

- Summary
  We provide a wrap up of all chapters
- An evaluation overview
  We describe which chapters studied which perspective of the new developed Hypermodelling approach
- Contributions
  We point out the main contributions of this thesis
- Future work
  We provide a detailed outlook to main future work and research paths
- Concluding Vision
  We provide insights about our own conclusion and future vision for Hypermodelling and software engineering

13.1.2. Reading Guide

First, we provide a summary of the thesis. Then, we discuss which different perspectives of Hypermodelling have been studied and give an overview about those. That following, we get into the details of the concrete contributions of this thesis. Then, we present avenues for further research. Finally, we close this thesis by presenting our concluding vision for Hypermodelling and software engineering.

13.2. Summary

The central goal of this thesis is to introduce and evaluate the Hypermodelling approach. Hypermodelling is a Data Warehouse based source code analysis technique and motivated from the continuous growth of code repositories and the current state of the art in different areas of software engineering.

At the beginning of our work, we depict the current challenges in software engineering and describe how different research tries to deal with today's challenges. Those current approaches are multi-

dimensional modularization techniques, virtual separation of concerns, domain specific languages, Orthographic Software Modeling and source code analysis.

However, we look at another area that deals with large amounts of data. We present Data Warehouses as big data analysis method and infrastructure that is widely applied within enterprises.

Succeeding, we use the key contributions and addressed problems by prior art in software engineering and reveal demands that a new technique should meet. We recognize that no technique of the current state of the art addresses all of this claims and propose to use a Data Warehouse based approach to integrate the various claims into one new technique. As solution, we present the Hypermodelling approach, utilizing Data Warehouse technology, and define it as follows:

Hypermodelling is the technique to structure and access concerns of source code. The access is done by an access translator that uses a multi-dimensional concern model. The access translator is an abstract machine that supports navigation, computation, aggregation, mining and any other kind of read, write or presentation operation on concerns. The multi-dimensional model is used to structure concerns and contains data about:

- Concerns defined within source code;
  AND / OR concerns that are specified by additional data, whereby this data contains a concern specification, associated with at least one region in source code
- At least one numeric indicator representation of at least one concern or a concern relation. Thereby, an indicator links one or more code fragments or modules to a concern or a relation
- At least one relation between concerns or code fragments,
  whereby the relation can be used to aggregate at least one sum of at least one concern relation;
  AND the relation maybe also a hierarchy

Additive, any mean of concern detection to retrieve concern indicator associations may be used.

Optimally, Data Warehouse technology is used to realize Hypermodelling. The access translator equals an OLAP Processor. The indicators equal Data Warehouse indicators. The Data Warehouse dimensions equal concerns. If multiple artifacts belong to a concern, e.g. dimension, those are the members of the dimension. The code hierarchies equal hierarchies within the Data Warehouse. The indicator concern associations and relations in code equal facts.

That following, we focus on a technical architecture to use today's Data Warehouses to realize the Hypermodelling approach. The theoretical contribution is rounded up by providing a concrete implementation for Java source code for a Microsoft Data Warehouse. Then, we study different aspects of the application of Hypermodelling.

We address the obvious usage of Data Warehouse systems with Hypermodelling by presenting a query based code investigation. This indicates that Hypermodelling is suitable for code analysis.

Afterwards, we evaluate the feasibility to create cockpits for source code structure to enable investigations without query knowledge. Our implementation illuminates effort to realize such cockpits and describes additional advanced use cases. This indicates that Hypermodelling is suitable to create further cockpits with reasonable effort.

We close the traditional reporting application scenarios and refer to economic planning scenarios with a Data Warehouse. We evaluate if the migration plan of a program can be done with Hypermodelling. First results indicate suitability and the need for further research.

We study an additional scenario to use Hypermodelling within the integrated development. For this reason, we provide an implementation that is done especially for the IDE. We evaluate the implementation with various usage scenarios. Supplemental, we show that the prior used Data

Warehouse based implementation of Hypermodelling can also be used as search engine and be integrated into the IDE. Thereby, we provide concrete details of future reference architecture for an IDE with integrated Data Warehouse components. All together, the IDE use case provides supplemental support that Hypermodelling is a useful approach within IDEs, too.

We strengthen the indications of the usefulness of Hypermodelling within the development environment, by the following application scenario for live clone detection and recommender systems. Thereby, we show examples and insights how a Hypermodelling based recommender system can work. This indicates, again, the technological benefit of Hypermodelling.

We finish studying application scenarios of Hypermodelling by giving a detailed comparison to related work. We see that Hypermodelling is unique and future work needs to study plenty of possible synergies.

Additionally, we find, investigate, explain and transfer research from psychology that focuses on mental information structuring in the appendix. Out of this, we provide a new program comprehension model and create a theory that mental structures manifest in programming language structures. All over, this research provides additional support that a dynamic and multi-dimensional approach like Hypermodelling is needed. Furthermore, we use Hypermodelling to do a first preliminary investigation and find evidence that mental categories seem to materialize in source code structures.

All in all, we provide a new approach of Hypermodelling, different application scenarios, implementations and theoretical foundations that indicate a benefit of Hypermodelling.

13.3. Evaluation Overview

Hypermodelling is derived out of claims of different areas by inductive reasoning. Aside of the general realize ability, we also see the need to verify that there is concrete benefit from this new approach. Thus, we present specific application scenarios to demonstrate the usefulness of the technology aside the obvious motivation that a multi-dimensional concern space needs multi-dimensional technology. We see this as necessary, because concrete application scenarios illuminate and reflect experiences with Hypermodelling better than a pure theoretic argumentation. Lastly, we argue for a key point of integration of our technology and see the need to show that multiple areas in software engineering can benefit from our approach.

Therefore, this section focuses on the concrete usage of Hypermodelling. We point out how we evaluate different parts of Hypermodelling in the diverse chapters of this work. Hence, we provide in the following a brief overview how different excerpts of this dissertation serve as evaluation for Hypermodelling.

13.3.1. Technical Realization

First, we evaluate the static qualities and the internal complexity of our approach by applying our framework to create an instance for a concrete Microsoft Data Warehouse (Microsoft Analysis Services 2008). We present a relational schema that represents Java source code and its relations. Since most Data Warehousing solutions dock on relational data sources this would make it possible to “transfer” our solution on different Data Warehouse technologies. As addition to the relational source code model, we also develop a rudimentary parsing mechanism to load one project into the model for further testing purposes and demo scenarios.

Additionally, we define different cubes of source code and associated artifacts to provide examples how further cubes can be implemented. Thereby, we specify processing rules to compute the relations for the multi-dimensional data structures. This way, we show that a technical realization is possible. We present this in Chapter 6.

13.3.2. Code Analysis

We investigate the suitability for code analysis of our technique by constructing the scenario of inheritance investigation. We take this scenario and show how different perspectives of inheritance

in a program can be revealed. The scenario does not only comply with the utilization of our approach for analysis, but rather gives insights about a real world application. We see a long and holistic application with different charts and drill downs into details that demonstrates Hypermodelling as suitable for this kind of applications. We present this investigation in Chapter 7.

We advance the scenario even further, by showing insights how we use the prior named inheritance investigation as foundation to create a management dashboard. This provides all necessary indicators about inheritance at one place. The dashboard enables the argumentation that Hypermodelling offers advancements to similar dashboards that have been already proofed to be useful in software project management. Furthermore, we use a dashboard realization as experiment to quantify the effort to realize such a cockpit. This indicates a small effort of the realization of such applications, what again supports the advancement of our approach. We show the construction of this dashboard in Chapter 8.

In Chapter 7 and Chapter 8 we show also concrete usage samples. For instance, we compute a cohesion indicator or investigate dependencies. This way, we ensure that our approach contributes to real world code analysis scenarios. Additionally, we present new scenarios, like mining developer knowlege that Hypermodelling enables. Together, this is another reason for the value of our approach, because we fulfill present requirements and enable new possibilities.

As addition to all of those scenarios, we note that the data that we compute in those investigations can serve as foundation for applying data mining algorithms to reveal potential structures within the data. In general, these scenarios show that analysis investigations to reveal different facts for further information processing are possible with Hypermodelling.

13.3.3. Code Structure Planning

We evaluate the application ability of Hypermodelling to plan future code structure. Thereby, we present first insights about concern planning in Chapter 9. This is done by a showing a first and generic concern planning approach and its preliminary evaluation by the migration of a demo application. In general, this reveals indications that Hypermodelling can be useful in the area of concern planning, too. Furthermore, this attempt shows that more investigations are needed. This strengthens again the argument that Hypermodelling enables future research.

13.3.4. IDE integration

We create an application for the development environment Eclipse to study the application ability of Hypermodelling within the IDE. With this, we show that an OLAP like query based approach can be applied within an IDE in Chapter 10. This enables us to query for concerns out of code structure by using drag and drop capabilities. Thereby, we evaluate a first usefulness of the application within the IDE with different usage scenarios.

As addition, we credit the development of future IDEs and show how Data Warehouse technology can be utilized as source code search engine.

All in all, this IDE application shows that Hypermodelling can be applied successfully as query tool within the IDE.

13.3.5. Query based Advancements

In order to strengthen our argument that Hypermodelling can be used to apply results of source code Mining, we show the use of Hypermodelling for code recommendations and clone detection (see Chapter 11). This shows another beneficial approach for research and practice of Hypermodelling.

13.3.6. Related Work

We wrap up all the shown features and applications of Hypermodelling and compare it to related work. Thereby, we summarize the features of related work and the ones of Hypermodelling (Chapter 12). The comparison to related work shows that Hypermodelling is an integrative technique

and it affects many areas of software engineering. This indicates again the value of Hypermodelling for software engineering research.

13.3.7. Code Study Model

We present a code study model and relate it to Hypermodelling (Appendix A). We do so, to look beyond our own nose to reveal if other areas support the need for a flexible and multi-dimensional source code access. The presented code study model shows that research of psychology also assumes that there are different vantage points of categories. Also, there objects belong to different categories at the same time and the advantage point influences which category is recognized. Our Hypermodelling approach fits to this, since it allows multiple viewpoints. Additionally, we use Hypermodelling to verify the comprehension model and see that Hypermodelling is suitable to do statistic analysis. Therefore, we see this as additional support for our work.

13.3.8. Evaluation Overview

We described how we evaluate different aspects of Hypermodelling. We focused on diverse application scenarios of Hypermodelling that indicate benefits within diverse areas of software engineering. The different applications that we use for the evaluation also address the integrative role of Hypermodelling.

In order to focus on the evaluation in the different areas again, we give a sum-up of the presented evaluation approaches within those areas in Table 13.1. Thereby, we show the different chapters and the evaluation approaches. We see that different chapters contribute with different evaluation scenarios to this work. We study static properties of Hypermodelling, observe it in scenarios, apply it in experiments, describe its properties and use informed arguments to evaluate the approach holistically. This different perspectives and study methods underline there are several benefits in Hypermodelling.

Table 13.1. Evaluation Methods

Chapter	Evaluation approach	Evaluation Contribution	Comment
Chapter 6.	Static Analysis	Relational and multi-dimensional Model for Java	The static properties of Hypermodelling are studied by deriving a technical implementation.
Chapter 7	Observational, Case Study	Exemplary analysis scenario	Study the general application to investigate code structure
Chapter 8	Experiment, Analytic, Dynamic qualities	Code Cockpit creation and application, Effort measurement; Advanced reporting capabilities	Effort and application ability study to create dashboards
Chapter 9	Experiment-Simulation	Example code migration scenario	Examples show new possibilities and the need further research about concern association movements
Chapter 10.	Descriptive - Informed argument, Scenario	IDE plug-in; evaluation through application scenarios	Application scenario shows argumentative support for Hypermodelling
Chapter 11	Descriptive - Informed argument, Scenario	Query based recommendation process	Enables the comparison to a specified scenario of related work
Chapter 12	Descriptive - Informed argument	Comparison to related work and its features, Discussion about trade offs	In detail comparison about the differences to related work
Appendix A	Descriptive - Informed argument, Scenario	Hypermodelling is related to the human cognition, Complex statistic code analysis	An additional perspective about research about categories in psychology; advanced code analysis

All together, the different evaluation methods are used to illuminate different aspects of the Hypermodelling approach. Therefore, we see plenty of support that Hypermodelling is a technique with value and enables future research.

13.4. Artifacts and Contribution

Based on the motivation out of the state of the art and today's growing code bases, this section presents the general 11 contributions and steps-forward that we were able to make within the scope of this dissertation.

13.4.1. New Perspective for Code Analysis

First and foremost, this thesis introduced the Hypermodelling. This enables researchers and practitioners now to use common Data Warehouse technology and multi-dimensional computation models to investigate source code. Now, big data handling technology and a whole investigation infrastructure is immediately available for source code investigations. Among others, this provides benefits like multi-dimensional operations, navigation, aggregations and reuse of whole Data Warehouse tool suites for software engineering that has not been supported by prior art.

13.4.2. Implementation Guideline for Data Warehouses

An implementation guideline and framework provides concrete details how Data Warehouse technology can be used and refers to exemplary applications that can be created on top of them. In detail, we give concrete descriptions how Data Warehouse technology can be used to integrate source code and related data to do analysis. This enables to integrate data from various sources into a Data Warehouse. Furthermore, these integration capabilities allow now to respect the different perspectives of a program to be integrated in analysis scenarios. In addition, the exemplary application areas point out that our technology can be applied for different use cases.

Compared to the prior art, this is an advancement, because with Hypermodelling central tool suite for analysis, code search, code reporting, IDE enhancements and code planning is available what makes the creation of custom tool sets for code analysis obsolete. Before our contribution, no central platform and guideline how to use Data Warehouses was at hand. Now, future work can leverage Data Warehouses and all their capabilities for software engineering, without having the necessity to create custom tools for each scenario.

13.4.3. Hypermodelling Implementation for Java

We provide a detailed implementation of Hypermodelling for the Java programming language. This concrete implementation leverages the immediate application of the Hypermodelling approach for Java. Furthermore, our computational models give specific insights how relations, dimensions and a relational application manifest and get tied together in a Data Warehouse. This can be used as template or as reference implementation for other programming languages.

This is a huge improvement compared to prior art, because now analysis and further applications of source code can be done with OLAP methods. Prior art does not provide computation directives for aggregations of structure within source code. Now, researchers and practitioners can investigate Java source code with Data Warehouse technology. Thereby, they have visualization and query capabilities of Data Warehouses directly at hand.

13.4.4. Application Evaluation for Code Analysis

We provide a detailed investigation of source code and demonstrate the suitability of Hypermodelling for this kind of task. In order to provide such a holistic study about inheritance and dependencies within source code much larger effort would have been needed with prior art. Furthermore, our provided visualizations in the study with different charts demonstrate the benefit of using the Data Warehouse tool set. The study shows that Hypermodelling restricts the effort for source code investigations to a minimum. Now, ad-hoc investigations of source code can be done.

13.4.5. Code Cockpits and Effort Evaluation

We provide a cockpit and reduce the effort and required knowledge to do a code study with Hypermodelling to the minimum. We show that investigation knowledge of manual code studies can

be integrated in a code structure cockpit with reasonable effort. Even more, we provide an outlook to future cockpits that contain role specific information like developers. This kind of cockpits enable now structured code investigations in an easy way. Supplemental, the implementation effort of the cockpit shows that such cockpits can be created within a reasonable time frame.

Compared to prior art, this is a huge advancement, since for the development of few cockpits whole research projects have been needed. Additionally, prior cockpits were not able to be used for code structure investigations to reveal details about the codes architecture. Now, researchers can determine and implement the most desired cockpits for practice.

13.4.6. New Indicator based Code Planning Scenario

We advance the current state of the art of code migration planning by the proposal to use Data Warehouse based indicator planning for future code structure. Thereby, we describe how the progress of code updates can be measured. We discuss an example update of a demo application and evaluate our approach preliminary.

This is an advancement to prior art, because current practices in code structure planning are currently not done with indicators. Now, future research can compare and advance current planning mechanisms with Data Warehouse planning mechanisms.

13.4.7. Lightweight IDE Hypermodelling Implementation

We provide an implementation of Hypermodelling for the IDE. Thereby, we present a drag and drop approach for multi-dimensional query composing and a filter based result presentation. Using multi-dimensional queries within the IDE is new, especially combined with filter capabilities. Through this approach, developers have now a mechanism at hand that allows creating queries easily out of their known views, without having the necessity to learn a complex query language. Furthermore, through result filtering, they get presented the results of a query in their most used view, without having to focus how to interpret a custom result presentation. Evaluative application scenarios show the usefulness of our approach.

In addition to the IDE implementation, we also give a vision without a future IDE with active Data Warehouse components. Thereby, we illuminate a concrete high level architecture that can be used to enable further and advanced applications in the IDE. Compared to prior art, our new way of query design and representation and a Data Warehouse enabled IDE is a huge advancements, because other researchers see the future of the IDE also to be more analytic [266]. Our vision contributes to this goal, by proposing specific analytic components within the IDE.

13.4.8. Indicator based Code Search Engine

We provide an advanced code search engine based on a Data Warehouse back end. We describe how the search engine utilizes the multi-dimensional model and point out that indicator enabled search engines are now possible.

This is a huge advancement to current code search engines, since now quality indicators can be used within a code search process. Current search engines do not support result filtering based on indicators. This makes it impossible to reveal only code that corresponds to a certain quality. By utilizing the Hypermodelling approach as code search engine, the effort to create a code search engine is reduced to a minimum and quality indicators can be integrated easily in a query. Now, custom code search engines can be implemented easily.

13.4.9. Scenario Code Recommendation and Clone Detection

We advance the current state of the art in clone detection and code recommendation by proposing to use a query based approach with a Data Warehouse to handle large size code bases. Our query based approach is more flexible then current, static, recommendations of another research group. We contribute to the area of code recommendation with the possibility to exclude or include code fragments out of quality indicators out of project structure. Furthermore, we point out that a query

based approach can be easily customized and does not need recommendation set generation any more. With Hypermodelling, recommendations can be generated just with queries against a central repository.

Clone detection is advanced by the proposal that differences to a potential clone can be computed with the approach. This makes it possible to use the determined differences and similarities to a potential clone and show it to a developer. However, here again, the main benefit is that a query based approach is easy to customize, where excluding or including certain code trees can be done by just adjusting a query and a Data Warehouse infrastructure can be used.

13.4.10. Category-based Code Study Model and Evaluation

Lastly, we contribute with an excursus about mental knowledge structuring in categories and provide a program comprehension model based on this research. Our model advances the current state of the art to regard separation of concerns as natural given knowledge structure, by introducing a mapping of separation of concerns to categorization research of psychology. Furthermore, we provide a first evaluation and find first indications that the usage of means to separate concerns is similar to categorization. Now, future researchers can investigate these similarities further.

In addition to the comprehension model, we use it to argue for a multi-dimensional viewpoint towards software that is supported by Hypermodelling. We explain in a discussion that the multi-dimensional viewpoint of Hypermodelling corresponds to our research how mental categories are structured. This way, we provide a verification of our approach from a complete other area.

13.4.11. Emerging first Results

Supplemental to these Hypermodelling centric contributions, we provide emerging first results in our source code investigations that are scattered throughout our work, when we use Hypermodelling for analysis. Those results need to be investigated with more source code in the future. We see the most important facts that we find out incidentally as follows:

- Inheritance is more done with classes, compared to interfaces

  We found out that inheritance is mainly done with classes, even though that class inheritance means strong couplings. Strong inheritance couplings are often considered to be a bad programming style. Our current investigations show that classes are far more extended then interfaces implemented. We see this as important fact for further research, where it is often said that implementing interfaces is better than extending classes. If developers do bad things, we as researchers have to ask why and have to come up with methods that overcome the why reason. Therefore, we see the need fur further investigations about this finding.

- Developers have different skills and we can reveal them from code

  Today we read in every magazine that the industry competes for the best talents. Software developers are such talents. We show the first applications of Hypermodelling to mine skills of programmers. In special, we demonstrate how we apply Hypermodelling to mine developer knowledge and done Style Issues out of source code. Future research needs to advance our first attempts in this area.

- Mental categories manifest in source code similarly to psychological categories

  We show that means to apply separation of concerns are similar to psychological categorization theories. Our first evaluation is not enough evidence to verify this completely, but it illuminates the need for further investigations. Therefore, future research needs to reveal in more detail how and if mental categories manifest in source code.

13.4.12. All Together

All together, our research contributes to diverse application areas and reveals that Hypermodelling is what it is meant to be: An integrating technique for software engineering, from which different

approaches and tools in software engineering can benefit. Future research can now built software engineering tools based on Data Warehouse technology.

In summary, we developed a new approach for software engineering and showed applications that underline the benefit of the approach. Additionally, we revealed incidental results about source code that need to be verified in the future. Now, researchers and practitioners can use Data Warehouse to do source code do their own source code investigations or develop and utilize the different application scenarios. All in all, Hypermodelling is a contribution that advances prior research and reveals new possibilities for research and practice in the future.

13.5. Future Work

The development of Hypermodelling and assembling this thesis, brought up plenty of new questions for future work. Thereby, also ideas for special use cases in combination with other work came up. In the following, we present areas where we see the most emerging need for further research.

13.5.1. Slowly Changing Dimensions

First and foremost, we see future implementations of Hypermodelling in the need to determine how source code evolves over time. Within Data Warehousing this is called slowly changing dimensions and ensures that changes over time do not lead to semantically wrong computations [143]. For instance, we imagine a method signature changes over time. Currently, when different versions of this methods signature are loaded with Hypermodelling, the method gets recognized as two different methods. This gets problematic when a time-dependent report is wanted. When the usage of different revisions of a library is analyzed, this problem excels. Currently, the libraries changed methods are treated as different methods. Therefore, the usage amount of a changed method is not computed additive and divided in two distinct methods. This is semantically wrong. Hence, we see capability to report historical reality as a crucial future feature. Therefore, slowly changing code dimensions need to be investigated and implemented. Additionally, investigating revisions is also related to our code planning approach with indicators. Therefore, we see the necessity to recap and advance our planning technique in this context, again.

We did already do some digging and found out that other source code analysis and search infrastructures are also not credit code evolution. However, in the area of software ontologies, researchers focus of program evolution and change types [90]. Therefore, we see the first step to develop a slowly changing code dimensions in investigating the research about program evolution change types.

13.5.2. Extended Data Integration

Secondly, we see a next step in future research in integrating more data and methods from other areas. In the following, we present different research areas, holding interesting data and methods that can leverage new applications of Hypermodelling.

We consider developer centric information as interesting. It would be valuable to measure the time how long a developer regarded different source code artifacts. With this information, queries for potential hard to-, or easy to read source code can be done. Furthermore, this kind of information can be used to determine which code has been most often implicitly reviewed. We also imagine applications like real time mining. Such can be that a developer studies different source code elements and gets active pop-ups, proposing what other developers did after they studied the same fragments.

Additionally, developer study information allows estimations about their potential knowledge. In a similar context, researchers already created degree of knowledge models [105] that specify how the knowledge of developers evolves or decreases over time. We see the integration of such models and their computation as potential benefit in scenarios about investigating a developer's knowledge.

As another type of data we consider additional code quality metrics [133]. Code quality information is important for source code audits. Auditing is especially important in outsourcing scenarios, where

ready-to-use code is delivered and merged into a companies own code base. A main reason for outsourcing is to save time and resources. Hence, a quick quality investigation of the delivered code is needed. Hypermodelling's persuasive query capabilities can be combined with quality metrics to investigate the delivered code structure. This can reveal spots where adjustments of a supplier's code need to be done, in order to fulfill the desired quality metrics. Therefore, we see a future research trial in the integration of additional code quality metrics into Hypermodelling.

Other research reveals insights about scientific debugging algorithms [267, 268, 269]. We are confident that integrating data and methods from this approach can lead to synergies, since Data Warehouses offer statistic computation and analysis capabilities. Therefore, we see a detailed investigation about potential synergies needed.

Currently, multi-cores and multi-threaded programs are rising more and more. Even smartphones got already eight-core processors. In order to manufacture multi core applications, developers use program analyzers and profilers that help to reveal and tune the program parts that are mostly responsible for slow downs [168, 226]. Supplemental, when multi-core programs are developed, debugging gets very difficult, since it is hard to discriminate which thread-state combination is responsible for a crash. Therefore, researchers investigate bug determination mechanisms that work, among others, with failure dumps and thread scheduling data [61, 258]. Hence, we imagine that this data can be integrated into cubes and the query capabilities of Hypermodelling can help to investigate multi-core applications. Therefore, further research about multi-core applications and Hypermodelling is needed.

Lastly, we see the need to integrate data of feature oriented programs. For future work, we see the possibility to integrate a feature model into Hypermodelling, to allow the investigation of feature oriented programs. Here, we are confident that feature analysis and feature evolution can be one potential use case. Furthermore, if the prior mentioned runtime and bug information is integrated at the same time, we see potentials to advance bug detection in the area of feature oriented programming.

In general, we need to note that each mentioned data that is integrated with Hypermodelling gets associated indirectly with other integrated data. For instance, if features are associated with a code fragment and a bug is also, an indirect relation between those two exists. This relation can then be analyzed with Hypermodelling. Therefore, we imagine that the integration of the prior mentioned data can lead to new and innovative use cases.

13.5.3. Physical Indicator Association

Similarly, to the prior mentioned data integration, we see also a future research trail in analyzing physical indicators of a program that runs a machine. In the following, we give a brief overview about the idea.

When software operates a machine, multiple physical variables can be measured via sensors. For instance, heat, pressure and energy consumption can be measured. Engineers face the challenge to analyze indicators and source code of a machine to gain insights if a machine operates within the desired set of parameters. Thus, a technology is needed, allowing the investigation of indicators as well as the code structure at the same time. A main application of a Data Warehouses is computation of economic indicators. – So why not use physical indicators?

We imagine that Hypermodelling can be used to map energy and time to methods of software. Our current approach shows how to represent software in a Data Warehouse. Physical indicators can be added to this model. Furthermore, profiler data can be added to determine which methods are executed most often. Then, probably consumed energy in most cases can be computed. With this information the developer can query the spots where the most energy is consumed.

Another idea is to use failures of software systems as indicators. Just imagine huge software systems that are composed of 1000s of components – maybe even parallel executed. All code is executed asynchronous. One time the different parts of the software are in one state; another time they are in another state and lead to a failure. But from where is the failure coming? - This is hard to determine.

Or we imagine a car. There, software product lines and plenty of different hardware gets used. When a failure occurs in car model a – and also in car model b, it may not occur in model c. However, since all the cars contain shared and different components the origin for a failure is hard to find.
We abstract the case of different car configurations as follows:

- We call the software modules/components and hardware that is deployed to a car configuration. Likewise we call the state of modules of a parallel application configuration.
- A configuration ties different components together. These components are the software and hardware modules on a car that are composed in a configuration together. In case of parallel software a configuration is the current state of the modules. We show different configurations in Table 13.2, which associate different components together. Just imagine the different configurations are variations of a car model or different modules that are executed together.
- When a failure occurs, it can be mapped onto a configuration- component line.

Now, we imagine that this table is filled with 100s of cases of configurations and components. We let the parallel software run again and again and save the configuration in the table. We built plenty of cars and save their configuration in the table. When a failure occurs, we associate the error with the corresponding configuration. After we found enough errors, we query the table for the component combinations that lead most often to errors. This way, we gain insight which component configuration is probably responsible for the error.
In Table 13.2 the combination of component 2 and 3 leads to a failure in two times. We can conclude that the origin of the error it is not only component 2, because the error is not occurring in configuration 4, where component 2 is also used. Component 3 is successfully applied in configuration 3. This way, we assume that the errors origin is probably the combination of component 2 and 3. This way, we have a first indication where to start an investigation.

Table 13.2. Sample for Failure Mapping

Configurations/ Components	Component 1	Component 2	Component 3	Component 4	Failure
Configuration 1	X	X	X	-	X
Configuration 2	-	X	X	X	X
Configuration 3	-	-	X	X	-
Configuration 4	-	X	-	X	-

We are careful, not claim our approach as universal, since it is a preliminary idea. Therefore, we see the need for further work to investigate how physical indicators, like bugs and energy consumption of software operated machines can be analyzed with the Hypermodelling approach. We see this case as especially important, because today's car production companies ship millions or varied cars to their customers and the garages deliver inspection results to the manufacturers that are integrated in their corporate memory. Hence, plenty of information for analysis with Hypermodelling is already available in the companies.

13.5.4. Visualization

Our work grasped visualization of source code structure only at the surface. We used the common visualizations of Data Warehouse tools. However, good data visualization is art [174]. Therefore, we see the need for additional research about other visualizations.
For instance, we see the need to investigate the projections and the ideas of Orthographic Software Modeling that inspired our work in more detail. We assume that its model projections can be coupled with Hypermodelling. For instance, an idea can be to enable slicing models or to use other synergies of both approaches. However, further research is necessary.
Other researchers focus on different visualization kinds. For instance, some generate 3D models [251]. Others redevelop radical new concepts for the IDE and put code fragments into bubbles [48, 49] or create zoom capabilities for abstractions [78]. All of those techniques work on their custom

source code access technology. We consider research as necessary if the query based approach of Hypermodelling can be used as underlying technique to ease development for such graphical tools that access source code structure. Future research needs to reveal synergies here, because we see advanced visualization as one of the key features in the future of software engineering.

13.5.5. Related Research

As last vision for future work, we name the necessity to investigate related research and its synergies further. Chapter 12 shows plenty of potential synergies with related work. Therefore, we refer to Chapter 12 as starting point for plenty of further research trails.

13.6. Concluding Vision

Code bases are growing, software is running on more and more devices and the complexity of software development rises. Millions of applications are manufactured today. We are working towards a smart information infrastructure: Smart cities, smart grids, car interactions and billions of people interacting using their smartphones. New methods for software development are needed to face this complexity. We presented the Hypermodelling technique and addressed some of today's challenges. Hypermodelling utilizes big data handling, Data Warehouse technology that is already applied within enterprises. This way, we contribute with two main benefits to software engineering: We reuse an infrastructure that is already available in enterprises and we offer an approach for software engineering that matches the big data size of modern code bases. The reuse of already existing technology makes a mid-term application of our research possible.

Furthermore, data integration is a key demand in information science. Software engineering is complex area that is facing data integration challenges today. Hence, methods and tools from information science need to be adapted to advance software engineering research. Hypermodelling is a first step in this direction by utilizing Data Warehouses and data integration. In the future, we see plenty of related work that can be combined and integrated with our Data Warehouse based approach. Thus, we consider another key benefit of our work in its integration capabilities.

This integration is especially important, because machines run on plenty of microchips. Hence, more and more intellectual company value is glued into microchips. The software industry is not limited to the world of computer scientists any more. Traditional machine manufacturers ship devices with thousands of lines of code to their customers. A company’s value is not purely founded in their products anymore and it is rather transforming to be partly located in the intellectual value of software, where on their machines are running. Hence, companies need efficient tools and methods to do quality control and for their software like they do for any other engineering product. These tools and methods need to integrate in their corporate system landscape. With Hypermodelling source code can be efficiently investigated and different tools for quality control can be integrated on a Data Warehouse platform that is commonly available within enterprises. Therefore, we see Hypermodelling as a possible enabler to advance software development processes in manufacturing companies.

However, the different applications of Hypermodelling open also plenty of potential use cases. Hypermodelling opens new research paths in the areas of data integration, program analysis, development environment applications, code migration, software visualization, code search and code cockpits. We believe that these capabilities will lead to develop new statistic program analysis techniques that complement pure static program analysis in the future.

All together, the research conducted in this dissertation covered different areas of software engineering and evolved into an interdisciplinary technique founded on Data Warehousing. This enables now to use the whole Data Warehouse toolkit for program analysis. Future researchers and practitioners can use our results to develop or use Data Warehouse based code search engines, do code analysis with OLAP means, apply cockpit based code structure audits or utilize any other presented application scenario of Hypermodelling. Additionally, potential synergies with related work and various proposed research trails will probably lead to new applications.

Part V. Appendix

Appendix A. Hypermodelling Cognition

“Madness?”
King Leonidas. 300. 2007

This chapter shares information with:
"T. Frey, M. Gelhausen. Strawberries are nuts. CHASE '11 4th international workshop on Cooperative and human aspects of software engineering. ACM. 2011."[95],
"T. Frey, M. Gelhausen, H. Sorgatz, V. Köppen. On the Role of Human Thought – Towards A Categorial Concern Comprehension. In Proceedings of the Workshop on Free Composition (FREECO) at the ACM Onward! and SPLASH Conferences. USA.2011."[96] and
"T. Frey, M. Gelhausen, G. Saake. Categorization of Concerns – A Categorical Program Comprehension Model. In Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at the ACM Onward! and SPLASH Conferences. USA. 2011."[94]

Abstract. Separation of concerns origins the inspiration how scientists study and understand problems. However, do humans always think scientifically? It is time to consider the human factor in software engineering. We show first indications that our minds do not structure information in an absolutely separated and sharp way. This is done by comparing concerns with research from psychology about categories. Todays program comprehension models lack associations with the paradigm of separation of concerns. Therefore, we present a new and holistic program comprehension model based on categorization studies of psychology. The cognition in this model is influenced by the context, wherein a programmer investigates the code. The comprehension process starts with some ad-hoc concerns that are about to be refined by following an investigation strategy and a vertical process study. Through this study, the concerns refinement may imply an update on the knowledge and the adoption of a new behavior for the investigation strategy. Additionally, we use Hypermodelling to evaluate that main theory that categorization is applied by software developers. We query for details if annotations are distributed randomly and if packages share similarities to categories. This way, we reveal first evidence that supports our theory: Categorization is an essential process in a programmers activities. Now, future research can focus which specific activities in programming correspond to applying categorization.

A.1. Introduction

One of the main motivations to create Hypermodelling is its origin in the paradigm to separate concerns. Dijkstra formulated this principle 1982 [80]. Ever since, software developers and researchers carry this principle with great exultation, like a mantra, in front of them. Though still, concerns in object-oriented programming languages, like Java, are intertwined and often not absolutely separated [87]. Developers face the problem of detecting concerns in a program [195]. Programming languages offer various means to separate and compose concerns [47]. Out of this limitation, we created Hypermodelling.

Right in this chapter, we ask if running after a principle and fighting for a multi-dimensional viewpoint towards software is right. Clearly, the previous chapters show various benefits of Hypermodelling, and a clear support that a multi-dimensional viewpoint of software is full of advantages, but nevertheless we cannot justify that a multi-dimensional perspective is right and natural for humans. We see the emerging need to look beyond our own nose.

In order to credit the role of the human explicitly, we dig into the work of a programmer. Developers realize mental concepts, what we call concerns, in modules. Consequently, programmers face the challenge to comprehend source code. For this reason, researchers created different program comprehension models. These models [50, 240, 234, 160, 208, 163] describe the creation process

of a developer's mental program model. Some of them [160, 234, 73, 209] state that previous knowledge of a programmer is used to gain intelligence about a program. Other models propose that developers use so-called beacons [50, 240, 36, 233, 73] to detect familiar structures in code. Beacons are well known concepts (e.g. method calls) to a developer that are used to derive indications about the code. Hence, the role of such conceptual knowledge in program comprehension is of interest to researchers [213, 51]. Thus, how conceptual knowledge of a programmer manifests in code is a central element for program comprehension [67, 68]. Surprisingly, none of the former referenced program comprehension models addresses the fact that programmers need to comprehend and apply separation of concerns. Hence, a program comprehension model that respects the way of separation of concerns is needed. Current models only consider the fact how code is understood, but not the way how concerns are encoded [240, 68, 221].
Hence, we need to consider the role of human thought and separation of concerns in program comprehension. However, separation of concerns origins the inspiration how scientists study problems. Scientists, still counted as humans, split problems into smaller ones and solve them [160]. We find research that is similar to separation of concerns in categorization (psychology) [221, 175, 176]. Therefore, we transfer research about categorization to the area of separation of concerns. We present similarities between separation of concerns and categorization theory in psychology. Then, we use these similarities to derive a new and holistic program comprehension model. We apply Hypermodelling to do an evaluation of our findings of similarities between separation of concerns and categorization. Again, we show a suitability of Hypermodelling for code analysis scenarios and bring up additional indications that Hypermodelling's capabilities to support flexible and multi-dimensional perspectives is useful. Furthermore, a discussion shows the detailed relation to Hypermodelling. All in all, the contribution of this chapter is as follows:

A.1.1. Contribution

- Our contribution is a holistic program comprehension model based on categorization theory. This model respects research about categorization and separation of concerns. Hence, we enrich program comprehension through the direct association with an area of psychology. We enable future research about the detailed role of categorization in program comprehension.
- We evaluate the theory that categorization is an essential process in software development by presenting statistics about categories in source code. In special, we use Hypermodelling to show that packages in source code share similarities to categories. Additionally. this also shows again, Hypermodelling is suitable for source code analysis and investigations.
- We provide different findings and verify different assumptions about category occurrences in source code. Thereby, we show indications that fields, annotations and method parameters are used criterion by programmers to structure types into the package hierarchy.

Hence, the main contribution of this chapter is an excursus and that a multi-dimensional perspective is natural for developers out of research in psychology. Additionally, we show, again, that Hypermodelling is useful for software analysis. Additionally, we contribute to our work about Hypermodelling by providing insights that a multi-dimensional approach is similar to a program categorization structures.

A.1.2. Reading Guide

First, we reveal a brief introduction about mental categorization in psychology.
In the main part, we compare categorization and separation of concerns and establish a link between the two areas. Succeeding, we use this similarity to establish a program comprehension model. That following, we use Hypermodelling and reveal statistics about categories in source code. Lastly, we close the work about the new theory by giving a brief summary about interesting findings and a discussion.
We finish the chapter, by describing related work about program comprehension and how it results to our approach. Finally, we co conclusions and give a summary.

A.2. A brief Introduction about Categorization

Categorization is an essential part of cognition, and is also fundamental for the process of comprehension. Categories play a central role in perception, learning, communication, and thinking [150]. By applying categorization, objects are grouped together into classes, based on similarities. These classes are called categories or concepts [176, 186].

One feature of categorization is to make quick predictions [255]. These can be derived from limited available information, but can lead to avoiding understanding of an object completely. It would be too time-consuming to comprehend every object totally. This way, it is possible to drive a car just because all cars are similar or to beware of a snake. As the examples show, predictions are used for acting adequately or for adapting behavior [44]. Therefore, the membership of elements, as part of a category, enables the possibility to use prior knowledge, about a category, for new objects, that belong to the same category. Thereby, knowledge about other category members is used to derive features of a foreign object [264].

Category membership enables to use knowledge of a category. New objects are associated with a category and the knowledge of the category is used to assume properties of the new object [176, 186, 264]. To argue in favor of the existence of categorization, it has to be considered that without it, every object or incidence would appear inimitable. Hence, there would be no way to make predictions of new unknown objects. Thus, categories can be seen as a product of interactions, containing perceived resemblance relations in the environment, prior knowledge and context of utilization.

Categories can be formed for physical objects like specific animals, but also for entities that do not exist as physical objects [177]. For instance, democracy represents a conceptual ideal of government. This way, categories classify abstract functionality for a democratic system directly. Likewise, it is possible that categories of physical objects are used to deduce functionality about category members. Thus, no simple distinction between categories and functionality of those can be made. However, categorization is a dynamic process which updates the knowledge of categories, their members and their relations constantly throughout new experiences. Thereby, even minimal prior knowledge has the effect of greatly speeding up learning of category [134]. Likely, there are basic categories, representing important and often used elements, enabling faster processing and association [181].

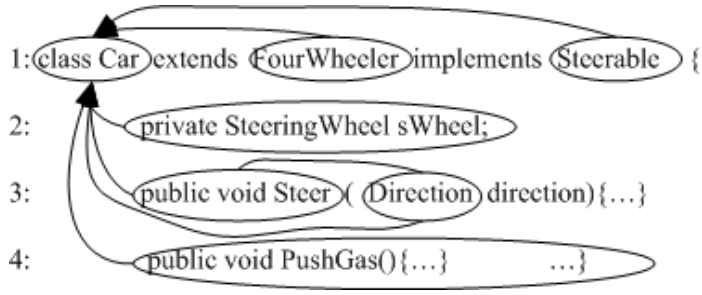
Additionally, there are different kinds of categories. Taxonomic categories represent hierarchies of increasingly abstract categories like terrier-mammal-animal. Script categories are used to group elements that play the same role together. For instance, in the case of breakfast: Eggs and bread belong to the category breakfast foods and are exchangeable. Both can be eaten. Thematic categories group objects that are associated to each other or have a complementary relationship like a dog leash or a clothesline [192]. We humans use taxonomic, script and thematic categories to equally categorizing and understanding objects. This means that every kind of category is used for categorizing objects while none of them is favored. It is also possible to combine different categories and to generate new categories [238, 187]. Also categories can be structured hierarchically. Subcategories have features of their super-ordinate in a certain probability [236].

Categories can be formed spontaneously to fulfill a certain task. These are called ad-hoc categories [44]. For example, things that can be sold in a garage sale can be defined in a spontaneous category. Also, an object can be associated with different / various categories at the same time. This is called cross-categorization. The context an object is seen in influences which category it is associated with [144]. Therefore, the objects categories can be different kinds of category (e.g. the formerly mentioned taxonomic and thematic) [192, 193]. This is important, because research indicates that the speed of association with a category differs with regard to the different kinds of category [222].

Categories are normally formed on basis of features of an object. In case of a car this can be: four wheels, a steering wheel, a gas pedal, a break, a car body, and so on. The categorization theories cover the way how a category is created in the mind based on features. We map this feature definition to vertices of a graph of a programming language. For instance, Figure A.1 shows the class Car. The ellipses mark the code elements corresponding to the graph's vertices. The super class and interface,

fields, methods, and their assignments represent features that are associated with this class. Even more, an object instance of this class would be a vertex. Field values of an instance would also be vertices. Programming languages allow applying separation of concerns with modules and to group vertices together to another vertex. For instance, Direction can be a category itself but also a feature of another one. Thus, we need to think about the difference between a category and the members belonging to it.

Figure A.1. Source code with Category Features



An element can be a category itself or an element of a category. For instance, a pen is an object and a category label at the same time. We refer to a pen as a specific object and also as label for the category. In a shop, we search for the pen category. We buy a specific pen and refer to an element of the category. On the bill, the category is listed, but it refers to an object. Circles that may not ever end can be discussed to discriminate between categories and elements of them. Generally, the thought defines the category, not the object itself. The knowledge that there are many pens, sharing similar features, creates the category. Our work presents theories that are explaining the category creation thought and the association of elements with the category.

A.3. Categorization of Concerns

In this section, we present categorization theories and give examples of similar programming constructs. Afterwards, we present research about category types and cross-categorization.

A.3.1. Three different categorization theories

For classifying objects, three main theories exist how objects are categorized. The distinct theories were created, because some examples are not explainable through one main theory. First, the classic view tried to explain how a human categorizes. Then, the prototype theory was built and, finally, the exemplary view was created. Nowadays, hybrid theories claim that reality is consisting of a combination of theories [35, 151, 239]. We cannot clearly say which theory or which combination represents the whole truth. We can state that experiments support one or another theory. Hence, we present these different theories and compare their essence with example mechanisms to apply separation of concerns.

The Classical View

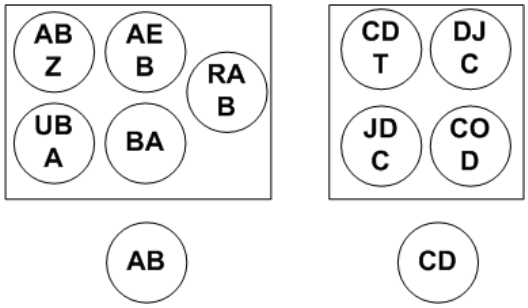
The classical view claims that categories are discrete entities characterized by a set of features which are shared by all of their members. Features are characteristics of a category member. Such characteristics can be physical entities but also more abstract concepts like activities. These features are assumed to establish conditions, which are necessary and sufficient, to capture the meaning of a category. All members of a category possess equal quality to the respective category [55]. In short: all members of a category need to have the same features.

We show an example for the classic category view in Figure A.2. Two categories, AB and CD, are shown in different squares. The circled elements are members of the corresponding categories and the letters within the circles represent different features of an element. All the different members

share the same features (AB or CD), defining the categories. As described in Section A.2, such features are the vertices of the program graph.

Typical examples for such categories in source code are classes. Objects created from a class share exactly the same features, just with different values. Thus, a class can be considered as a classic category example. Another example is inheritance of various classes from a common super class or interface. In such a way all methods and fields are inherited and various extenders share the same feature set.

Figure A.2. Classic Category View

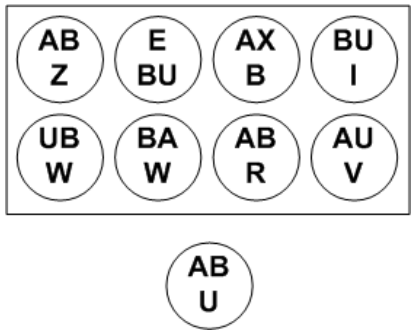


The Prototype Theory

The prototype theory claims that categories are represented by a bundle of features that are typical for a certain category, but not inevitable or sufficient [237]. A category "bird" has features, like "flying" or "building a nest". Even if not all birds got these, like for instance a penguin, they still belong to the bird category. The prototype of a category merges typical features of a category. No exemplar has to match the prototype completely. New objects are classified, out of an affinity with the prototype. In short: A category is defined by a kind of family resemblance where the mother and the father have no shared features, but the children have features of mother and father. All together, they form the category of a family and no member owns all the features of the abstract family prototype.

The theory is visualized in Figure A.3. The prototype is the ABU element where different features occur together in the various elements. We show that not all elements share the same features and there is element existing that realizes the complete prototype. The inclusion of characteristic features illustrates, why some elements are perceived as typical instances of a category, in comparison to others. Characteristic features of a category are abstracted during learning and merged as a representation of a prototype representing the category. Thus, all typical features are associated with the prototype. Hence, the prototype is a representation of an amount of objects that share similar features [221]. Therefore, the category definition is in this case not sharp any more.

Figure A.3. Prototype Theory



In code, this theory can be realized through annotations. For instance, absolutely different classes can be marked with various persistence annotations in Java. Typically, developers can choose between various annotations that are indicating persistent features. Developers can map classes and their fields directly to tables and columns. There is no necessity that all classes need to be marked this

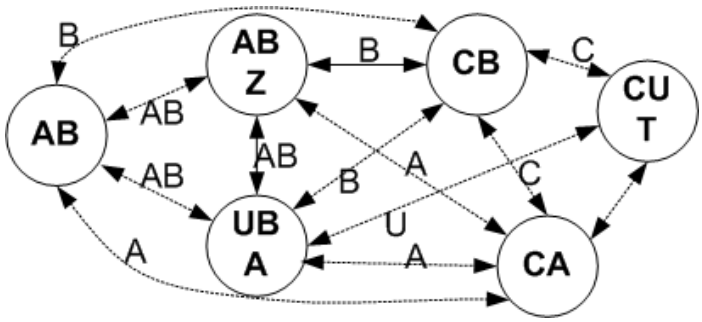
way. Thus, the annotations used can differ for various classes. Even though, developers associate all classes with such annotations with the persistence category. The classic view is not sufficient to explain the phenomenon, because annotations can differ for every class. We consider the prototype theory as explanation. A prototype class serves as mental representation for the developer. This prototype contains all persistence annotations. Developers compare this prototype with other classes to decide if they are belonging to the persistence category.

The Exemplary View

The exemplary view assumes that in contrary to the prototype theory, single exemplars are engrained together with the category denotation. Each new exemplar represents a category on its own. By recognizing a new exemplar a learner compares it to already known categories. The learner assumes that an object might have similar features as the exemplar compared to which it has the most similarities. Thus, similarity comparisons are made with the exemplar itself and not with an abstract prototype. This way, a certain animal might be categorized as a rodent, because it reminds of a mouse, whereas another animal is categorized to the same category, because it reminds of a squirrel or a chipmunk [52].

We visualize the exemplary view in Figure A.4. Objects are depicted by their features and arrows connect same features of exemplars. Every exemplar is a category on its own and just the features remind of another object. The AB exemplars remind stronger of each other through two shared features. If any exemplar is recognized, the exemplars that share the most features are taken as reference to derive further features or functionality of the object.

Figure A.4. Exemplary View



The pointcut-advice (PA) mechanism of Aspect-oriented programming allows in many cases to avoid modules with multiple concern associations. Also, the Hyperslice / MDSOC approach tries to avoid such kind of overloaded modules. Generally, the Hyperslices follow the idea to encode every concern in its own module. Modules, encoding only one concern at a time, are similar to the vantage point to regard every exemplar on its own. Every resemblance to other modules is just defined because of shared features. Hence, we see Hyperslices as similar to the exemplary view. The PA mechanism enables to group program elements together and apply advice code at them. The grouped elements share common features like network accessibility or security. Thus, we classify the PA mechanism as hybrid similar in between the exemplary view and prototype theory.

Mapping Theory and Separation of Concerns

We see the means to apply separation of concerns and categorization as similar, because of previous described comparison. In Table A.1, we show mechanisms to apply separation of concerns and the categorization theories that share similarities. The different values in the rows present the mechanisms of the before described examples and their mapping to the columns.

Table A.1. Similarities of Theories and means to apply Separation of Concerns

Classic	Prototype	Exemplary
Classes-objects	Pointcut - Advice	
Inheritance	Annotations	Hyperslices

A.3.2. Types of Categories

Besides the three theories, different types of categories are used to group objects together [188, 192]. Following, we explain these types of categories and describe a cross-classification.

Taxonomic Categories

Taxonomic categories represent hierarchies of increasingly abstract categories. An example for a taxonomic structure is terrier-mammal-animal. In case of programming, this kind of categorization is solved by specialization through inheritance. Thus, we see inheritance as a representative for taxonomic categories. Another taxonomic structure is represented through package hierarchies. We see this as similar to the taxonomic category kind, because the elements within a package normally have a somehow related meaning and packages are arranged in taxonomy. Thus, we see the hierarchic structure of the packages as taxonomic category, but the membership of elements in the distinct package itself has to be described in a different way.

Inheritance of multiple classes or interfaces is often possible in programming. Therefore, only class inheritance with one super class is a representation of this type of category.

Script Categories

Script categories are used to group elements that play the same role together. Such elements are interchangeable within the role. For instance, eggs and bread belong to the category breakfast foods and are exchangeable. In programming, classes that are implementing interfaces, are interchangeable and offer the equal functionality. This is similar to script categories because all the classes that implement an interface are interchangeable.

We note that classes can implement multiple interfaces. We assume that interfaces can also represent categories. This way, a class can belong to multiple categories. Script categories do not explain the multi-category assoc. Later, cross-categorization is proposed as explanation.

Another case are object instances of a class. All of them offer the same functionality. Routines in code that work with one object also work with the other instances. All objects of a class are interchangeable. Thus, we see a mapping to script categories, too. Single Annotations can also partly be considered as a typical example for script categories. Classes can be marked persistent. The Classes can differ absolutely in their meaning and also in their functionality but they are interchangeable; any class can get a persistence annotation and is saved with the same routines within an application framework.

Thematic Categories

Thematic categories group objects together that are associated with each other or have a complementary relationship. Such a category can be a clothesline. Clothes and lines are distinct categories. The assoc forms the new category of a line to hang clothes on it. Nobody would talk about a line to hang clothes on it. Everybody understands the combined term of a clothesline as its own category. Such a grouping of elements is done through packages. All classes and interfaces in a package belong to a certain theme and are connected. Compositions are a classic way of combining. For instance, a class uses multiple instances of other classes and combines these to derive a new functionality. Another often used way for composition is implementation of multiple interfaces, which concentrates the functionality of different concerns at single point of code. This code part is then probably recognized as its own category.

Mapping Categories to Separation of Concerns

We present the described categories types and similar separation of concerns mechanisms in Table A.2. To respect the exemplary comparison, done before, several mechanisms are appearing

multiple times. Columns indicate the category type, and rows represent their corresponding mechanisms. For instance, the package mechanism is used in two ways. Interfaces are appearing in script and in the thematic column, too. Advices and Pointcuts are added. The PA mechanism itself is not hierarchical and therefore not taxonomic. It can be used to affect totally distinct and interchangeable objects with concerns like security or logging. This is normally done through the application of Advices at various Joinpoints that are selected by Pointcuts. This seems similar to script categories. However, Advices add functionality or alter behavior of code, which is similar to a composition (thematic).

Table A.2. Similarities to Category Kinds

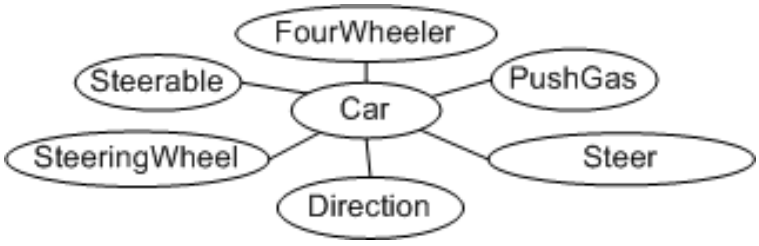
Taxonomic	Script	Thematic
Single inheritance	Pointcut - Joinpoints	Advice
Package-hierarchy	Classes-objects	Package-members
-	Annotations	Compositions
-	Interface implementation	Multiple inheritance

Cross-Categorization

As indicated by composition of elements before; an object can be associated with multiple categories at the same time. This is called cross-categorization. An example can be an “Account” class in programming that is also persisted in the database. Hence, "Account" represents a domain concept and a persistent entity in a database. Additionally, it can also represent an object, requiring secure access. However, programmers refer in all cases to it as the “Account” type. The actual meaning, may it be class, database representation or object, is discriminated through the context. Research in psychology shows that cross-categorization is something absolutely natural [144]. Objects belong to different categories at the same time. The context influences the actual meaning and the inference about the way of operation of an object.

An example for cross categorization in programming is given in Figure A.5, by representing Figure A.1 in a different graphical view. The Car class is belonging to its different vertices (categories) at the same time. It is a FourWheeler, Steerable, driving in a Direction and so on. All cars can be classified into this categories. We compare this with cross-categorization, where an object belongs to multiple categories at the same time. Here, these categories are just concerns. Thus, we see the association of a source code fragment with multiple concerns as the same like cross-categorization.

Figure A.5. Multi-Category Associations of Figure A.1



A.4. Category-based Program Comprehension

We use the prior similarity of categorization to separation of concerns to describe program comprehension in the following. We present different sections that describe a new program comprehension model based on research about categorization and programming.

A.4.1. Characteristics of categories

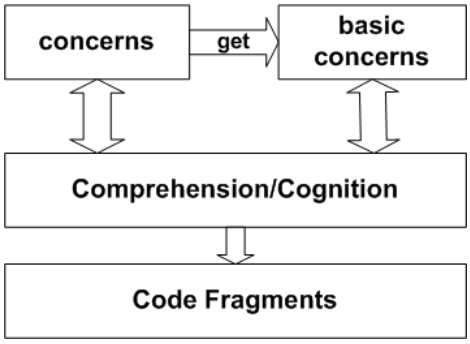
Categories are essential for the comprehension of specific objects and discrete conceptual entities [177]. In source code both of them are occurring, too. There are concrete mechanisms to

indicate concern assocs of elements. Such mechanisms are packages, classes, inheritance, methods, annotations and similar means. Other concepts are realized by compositions of various programming constructs. Design patterns are examples of such compositions [106]. Developers associate the elements of realizing concepts with the corresponding design pattern. The functionality of these elements has discrete, flexible characteristics whereas not every implementation is completely equal to another one. Thus, the category of a design pattern is more like an abstract concept than a concrete mechanism. Categories manifesting in code are similar to physical object categories and abstract concepts like democracy. Hence, there are concrete mechanisms in programming to realize categories and composition mechanisms to realize elements of conceptual categories.

The comprehension of fragments seems to be dependent on the experience [218], and thus probably on the knowledge of a developer. That is also the case with categorization. The learning of categories and also speed of comprehension differs, depending on prior knowledge [134, 181]. We find supporting arguments in research about program comprehension: Some source code lines are obviously more important than others [73]. We see these lines as a representation of categories or features of categories. To categorize the fragment, features (i.e. those lines) are recognized and used for category association. Another possibility is that the line itself represents an element of a category that is recognized. We claim that concerns are acquired and accessed by studying programs. This is similar to how categories are learned. Therefore, we assume that the importance of a concern and the frequency how often it is studied, influences the decision whether it is recognized as a basic concern. This distinction is made to acknowledge the existence of basic categories. As with categories, the comprehension of basic concerns is gained quicker than of normal ones. We consider framework classes as an example for often used elements. Therefore, we assume that framework classes are an example of basic concerns.

We present the concern learning process in Figure A.6. Developers comprehend code and use basic and normal concerns. We visualize the access of the code with the arrow from comprehension down to the code fragments. A developer accesses a concern every time it is studied. Thus, the usage frequency of the category increases. This means: Just studying the source code can lead to classifying a concern as an important one and it can be understood faster when it is occurring again. We visualize the continuous usage of concerns for comprehension as well as their update by double-sided arrows. Additionally, the importance of a concern and its frequency of usage is refining a firstly recognized normal concern into a basic concern. No knowledge of the source code exists, when a code study begins. But as the knowledge grows, certain concerns can get important ones. We present this transfer by an arrow from ‘concerns’ to ‘basic concerns’.

Figure A.6. Learning basic Concerns



A.4.2. Usage of prior knowledge for prediction

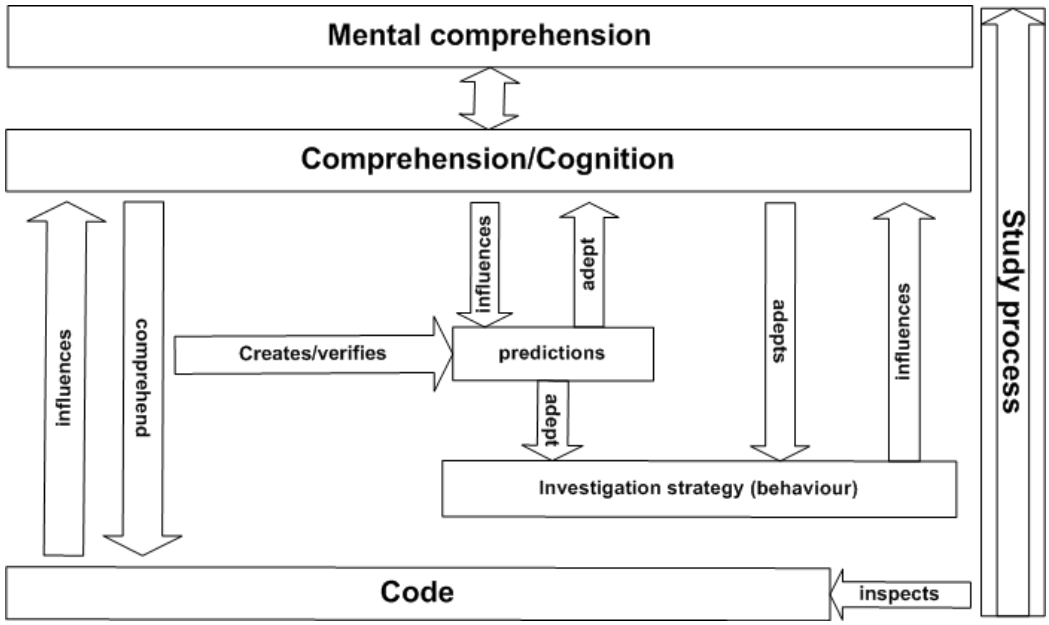
Developers predict functionality of source code with the help of prior knowledge. This is obvious, because often recurring elements are used in a program. It is assumed that a developer creates hypotheses about the functionality of a program by using prior knowledge [50, 160, 234, 73, 209]. That usage of prior knowledge for comprehension seems also to be the case in categorization

[264]. We believe that building a hypothesis about a program is equal to predictions of category membership [255]. Hence, the prediction about the functionality comes from categories. A prediction leads to adequate behavior. Such a behavior can be a change of the investigation strategy. In categorization adapting behavior is an important appliance [44]. Thus, it seems equal to the code inspection process.

We developed visualization for the process of predictions and behavior. We present it in Figure A.7. With respect to the program comprehension scenario, the behavior is called investigation strategy. On the left side, the investigation process is vertically visualized. It is arranged in a box with an arrow to express the continuous process. All actions shown happen within this process. The arrow directions visualize the way of interactions between the elements: Code is studied and comprehended through the influence of known concerns. The comprehended code influences the cognition and thus the comprehension of further fragments. For example, new concerns can be learned by studying code and thereby the cognition is changed. Comprehending code leads also to predictions of the features of the code under the influence of known facts. By building such a prognosis, the current state of comprehension influences the prediction. Because of predictions about studied source fragments, the investigation strategy can be adapted. The strategy also influences the cognition of a studied code fragment and is adapted by the current state of comprehension. It is a mutual influence: In the study process this can happen multiple times and when the comprehension adepts the investigation strategy the perception of newly studied elements will be changed. Anyway, sometimes wrong or correct predictions are made. In such an ongoing process of comprehension, predictions can be verified as being wrong or right. It is important to note that this strategy does not claim to be verifying predictions continuously. A direct verification is probably happening sooner, when uncertainness about the predictions is at stake. Likely, in a normal study process, the predictions are, in a first step, verified by the assumption that they are true, but further investigation might support the falsification of them.

The verification of these predictions is then used to update the knowledge about concerns. Such an update can be the proof or falsification of the predictions that are associated with a concern. Since these concerns influence the cognition, the strategy can be changed again. For example, the recognition of a code fragment can lead to predictions about its functionality. Further investigations can then lead to supporting or falsifying the previously made assumptions. Because of this, the knowledge of the concern has to be updated. This leads to new predictions that can lead to a change of strategy in investigation.

Figure A.7. Prediction and Behavior



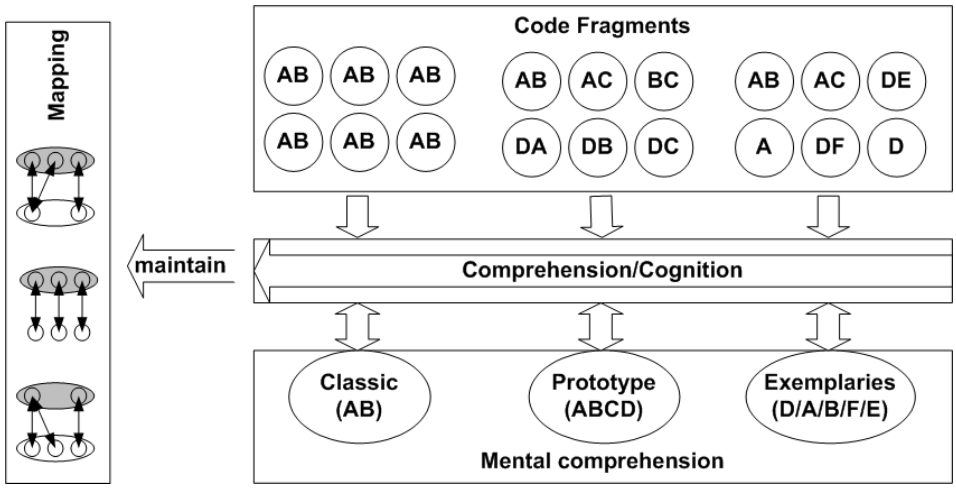
A.4.3. Different means to apply Separation of Concerns

All over, we see source code and the means of separating concerns as being similar to the different theories of how categories are arranged. Our main point is that means of separate concerns are similar to categorization theories; elements of source code are qualified and grouped into categories through them.

We present a comprehension process and present its visualization in Figure A.8. Concerns in the source code are shown (Code Fragments). The single sided arrows visualize that code is comprehended and studied. The letters represent the concerns and the circle around them represents a module. The different means of separating concerns and modules can be mapped to the corresponding cognitive representation. The mapping box shows different possibilities of mapping. This mapping between the mental representation and the realization in source is maintained continuously during the study process. Therefore, the arrow in the box indicates the continuous comprehension of a developer. The small arrow pointing to the mapping box, indicates the constant mapping refinement during the procedure of comprehension. In the mapping block the different circles and arrows show different variants of mapping. We symbolize this mapping by arrows from the mental space (grey) to code mapping (white). The grey circles in the mapping are concerns and the white code circles are modules. Not every concern is realized in a single module and sometimes modules represent multiple concerns. The reason for this are limitations in separating concerns in the source code and mistakes of programmers.

The arrow across the comprehension box symbolizes the continuous process of comprehension and shows that concerns will be recognized differently. At their first occurrence they will probably be recognized as a single exemplar. For instance, a class is seen for the first time. Since a developer does not know other occurrences of this exemplar just a single model is created as mental representation. As more and more code fragments appear, using the class, a developer is reminded of the other fragments. This way, an abstract prototype is used for representing this mentally (prototype theory). The developer assumes that diverse reoccurring features can be used for discriminating the category. If all the features of the studied objects are the same, a category following the “classic view“ can be used, instead of an abstract prototype. Thus, it depends on the features if the category of reoccurring elements is based on the prototype theory or the classic view. We show this use of formerly seen exemplars with the double ended arrows. Additionally, we show exemplary code fragments, where the letters describe features of the fragments. Vertically, at the bottom, the different corresponding categories are arranged.

Figure A.8. Concern Theory Mapping



A.4.4. Multi-Category-Association

In software development, maintenance tasks appear, as for instance, fixing bugs. Dynamically multiple elements of the code are associated with such a task [139]. Likewise categories can be

formed dynamically [44]. Hence, this indicates that program elements can be part of a spontaneous category. Similar is the multi category membership in categorization. Elements can be part of script, thematic and taxonomic categories at the same time. For instance, a class belongs to a certain package, but in the same moment it can also be a member of an inheritance hierarchy and can be affected by annotations. The actual comprehension depends on the intention and context of a developer when he studies a fragment. This can be compared with polysemous words where the actual meaning is driven by the context [144]. Thus, the association of a source code fragment with multiple concerns is the same as cross-categorization [192, 193].

We present our view of a multi concern assoc in Figure A.9. There code fragments are associated with different concerns. The concerns are symbolized by the letters and the code fragments by the circled letters. The fragments belong to different concerns at the same time. We visualize this by the concern letter associations.

Figure A.9. Cross Concern Association

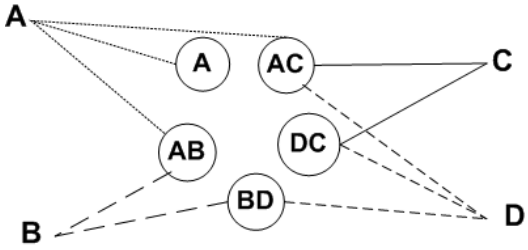
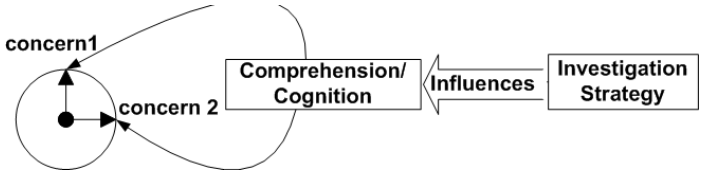


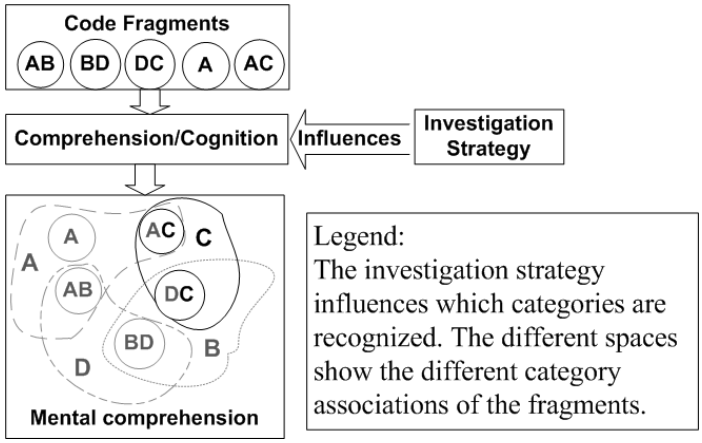
Figure A.10 shows a single fragment that belongs to two concerns. The comprehension which concern is recognized is influenced by the context in which the fragment appears. For the study process, this context is the investigation strategy. Thus, the investigation strategy influences how and which concerns are comprehended.

Figure A.10. Cross Concern Comprehension



In Figure A.11, Figure A.9 and Figure A.10 are combined and shown with multiple fragments. The discovering of varying concerns through different investigation strategies is visualized by the example of grey and black concerns. The framed spaces represent the concerns. As shown, a fragment is framed by multiple lines to indicate the multi- concern fragments. In short, a fragment is consisting out of various concerns.

Figure A.11. Cross Concern Association Comprehension



A.4.5. Composition

The composition of code elements seems similar to the combination of categories to build new categories [238, 187]. Hence, we assume that new concerns can be created through the composition of other concerns, what happens quite often in source code; for instance, by creating data-access-objects (DAO). Such classes are clearly to be counted in the persistence layer. At the same time these classes are composed of domain objects that normally represent business entities. Therefore, it is wrong to associate a DAO only with the persistence layer, since they also belong to a distinct business domain in the application itself. Thus, a DAO is a composition of a business domain concern and a concern of persistence.

Developers often deal with abstract definitions and concrete implementations. We believe that a class can represent a concern. But when it is used in this way (e.g. as instance object) then it is like a feature of another element. This reveals problems of distinction between a concern itself and its instances. The different categorization theories offer no answer about categories and their members. Hence, for the sake of simplification we don't distinct between concern or category instance.

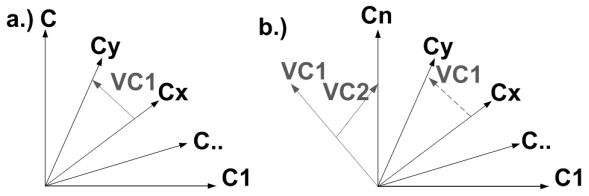
In order to understand a composed concern, a developer needs to know the underlying concerns. Another option to determine the way of operation of a composed fragment is by meta-information associated with the composed concern itself. Such meta-information can be realized by comments or a meaningful name of the fragment.

Compositions are quite common. We assume that compositional concerns are only recognized as concerns by a developer, when their meaning is quite clear. Not every composition will be recognized as a new concern. The usage frequency of a composed concern is an indicator if it is comprehended as concern itself. If a composition is never used, it will most probably not be recognized as a concern of its own; likely it will be comprehended as pure composition, but not as a concern.

Figure A.12 shows a composition of concerns based on concerns. The concerns are named C and the composed concerns are named VC (virtual concern). VC1 shows a composition of two normal concerns. The a. part of the graphic shows the construction of the first composed concern (VC1 – composed of Cx and Cy). The b. part shows the construction of another composed concern on top of a composed concern (VC2). To visualize, VC1 is equal to any other concern, a shifted VC1 is also shown in Figure A.12 b. We assume that VC1 will be recognized as concern, because it is used by VC2. The developer needs to know it to comprehend VC2. Maybe VC2 will not be recognized as a concern itself because it is not used anywhere else.

Anyway, we see this as a simplification of the real world where some compositions (VC1) get used more often and others do not (VC2). The usage of VC1 in VC2 in Figure A.12 is just one example for multiple usages within other concerns. Developers recognize such often used composition concerns like VC1 because of their reoccurrence in code. This is supported by the prototype theory. There, categories are learned via the occurrence of elements that are similar. Developers recognize the occurrences of a composition as category members. Thus, every occurrence of the composition in use is associated with the composition itself and the composition evolves into its own category.

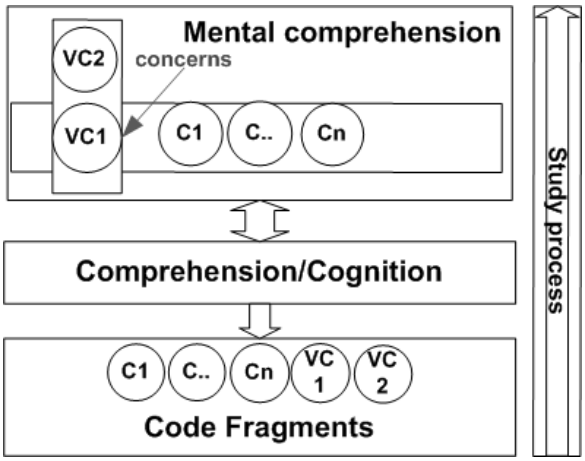
Figure A.12. Concern Composition



We visualize the integration of the compositional concerns in Figure A.13. The code fragments contain compositions that represent compositional and normal concerns. A continuous study process is shown. We show the mental representation of a developer wherein the composition is expressed by the vertical block. All normal concerns (C1…Cn) are arranged horizontally. VC1 is a member

in the horizontal and the vertical block at the same time. We do this out of respect for the previously discussed fact, that a composition can be recognized as a normal concern, as well. Anyway, it has to be made clear that there can be multiple levels of composition and this crossing has to be seen as an example of one of them. Developers build up the knowledge of the compositions throughout the study process. They can recognize compositions first as a pure "horizontal" concern. A further study process can then refine it to recognize it as composition. For example when VC2 is comprehended first, probably VC1 would get recognized as a normal concern and part of VC2. When it is recognized later that VC1 is also a composition, then the knowledge of it would be updated.

Figure A.13. Concern Composition Comprehension



A.4.6. Putting it all together

We present the concern study process in Figure A.14. The cycle expresses the continuous process of comprehension of a program. The arrow visualizes how the knowledge of a developer is built up and comprehension grows. The direction of the arrow also indicates the permanent level of knowledge. We do this to acknowledge that concerns can be ad-hoc - concerns, named after the ad-hoc categories. Such a-hoc concerns can evolve into permanent ones depending on their importance and occurrence in the study process. Finally, concerns can be basic concerns. Such basic concerns have a high frequency of usage or a high importance, as described in Section A.4.1.

Figure A.14. Study Process

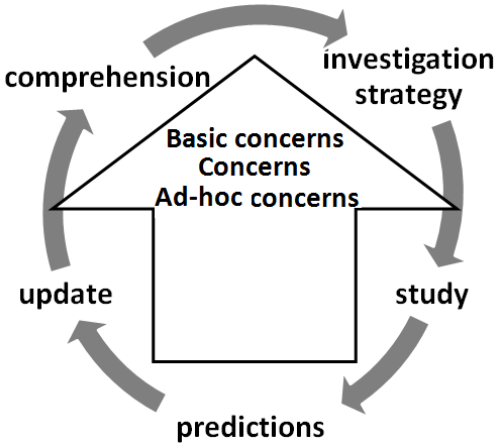
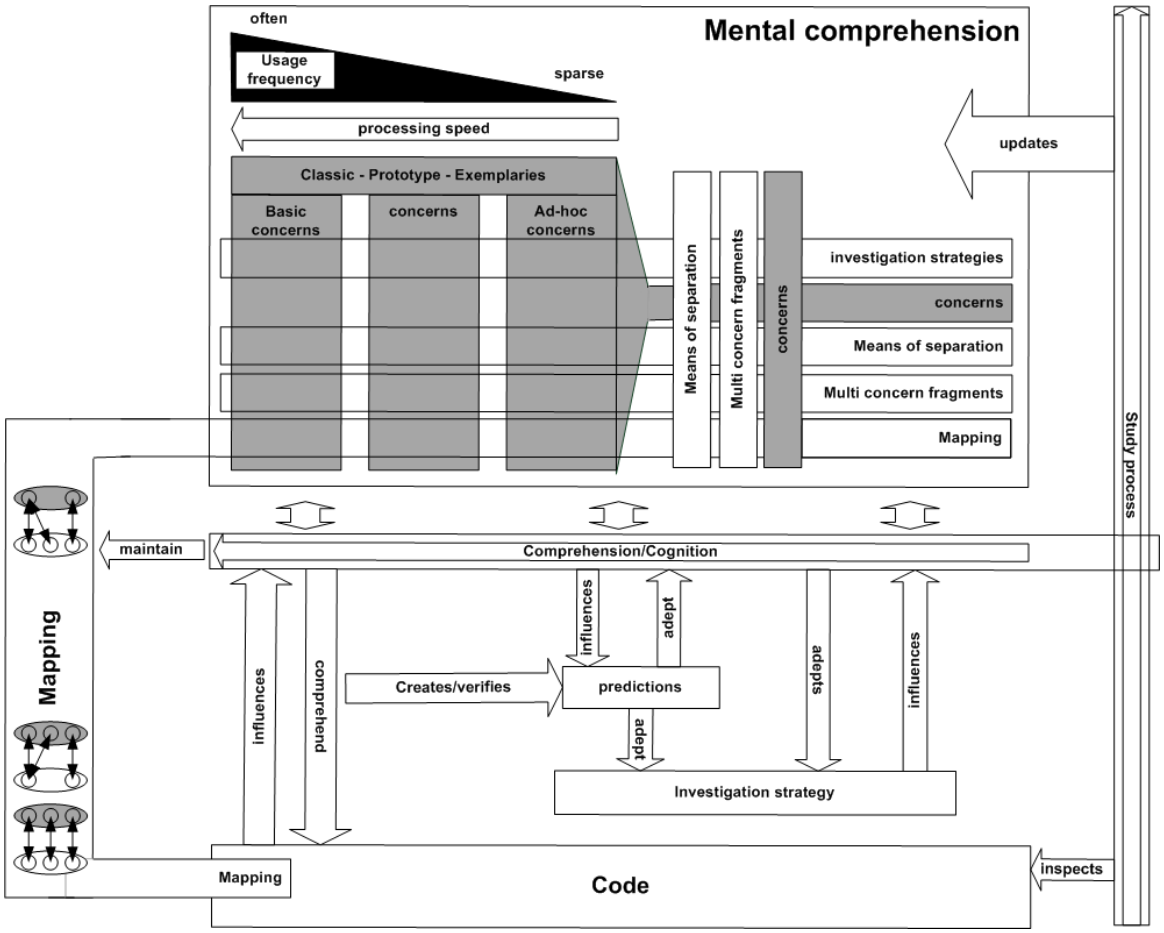


Figure A.15 assembles all facts, from Section A.4.1. - Section A.4.5. The two axes, study process and comprehension/ cognition, show the main actions that happen in program comprehension. The boxed arrows indicate the continuous comprehension process. All elements in the range of the

arrows are affected by the process of comprehension. The study process crosses cognition to indicate the consistency of the comprehension throughout a study. Vice-versa, the comprehension process crossing the boundaries of the study process shows the holistic approach of comprehension, which is not limited for a specific study process. Moreover, even multiple study processes can occur.

Figure A.15. Holistic Comprehension Model



The mental comprehension contains different representations of the concerns. They are refined during the study process and influence the cognition. Mappings between the mental representation and the concrete program elements exist and are maintained during the comprehension processes. The concerns are represented by the various theories (Section A.4.3). Below, the concerns are separated into the different kinds of basic, normal and ad-hoc concerns. The ad-hoc concerns are used for a specific task as described in Section A.4.4. The basic ones are important and often used concerns as described in Section A.4.1. The comprehension speed and usage frequency is increasing from ad-hoc concerns to basic concerns.

The bars crossing the kind of concerns symbolize the different other facts from this section. The grey vertical boxes represent the different types of concern (exemplary, basic, normal). The concern box is colored in grey to indicate it is a simplification of the different theories that are also colored in grey. The block that is showing the Means of Separation indicates that a concern can also be associated with the way the separation as it is manifested in the source code. The block named Multi concern fragments, indicates that a fragment can be associated with multiple concerns at the same time. We show a block of investigation strategies. It was added to Mental comprehension, because we assume that the experience of a developer influences investigation strategies. Successful investigation strategies are somehow associated with concerns. Thus, there must be an assoc between investigation strategies and discovered concerns. Research, e.g. [218], supports that investigation strategies vary, depending on different skills of developers.

The vertical blocks ‘Multi concern fragments’ and Means of separation visualize compositions. However, the concern is enhanced with the other vertical elements that indicate the different association.

In Figure A.8 mappings of source code in relation to the mental model are shown. The “Mapping” has been extended to visualize that they can also be associated with all the previously discussed elements and that they are to be seen in the Mental comprehension as well as in the code. Through this combination of the mappings it is also possible to contain the means how concerns are separated and associate them with the Mental comprehension. Like before, the small symbolization in the mapping bar indicates that different mapping variants exist.

As discussed in Section A.4.2, different actions may take place. The investigation strategy is adapted. Predictions of functionality of source code fragments being studied are either verified or falsified and the mental knowledge is created and updated. Generally, all the arrows represent the same relations as described in the whole section.

A.5. Evaluation

Right before, we introduced categorization as an essential process of a programmer's activities. Now, we use Hypermodelling to query for facts in source code to reveal if category similar constructs appear in code. In oder to verify if mental categorization manifest in code, we investigate how a few programming constructs are applied. In detail, we regard how the programming mechanisms are used to distribute features among programming structures and if those are used to group elements like categories together. If we can find such grouping structures in code, we have an indication that categorization is applied during programming and get support for the theory that programming constructs are used to manifest mental categories in code.

An obviously similar construct in programming to categories are packages. In the comparison to a category, packages have elements within them such as classes and interfaces, what we just call types. Those types have features, like method parameters, field variables and annotations. Furthermore, packages get often shown in hierarchies, whereby the hierarchies have no structural meaning for variable and method scope. Likewise, the developer is free to apply annotations where desired. Hence, our question is: Are packages used like categories? Can we find similar structures within packages, like researchers found about categorization in psychology?

In order to reveal answers to those questions, we investigate different perspectives of packages. We do so to get a holistic viewpoint of different indicators if packages share characteristics with research about categorization. In order to investigate similarities occur, we define different assumptions that we test in this evaluation. If all this assumptions are verified by in investigation, we have a strong indication that categorization is an essential process in programming and for our comprehension theory. The main assumptions (MA) are as follows:

- MA1: Annotations are one mean to attribute programming constructs. This seems similar to features of a category. Hence, we assume that annotations are not distributed randomly over packages, if packages are similar to a category.
- MA2: Types, fields, method parameters are program vertices like annotations. Therefore, they are also similar to features of categories. We assume to find first evidence that the same features (types, fields, method parameters) occur more likely within a package then outside.
- MA3: Research about categorization uses fuzzy boundaries of categories and different theories address that the amount of shared features as fixed or as fuzzy. We assume to reveal indications about a criterion that sharpens the boundaries of a package as category.
- MA4: Category hierarchies matter in case of categorization. Package hierarchies have no scope meaning in programming, but we assume to find indications that hierarchic categorization is used as well in programming.
- MA5: Research about categorization argues for different kind of categories. Methods, fields and annotations are used in completely different context, like the different category kinds. We assume to find feature category similarities for all of those.

All in all, the contribution of this evaluation is to provide first indications for the verification of these assumptions. We provide sufficient evidence to emphasize that categorization is used within program construction. Further research needs to verify deeper and advanced details. Out of this, future tools can be used to identify code structures that are wrong categorized out of category construction research.
We start the investigation by showing a simple investigation about the distribution of annotations (MA1). Thereby, we give a step by step example how a researcher might dig into detail to do conclusions or built new hypothesis within an Hypermodelling analysis. Then, we get into the specifics how we compute indicators for the field type distribution in packages. We apply those indicators by presenting the field distributions in packages (MA2). We follow this trail and advance it through certain restrictions. We do this to clarify if the restrictions can provide a criterion to sharpen boundary of packages (MA3). Then, we show how package hierarchies matter (MA4). After all, we determine if we get similar findings for method parameters as features (MA5).

A.5.1. Annotation distribution

Annotations have been introduced in Java version 5 [46] and get more and more used within actual programs. Some annotations express structural meanings, like deprecated elements, and others are used to weave additional functionality, like security, into a program. Therefore, our goal now is to verify MA1 and to determine that annotations are not distributed randomly over packages.
However, we start the investigation about annotations with the petclinic demo application.[1] Mainly, the application is a layered web application consisting of 31 Java files, containing application logic. It is making use of declarative transaction management, database access, and aspect oriented programming paradigms.
In the following, we first show an example report about the annotation distribution in the demo application. Then, we dig into another, larger, application and determine how annotations are distributed therein. Lastly, we do conclusions about the investigation.

Report of a demo application

Right here, we show the aggregated annotation distribution to packages of the petclinic demo application in Table A.3. The associated visualization of the table is Figure A.16. Annotations of the Java language specification like override or deprecated, have been excluded from the visualization of the report. This enables us to focus on annotations with a clear functional meaning.
Annotations are grouped together to the packages in which they are defined. For example, @Transactional is defined in the *.transaction package. The @Repository annotation belongs to the *.stereotype package. However, the abstraction to the package level is done to show that packages of the annotations also matter.

The wildcard '*' is used to indicate parent packages. Their name was stripped for readability issues and our focus on the innermost subpackage, in which the annotations are finally defined.

In the following, we interpret the report (Figure A.16, Table A.3). It excels that one package a has just xml annotations and no other ones (a). The reason is that the package represents the domain model of the application, as it can be verified by source code inspection. Especially sticking out is package e, where a heavy use of annotations out of the web package is done. Thus, the conclusion can be done that the package is responsible for the web access of the application. Clearly, also the meaning of the aspect package f., to contain aspects, can be derived from the consumed annotations. Package b, c and d make heavy use of transactions. This indicates their functionality has to be referred to the persistence logic of the application. In fact, these packages seem quite similar. Source inspection verifies this theory; these packages realize the access to a persistent data store with different data access technologies and are interchangeable within the application.

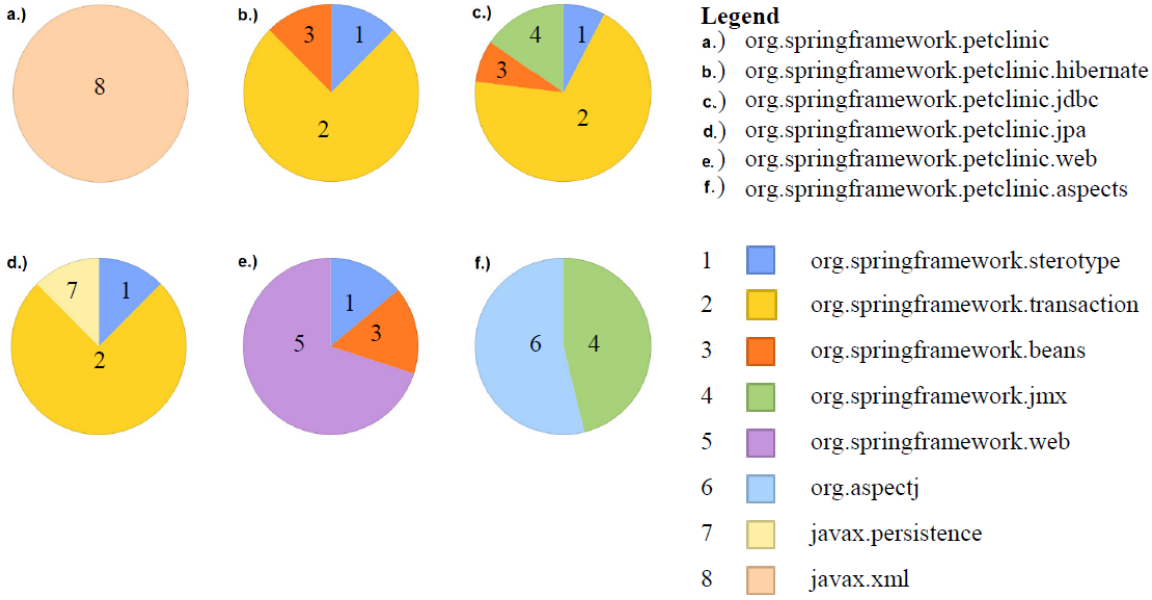
[1] http://static.springsource.org/docs/petclinic.html 7.12.2012

All together, it seems straight forward to see what a package is used for within the application, just out of the used annotations. Thus, there is the assumption that this is maybe the case in other applications as well. At least, we can assume that annotations do appear in a structured way within source code. In the following, we are going to investigate this.

Table A.3. Annotation Distribution in a Demo

package	All	*.stereotype (1)	*.transaction (2)	*.beans (3)	*.jmx (4)	*.web (5)	*.aspectj (6)	*.lang (excluded)	*.persistence (7)	*.xml (8)
org. springframework. petclinic (a)	5	-	-	-	-	-	-	2	-	3
org. springframework. petclinic.hibernate (b)	11	1	6	1	-	-	-	3	-	-
org. springframework. petclinic.jdbc (c)	13	1	9	1	2	-	-	-	-	-
org. springframework. petclinic.jpa (d)	11	1	6	-	-	-	-	3	1	-
org. springframework. petclinic.web (e)	53	7	-	8	-	35	-	3	-	-
org. springframework. petclinic.aspects (f)	13	-	-	-	6	-	7	-	-	-

Figure A.16. Report about Annotations Usage



A report how annotations that are defined in different packages (1-8) occur at types and methods within packages of a demo application (a-f).

Annotation Distribution in a large Project

Out of our prior indication that annotations can reflect the meaning of package, we dig into details. In order to have a more valid sample we use the alfresco project that we already described in the prior chapters. The amount of types in our indexed alfresco source code is 3462. The total amount this types members is 42823 (members=methods+fields). In total, the types and members contained the amounts annotations that we present in Table A.4. There, we also indicate the origin of annotation. We see that annotations are defined in the Java archives, in the alfresco project itself and the Aspect

archive. Like we show, the most used annotations are @Override and @SuppressWarnings. Both belong to the Java libraries. The @Auditable annotation is out of the alfresco project itself and occurs quite often. The rest of the annotations is not distributed frequently. At fields, not much annotations occur, like we show in Table A.5. At the Type level, annotations are sparely used as Table A.6 indicates. All together, We see that @SuppressWarnings is the most common annotation in the whole project.

Table A.4. Annotated Methods

Annotation	Occurrence count
Override (Java)	2067
SuppressWarnings (Java)	425
Auditable (Alfresco)	387
NotAuditable (Alfresco)	249
Deprecated (Java)	3
Test (Java)	3
Before (Aspect)	2
After (Aspect)	1
BeforeClass (Aspect)	1
XmlElementDecl (Java)	1
Total	3139

Table A.5. Annotated Fields

Annotation	Occurrence count
XmlElement	11
SuppressWarnings	20
XmlAttribute	14
Total	45

Table A.6. Annotated Types

Annotation	Occurrence count as classes	Occurrence count at interfaces
SuppressWarnings	20	-
Deprecated	3	1
XmlAccessorType	12	-
XmlType	12	-
XmlSeeAlso	2	-
XmlRegistry	1	-
RunWith	1	-
PublicService	1	-
Total	69	

We regard the distribution of the various annotations in packages. We want to know if there is a difference between the most common used annotations in the packages. Therefore, we query which annotations occur in which packages in an alphabetic ascending order and show an excerpt of the result in Table A.7. There, we see the occurrence of @Override and @SuppressWarnings in packages. We not that we do not sum up package hierarchies within the computation of the query result. We do so, to concentrate purely on the directly in a package occurring annotations and not on child packages.

Anyway, two additional annotations are not shown in the query result excerpt, since they did not appear in the first packages. Therefore, we imagine that the table has more columns and more rows.

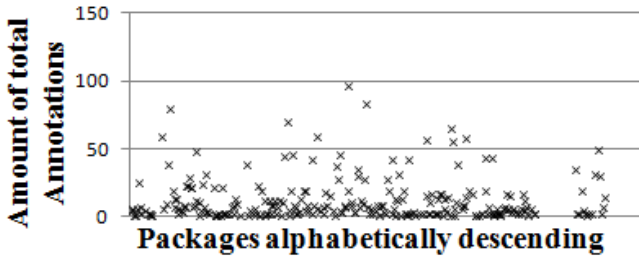
All together, there are 364 packages used as source data. The rows and columns are used to populate Figure A.17 to Figure A.20. In those figures, we see that the x axes show the different packages that are ordered alphabetically and the y axes counts the occurrence of annotations.

Table A.7. Excerpt of Annotations in alphabetic ascending Packages

Package	Total Annotations	@Override	@SuppressWarnings	...
org.alfresco	-	-	-	...
org.alfresco.config	5	4	1	...
org.alfresco.encoding	6	6	-	...
org.alfresco.error	1	1	-	...
org.alfresco.hibernate	1	-	1	...
org.alfresco.ibatis	6	4	2	...
org.alfresco.util	25	20	5	...
org.alfresco.util.resource	2	1	1	...
org.alfresco.util.exec	7	7	-	...
...	...	...	...	...

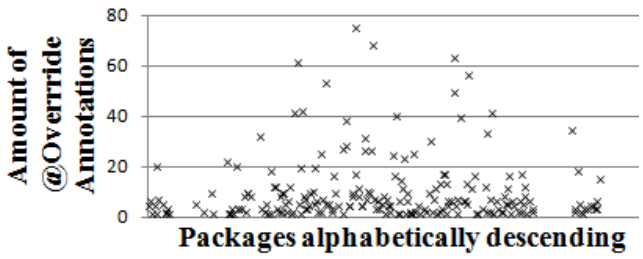
The distribution of all annotations over the entire packages in Figure A.17 shows that no concrete annotation distribution pattern seems to exist. Likewise, we cannot see a pattern for the @Override annotation alone (Figure A.18) or the @SuppressWarnings annotation (Figure A.19). However, we see a pattern in the distribution of the project specific @Auditable and @NotAuditable annotations (Figure A.20). When we consider the alphabetical order of the packages on the x-axis it seems that the distribution is depending on an alphabetically similar name. Hence, we divided the graphic in three regions: A, B and C. A shows clearly the highest density of annotations, followed by C. The points marked in B seem to be outliners. Form this graphic we conclude now that the distribution of the packages has somehow to correspond to the packages name. In the following, we explain the most likely reason for that.

Figure A.17. Distribution of all Annotations



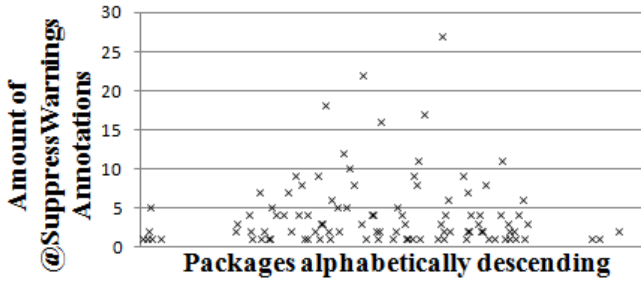
Distribution of the sum of all different annotations in alphabetically ascending packages. No concrete pattern is recognizable.

Figure A.18. Distribution of Override Annotations



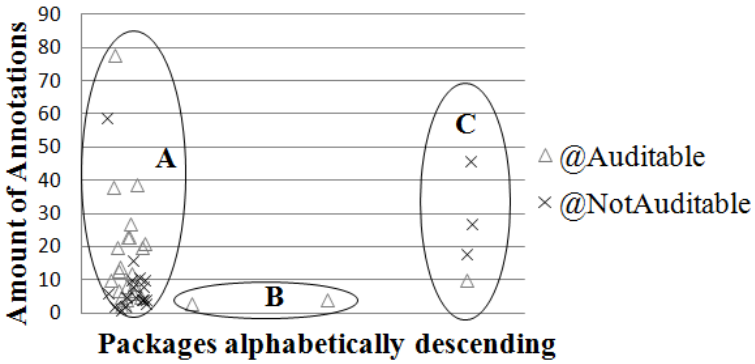
Distribution of the @Override annotation in alphabetically ascending packages names. No concrete pattern is recognizable.

Figure A.19. Distribution of SuppressWarnings Annotations



Distribution of the @SuppressWarnings annotation in alphabetically ascending packages names. No concrete pattern is recognizable.

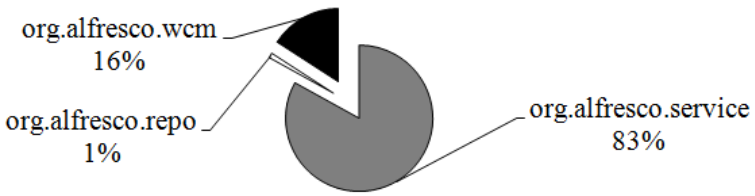
Figure A.20. Distribution of Audit Annotations



Distribution of project specific annotations with a semantic meaning in alphabetically ascending packages names. The annotations seem to appear in clusters. Most annotations occur in A, followed by C and some outliners can be seen in B.

Figure A.20 indicates that the distribution of @Audit annotations depends on the package name order. Therefore, we look at the design of the alphabetical order of packages. We see: Naturally, packages (e.g. org.b and org.c, org.d) that are children of the same parent package (org) follow alphabetically each other. Thus, packages that are located nearby are probably sub packages of the same parent package. Hence, our assumption is that the cluster A and C are probably children packages of two different parent package hierarchies. B is considered as outliners. In order to verify this "parent-package theory", we present the distribution of the Audit annotations and parent packages in Figure A.21 and Table A.8. The diagram and the table reveal that the annotations are mainly occurring in the *.service and the *.wcm package. Furthermore, the number of total methods indicates an independence of the amount of annotations from the amount of methods in the package. Therefore, we verify that the package hierarchy has likely an impact for annotation distribution.

Figure A.21. Distribution of the Audit annotations



The audit annotations occur only in three different package hierarchies. These three package hierarchies correspond to the three marked areas of A, B and C in Figure A.20.

Table A.8. Distribution of Audit Annotations

Package	Audit Annotations	Total Methods
org.alfresco.service	528	2149
org.alfresco.repo	7	21555
org.alfresco.wcm	101	701

Impact on MA1

The investigation of method annotations show a likely random distribution of code structural annotations (@Override, @SuppressWarnings). In contrast to this, annotations with a clear semantic meaning (@Auditable, @NotAuditable) show a cluster appearance. The clusters appearance occur in a relation to the different package hierarchies within code.

All together, we see support for our first assumption MA1 that annotations are not distributed randomly over packages. We see first indications that annotations appear in packages based on their semantic meaning. However, our investigation showed only a few annotations and we cannot conclude to much. Further research needs to find more detailed facts about annotation distribution within other projects. However, the data we presented right here, argues for our main assumption MA1.

A.5.2. Feature Distribution Computations

Out of the prior indications about annotations, we see the need to investigate similar circumstances that occur more regularly in source code. Since fields and method parameters occur more often than annotations, we need additional computations to compare their occurrence in and outside packages.

Therefore, we extend the prior done investigation about packages as categories and lean our computations towards a study that investigates the features distribution of category elements [221]. Simplified, this study is supporting the prototype theory, wherein a category is created by a set of elements that have more likely features of other category elements than elements outside the category. Thus, a category is a expressed as a likelihood wherein the occurrence of different category typical features follows a higher occurence frequency than outside the category.

Here, we define a package as category and the appearing types, like classes and interfaces, as elements of the category. Such type elements have features. We define this features to be the prior annotations, fields method parameters and so on. In the following, we present how we compute indicators for fields, what can be transferred to other features of a type.

We present excerpts of exemplary source code of the prior mentioned the petclinic application in the following source code in Example A.1. This source code is mapped to Table A.9, where be present different computed indicators. We discuss the source code and the computation the following.

Here, we refer often to features, such as fields, that are shared between types. In special, we refer to fields defined to be of certain types (class, interface, enum), which are defined in a type. Applied, we refer to private variables that are defined in a class and credit their type definition. Once a private variable is defined to be of the same type, like a private variable in another type, we speak of a shared feature. In fact, it is even wrong to speak of shared fields, because each field is a different variable, but it gets quite chaotic to read, when we do not apply this simplification. Therefore, a package contains types, those types contain features, like fields, that are defined to be of another type. If the field type definitions equal, we speak of a shared feature. For the sake of better readability, we use often field feature, field or just shared fields to refer to the field type definition in the following.

We see the indicator of the total amount of types. It describes which features occur at which type. Like in the source code, BaseEntity has a field with the type Integer, NamedEntity with a String

and so on and so forth. We see in column C and D that the features, with a generic type, can be computed by distinct and total features within a project.

The number of equal field appearances in types of a package is shown in column E. There, we compute the total occurrence numbers of fields in the whole package elements. We see that in the petclinic package only one String and Integer occur in multiple types. We sum those occurrences up as aggregate (e.g. E6). Therefore, as more features occur in different types, the total number of occurrence in elements grows. Lesser the number, lesser shared features. If multiple features are shared types occur multiple times and all their occurrences are summed up. This way, the total sum is often higher as the number of types in a package, because features are shared.

Furthermore, we restrain the computation of field occurrences by the criteria that a field has to occur in at least two types of the package (F). We show this possibility, because this kind of limitation enables to possibility to focus on fields that are shared more often between types. This limits the effect on just once appearing fields. In addition to the occurrence computation in the package, we show the same computations for all packages (G/H) .

We compute the ratio of field occurrences to the amount of types in and outside a package as indicators in E7/G7. Again, we show the possibility to compute the same indicator for fields that occur in at least two different types inside a package in F7/H7. We call this ratio indicator the "Category element feature distribution ratio" (CFDR) to refer to the rate within a package and "Global element feature distribution ratio" (GFDR) to refer to the share rate outside the package.

CFDR and GFDR can be interpreted as follows. If this indicator is above 1, we have more shared fields between types than types in the investigated set exist. Hence, features appear in multiple types. If it is beneath 1, it is likely that types have less features in common than different types exist. This means: The fields that occur in one type are not likely to occur in another type, too. However, in our investigation it is of interest if the CFDR is higher then the GFDR to determine if field features occur more likely in types of a package compared to their occurrence outside a package. For instance, we see that the CFDR for the org.springframework.petclinic package for all field features (E7) is higher than the GFDR (G7), what means that the shared features within the package are distributed on less different types then on the outside. This means that the likelihood that types share features is higher inside a package is higher then the sharing with types that are not in the package.

Example A.1. Field Features in petclinic

```
package org.springframework.petclinic;
public class BaseEntity {
    private Integer id;

public class NamedEntity extends BaseEntity {
    private String name;
    -...}

public class Owner extends Person {
    private String address;
    private String city;
    private String telephone;
    private Set<Pet> pets;
    -...}

package org.springframework.samples.petclinic.aspects;
@Aspect
public class UsageLogAspect {
    private List<String> namesRequested =
    new ArrayList<String>(this.historySize)-;
    -...}
```

Table A.9. Field Feature Indicator Example Computations

	Category - Package (A)	Element - Type (B)	Feature - Field (C)	Feature - occurrence (D)	Feature occurrence in at least one element of the category (E)	Feature occurrence in at least two elements of the category (F)	Global Feature occurrence in at least one element (G)	Global Feature occurrence in at least two elements (H)
1	org.springframework.petclinic	BaseEntity	Integer	1	1	-	1	1
2		NamedEntity	String	1	2 (B2, B4)	2 (B2, B4, B9)	3 (B2, B4, B9)	3 (B2, B4, B9)
3		Owner	Pet	1	1	-	1	-
4			String	1	- (counted in E2)	- (counted in F2)	- (counted in G2)	- (counted in H2)
5			Set	1	1	-	1	-
6		Local types: 3	Distinct Features: 4	Total Features: 5	Local feature occurrence: 5	Local feature occurrence: 2	Global feature occurrence: 6	Global feature occurrence: 5
7		-	-	-	Category element Feature Distribution Ratio (CFDR) = (E6/B6) = 5/3	Category element Feature Distribution Ratio (CFDR) = (F6/B6) = 2/3	Global element Feature Distribution Ratio (GFDR) = (G6) / (B12) = 6/4	Global element Feature Distribution Ratio (GFDR) = (H6)/ (B12) = 5/4
8	org.springframework. samples.petclinic.aspects	UsageLog Aspect	List	1	1	-	1	-
9			String	1	1	-	3 (B2, B4, B9)	-
10		Local types: 1	Distinct Features: 2	Total Features: 2	Local feature occurrence: 2	Local feature occurrence: 0	Global feature occurrence: 4	Global feature occurrence: 0
11		-	-	-	Category element Feature Distribution Ratio (CFDR) = (E10/B10) = 2/1	Category element Feature Distribution Ratio (CFDR) = (F10/B10) = 0 = 0/1	Global element Feature Distribution Ratio (GFDR) = (G10) / (B12) = 1= 4/4	Global element Feature Distribution Ratio (GFDR) = (H10)/ (B12) = 0 = 0/4
12		Global types: 4						

We provide a better overview of the prior described computed indicators of Table A.9 and aggregate them the package level in Table A.10. We see CFDR and GFDR as overall comparative indicators. In addition to those, we add their average for all packages. In our example, we see that the average distribution of CFDR is higher as the average GFDR. This means that the shared feature-type occurrence is higher inside packages then their occurrence outside packages, in average.

Table A.10. Indicator Computations for Package level Feature Occurrence

Category - Package (A)	**Elements in Category (B)**	**Feature Occurrence in at least X elements of the category (E/F)**	**Category element Feature Distribution Ratio (CFDR)**	**Global Feature occurrence in at least X elements (G/H)**	**Global element Feature Distribution Ratio (GFDR)**
		X = (one Element, E)	**E/B**	**X =(one Element, G)**	**G/Total Types**
org. springframework. petclinic	3	5	5/3	6	3/2 = 6/4
org. springframework. samples. petclinic.aspects	1	2	2/1	4	1= 4/4
	Total types = 4	-	Average = 11/6 = ((5/3) +(6/3)) /2	-	Average = 5/2 =((6/4) + (4/4)) /2

A.5.3. Field Feature Distributions

We apply the in the prior section described computations to alfresco project. Therefore, the results from which we discuss visualizations in the following, are computed like Table A.10.[2] In general, we correspond to MA2 and determine if we can find similar structures for types and their field features as for categories.

In order to find supporting evidence for MA2, we respect that categories can be defined by a set of shared features among category elements. Right here, we compare the CFDR and GFDR values inside and outside a package of the alfresco project. If the CFDR is higher within a package, a package has similarities to a category, because more features are shared with other category elements then with elements on the outside.

First, we show how all field type features are distributed without a restriction. Then, we use the restriction of project only feature types. Lastly, we relate our investigations to the main assumption MA2.

Field Feature Distributions without Restriction

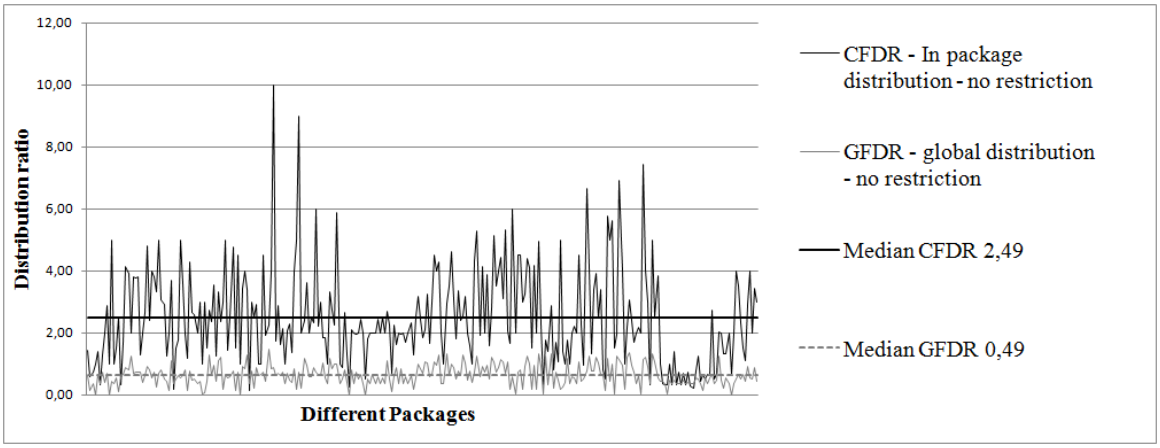
By computing the indicators, we filter the result to packages, containing at least one type with a field occurring in the package. Likewise, we do the same for fields that are defined to be of a primitive kind (int, boolean etc.). This means that packages that contain only interfaces or only primitives are likely to be filtered, because those do often not define any fields. Nevertheless, in case there are interfaces in a package that have no fields, they are still increase the total amount of types for those. This way, we do not filter interfaces completely out of our investigation, and only filter cases with packages that are irrelevant for the current investigation, since a zero occurrence of features would not contribute anything to the results.

Anyway, as data of the result set with the indicators, we have the total amount 3191 types, occurring in 283 different packages. We note that we do not respect the hierarchy structure of packages and treat every package on its own, like an independent unit. We do this, because in Java the package has no influence to the accessibility or the scope of variables. Hence, we see all packages as orthogonal to each other.

[2]We only show parts of the result data, because the dataset with the indicators that we visualize is very large.

Figure A.22 shows the distribution of features in the different packages. Thereby, we compute the CFDR. Every field feature is respected, even the ones that occur one time within a package. We connect the results for the packages by lines to have an easy to see overview, if the GFDR value hits the CFDR value. What we that the CFDR is in nearly all cases higher than the GDFR. This results in a median of CFDR above the GFDR.

Figure A.22. CFDR and GFDR of Field Type Features



We inspect Pearson correlations and significance tests (sig., two tailed, p value) for the values of the packages. We compute those for the amount of types in a package and CFDR and GFDR and show the results in Table A.11. The high p value and the low correlation for the amount of types in a package to the CFDR let it seem unlikely that a Pearson correlation exists here. The correlation to the distinct features indicates a little dependency. However, when we compare this to the correlations to the global shared features, we have a much higher likelihood there. Additionally, there seems to be a correlation of the GFDR to the CFDR value.

We interpret this as follows. The package size (amount of types in a package) seems to have an impact on the shared features with the outside world. When the package is larger it gets more likely that features are shared with other packages. On contrast to this, larger packages seem not to have an effect on the internal share rate. We assume this has something to with unsharp boundaries of categories and that the amount of types in a package is to low to reveal a Pearson correlation. We just can say that we cannot find a clear dependence of package size to kind of fields therein. Since the CFDR is higher then the GFDR and the GFDR has a correlation to the package size, we can say that for larger packages is seems likely to share features with other packages and overall the in-package shared features seem higher.

Table A.11. Correlations of CFDR and GFDR for Fields

Correlation from - To		Elements in Package (B) (Package size)	Total local distinct Features (C)	CFDR (E)
CFDR (F)	Pearson Correlation	-,020	0,394	-
	Sig. (2-tailed)	0,741	0,000	-
GFDR (H)	Pearson Correlation	0,604	0,856	0,514
	Sig. (2-tailed)	0,000	0,000	0,000

The non existing person correlation of the package size makes us wonder, if we get different results if we limit the types of fields to fields that occur at least twice within a package. In short, we use column F and H for the computation (Table A.9). Out of this restriction we recompute the indicators and clean the result to packages that contain at least a field type that occurs in at least two types. The result are 216 packages, left for our investigation. We stick to the 3191 types for the GFDR, because types the packages still can contain the same field types. Figure A.23 shows the distribution.

We see a similar picture to the prior one, where the CFDR is mostly above the GFDR. The main difference is the dropped median rate. Again, we compute statistic variables and present those in Table A.12. The statistics result in similar values to the prior ones. Still, CFDR seems to be independent from the amount of types and shares a correlation to GDFR and occurring features. We recognize that the p-value for CFDR to the package size has a dramatic decrease from 0,741 down to 0,161, but the correlation is still small. Thus, similar conclusions like in the prior case can be done. Therefore, we investigate if restraining the investigation to project only types retrieves other results, in the following.

Figure A.23. CFDR and GFDR of Field Type Features, Occurring twice

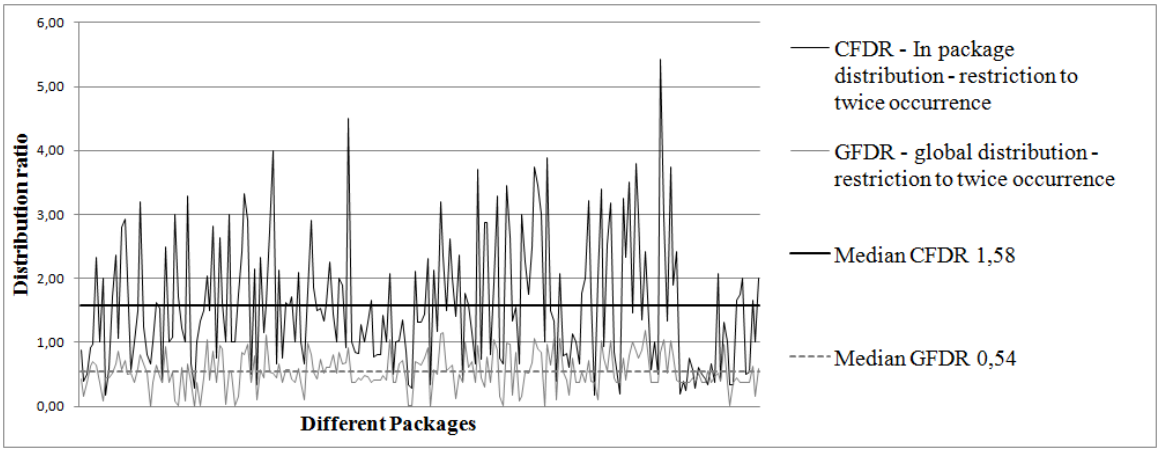


Table A.12. Correlations of CFDR and GFDR for Fields types, Occurring twice

		Elements in Package (B) (Package size)	Total local distinct Features (C)	CFDR (E)
CFDR (E)	Pearson Correlation	0,096	0,472	-
	Sig. (2-tailed)	0,161	<0,001	-
GFDR (E)	Pearson Correlation	0,599	0,787	0,596
	Sig. (2-tailed)	<0,001	<0,001	<0,001

Field Feature Distribution for Project Types

In order to get more detailed insights about package boundaries, we use semantic boundaries of field feature types. We do the same investigation like in the section before, but limit the field feature types to ones of the project. We do this, since we hope that the features that are defined within the analyzed project itself are "more semantic" than the used ones out of libraries and Java. The main reason, why we do so is because when we investigated annotations, the annotations that were part of the project (e.g. @Audit) showed only up in a few packages.

Out of the restriction, we end up in 1458 different types in 230 packages that have shared field types. We use this amount of 1458 types with features as reference value of global total types. Like before, we compute the different comparative indicators.

We show the GFDR and CFDR in Figure A.24. Thus, this figure is the comparative figure to Figure A.22. Similarly to its origin, the current figure shows that the CFDR value lies all the time clear ahead the GFDR. The shared field distribution is more done within a package than outside. This results in an even larger distance of the averages. Right now, the median is nearly three (2,92), where it has been 2,42 before. Outside it dropped to 0,42 where it was 0,49 before.

Again, we compute statistic correlations. Those are shown in Table A.13. In general, the dependency correlation is similar to Figure A.22. Hence, the average computations seem to support the

sharpening of results, but the statistic values give no support. Anyway, what we can conclude is that the restriction to project only features still brings up similar results then the prior ones without a restriction.

Figure A.24. CFDR and GFDR of project Restricted Field Features

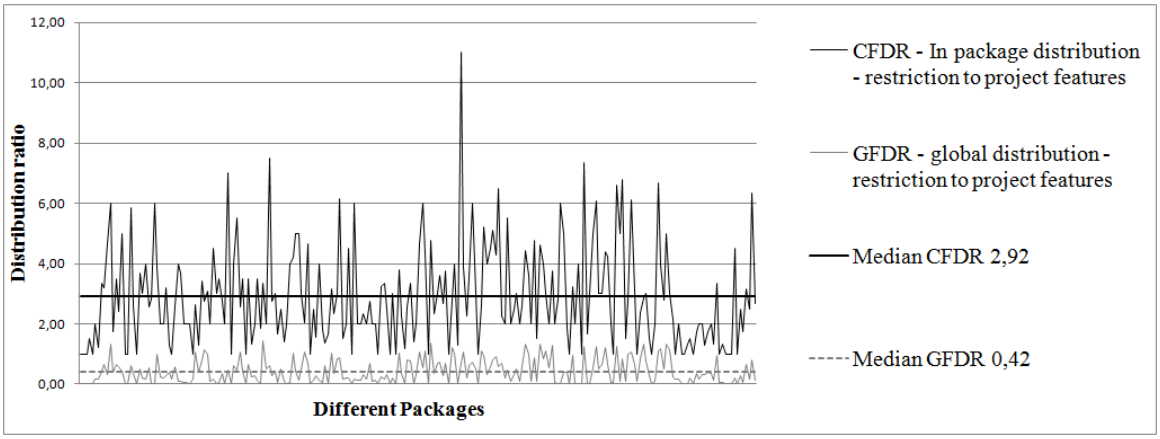


Table A.13. Correlations of CFDR and GFDR of project Restricted Field Features

		Elements in Package (B) (Package size)	Total local distinct Features (C)	CFDR (E)
CFDR (F)	Pearson Correlation	0,075	0,436	-
	Sig. (2-tailed)	0,258	<0,001	-
GFDR (H)	Pearson Correlation:	0,670	0,840	0,552
	Sig. (2-tailed)	<0,001	<0,001	<0,001

In order to get further insights and to have another comparison, we restrain the investigation to features that occur at least in two types within a package. This restriction results in 151 packages with computed indicators for the investigation. We show the internal feature distribution with CFDR and the external with GFDR in Figure A.25. The average distribution is at a higher level, but the distance to the outside distribution is, with around 1, similar to the value without a restriction in Figure A.23.

The correlations in Table A.14 are similar to the ones in Table A.12). However, the p-value decreased for the package size to the CFDR (0,049) and a low correlation seems to exist (0,161).

Figure A.25. CFDR and GFDR of project Restricted Field Features, Occurring twice

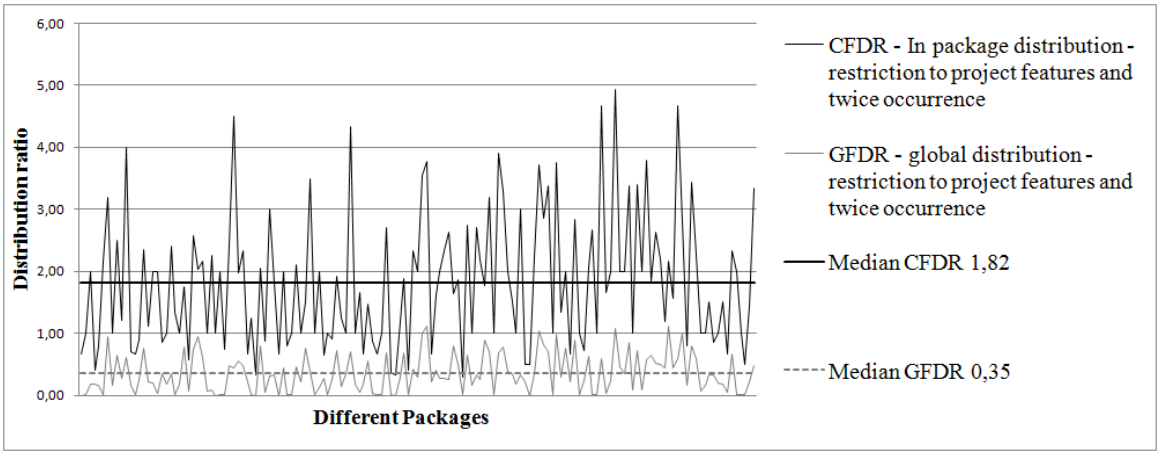


Table A.14. Correlations of CFDR and GFDR of project Restricted Field Features, Occurring twice

		Elements in Package (B) (Package size)	Total local distinct Features (C)	CFDR (E)
CFDR (F)	Pearson Correlation:	0,161	0,460	-
	Sig. (2-tailed)	0,049	<0,001	-
GFDR (H)	Pearson Correlation:	0,617	0,767	0,706
	Sig. (2-tailed)	<0,001	<0,001	<0,001

Impact on MA2

We found average share rates that seem likely higher in packages compared to the whole project. We repeated similar findings for field type definitions that occur at least in two different types. Likewise this was similar for restrictions to project only features. We found correlations for the share rate with all types in a project and the amount of types in a package. We did not find indications that the share rate in a package is depending on the package size.

Out of this, we conclude that we have support for MA2. Fields type definitions are somehow responsible for grouping types into packages, like category features are used to group elements into categories. Thus, source code packages share a resemblance to categories that they have a higher occurrence of the same features within. Fields type definitions are likely to be one criterion to be used for grouping classes into packages.

A.5.4. Boundaries of Packages (MA3)

We verify MA3, by showing that the boundaries of a package are likely to differ, once the criterion to measure the resemblance between the types in the package is altered. In special, we present how and if the feature distribution differs when we limit the investigation to an increasing demand the number of shared fields. We do this limitation, by increasing the number of shared features (Table A.9 E, F and G,H) to more then two. For instance, if we rise the amount of minimum shared features to six or more, we can assume that the corresponding features represent a likely more sharp criteria for a package than all features. We assume that an increasing restriction will have an impact to the sharpness of the package-category boundaries. In special we expect the following result:

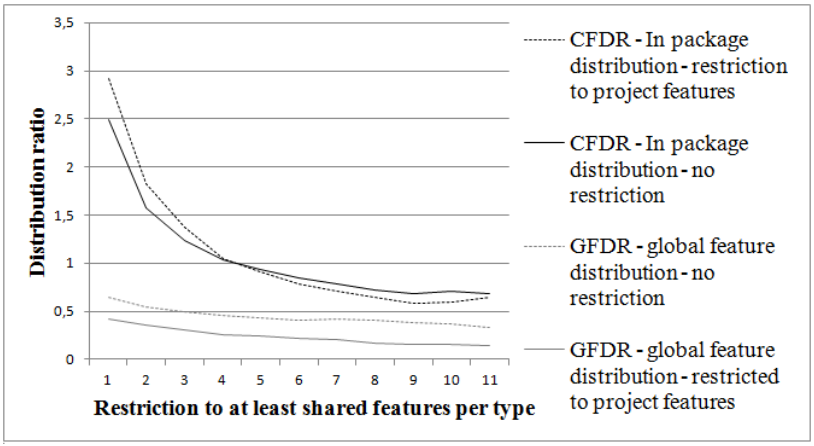
An increasing limitation to more shared fields will still result in higher appearance of the same features in a package compared to the whole project. This means that the CFDR averages will be higher then the GFDR averages.

We compute the different indicators and apply the restrictions. We show the result in Table A.15. Additionally, we compute the same values for "project only" fields. We visualize the results in Figure A.26. We verify the assumption that shared fields occur more likely within a package then on the outside. This is the case for all fields and for fields just of the project. We see that the CFDR rate drops down with the restriction to project only features as well as in the case of no restriction. In general, we cannot see a huge difference between using a restriction and using none. However, the main difference is that at the end (cases 4 shared features and above) the project only fields have a lower distance to the GFDR. Therefore it seems that the distribution for higher amounts of shared features is sharper for all types, then with a restriction to project features. However, in both cases, even with the restriction for at least 11 shared features, it seems more likely to find elements within a package that share the features. We compute correlations, in the following.

Table A.15. CFDR and GFDR Average, Feature Limitation, Project and Global

Feature occurrence in at least X Elements within the package	Project specific features (e.g. fields)		All features (e.g. all fields)	
X=	Category element feature distribution ratio (CFDR) average	Global element feature distribution ratio (GFDR) average	Category element feature distribution ratio (CFDR) average	Global element feature distribution ratio (GFDR) average
1	2,92	0,42	2,49	0,64
2	1,82	0,35	1,58	0,54
3	1,37	0,3	1,24	0,5
4	1,05	0,25	1,03	0,45
5	0,91	0,24	0,93	0,43
6	0,78	0,22	0,84	0,4
7	0,71	0,21	0,78	0,42
8	0,64	0,17	0,72	0,4
9	0,58	0,15	0,68	0,38
10	0,59	0,16	0,71	0,37
11	0,64	0,14	0,68	0,33

Figure A.26. Averages of Field Feature Distributions and Occurrence Restrictions



We compute correlations and present them in Table A.16. We see the correlations and significance test results. We recognize that the number of investigated packages for correlations decreases faster for the restricted investigation to only project specific types. The CFDR correlates to the package size in low levels with a high p-value, for nearly all cases. Therefore, we assume there is no trivial correlation or no correlation at all, between field types and the package size. Furthermore, we see the high p-value combined with the low correlation in most cases as first indicator that the CFDR maybe independent for the package size. The GFDR on the contrary has a low p-value and a higher correlation, what indicates a relation between the package size and the appearance of shared features in most cases.

All over, we get findings for a higher amount of shared features. Shared features seem more likely to occur in a package than on the outside. The same findings can also be repeated with a restriction to project only types. Packages can be defined more precisely, as we did with the restriction to share more features. In cases with a high restriction, the CFDR is still above the GFDR. However, once the packages shared features get defined more precisely, the difference to the outside shared features seems to narrow down. Therefore, we see an impact on the sharpness of categories by applying limitations. Therefore, we see strong indications for MA3.

Table A.16. CFDR and GFDR Correlations, growing shared Features Limitation

Feature occurrence in at least X Elements within the package	No restriction - correlation to amount of types in a package					Restriction to project types - correlation to amount of types in a package				
	Remaining packages (cases)	CFDR Correlation		GFDR Correlation		Remaining packages (cases)	CFDR Correlation		GFDR Correlation	
		Pearson	Sig. (2 tailed)	Pearson	Sig. (2 tailed)		Pearson	Sig.(2 tailed)	Pearson	Sig. (2 tailed)
1	283	-0,02	0,741	0,604	<0,001	230	0,075	0,258	0,67	<0,001
2	216	0,096	0,161	0,599	<0,001	151	0,161	0,049	0,617	<0,001
3	157	0,064	0,427	0,552	<0,001	93	0,129	0,218	0,565	<0,001
4	126	0,06	0,507	0,483	<0,001	62	0,148	0,251	0,511	<0,001
5	97	0,048	0,639	0,461	<0,001	47	0,132	0,375	0,519	<0,001
6	80	0,01	0,93	0,429	<0,001	35	0,016	0,929	0,342	0,045
7	64	-0,072	0,569	0,349	0,005	22	-0,21	0,348	0,133	0,555
8	55	-0,105	0,444	0,239	0,08	16	-0,354	0,179	0,136	0,615
9	46	-0,097	0,522	0,102	0,502	13	-0,25	0,411	0,056	0,857
10	36	-0,14	0,414	0,116	0,499	11	-0,384	0,243	0,005	0,989
11	31	-0,022	0,908	0,187	0,314	8	-0,703	0,052	0,175	0,678

A.5.5. Package Taxonomy (MA4)

Right before, we investigated the field feature distribution in packages. Thereby, we ignored that packages are commonly ordered in a taxonomic way. We ignored this, because the taxonomic structure of packages has no influence to the scope of programming structures, like type-field definitions. In case of annotations, we already saw that the taxonomic structure of packages has an impact, regardless of its non-syntactic meaning. Therefore, we do the an investigation for fields. We stick to the same investigation, like in the section before and just focus on package hierarchies. Our goal is to reveal if the taxonomic structure brings up similar results for package hierarchies as for 'lonely' packages. We do this to credit MA4, because research about categorization credits hierarchies of categories.

In order to dig into this assumption, we compute indicators for different nodes in the package taxonomy. This means, if there is an org.alfresco.util package, we compute the indicators for org, for org.alfresco and org.alfrescoutil and include every time the descendant packages that are lower in the hierarchy. We call every identifier like org. org.alfresco package node.

In order to focus purely on the hierarchies, we limit the amount of types in a package hierarchy to at least a certain number. We measured the average amount of types within a package to be 12. Therefore, we only query for nodes in the package hierarchy that contain at least 25 types. We do so, because we fear that the non hierarchic results can influence the investigation if hierarchies matter in the feature distribution. With 25 we are confident that we get mostly hierarchies and not large packages. When the average occurrence rate of types is 12 in packages, the number of 25 contains in average multiple two packages and we respect assure that our measurements target hierarchic relations.

All in all, we end up in the amount of 82 different package nodes in the package hierarchy. We compare this to the total of nodes of 378 in the package hierarchy when we do not apply the limitation to 25. Thereby, the 378 nodes consist of 283 that contain at least one field. Hence, the shrinking through the restriction assures that the projection to less package nodes contains aggregations of sub packages. This assures the selected 83 nodes as valid for our investigation. We note that the total amount of types in packages is growing to 3463 from 3191 before. The reason for this is that some packages with completely no features were excluded in the prior investigation. Right here, they are part of the investigation, since they appear in hierarchies and influence the results.

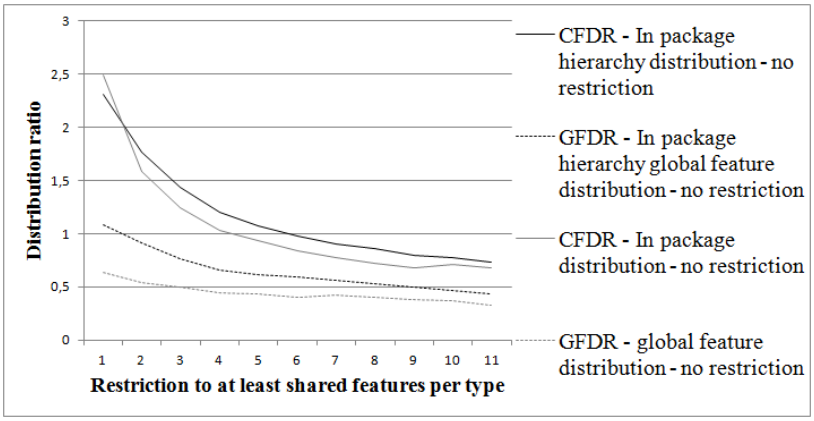
We compute the indicators and present them in Table A.17. In contrast to the prior investigation in Section A.5.4, we see that the amount of cases (Amount of remaining package nodes) stays nearly constant. The reason for this is that it is likely higher that a certain amount of features is shared within a package hierarchy as in a lonely package. However, we visualize the average CDFR and GFDR in Figure A.27. Additionally, we present two graphs of Section A.5.4 as comparative indicators and see that the values are similar to the ones before. The package hierarchy values are just at a higher level. The correlations and significance tests also end up similar to the pure package that was discussed in the chapters before. The CFDR seems to have no Pearson correlation to the package hierarchy size and the GFDR has a significant correlation to the package size. Therefore, we have the similar findings for package hierarchies as for packages.

All this supports our main assumption, MA4: Hierarchies of packages matter in field feature share rates. Category hierarchies matter in case of categorization. Package Hierarchies have no scope meaning in programming, but still, we find clear indications that hierarchic categorization is used as well in programming.

Table A.17. Package Hierarchy shared Feature occurrence

Feature occurrence in at least X Elements within the package	Remaining packages (cases)	Average/Mean		Correlation of CFDR to amount of package types		Correlation of GFDR to amount of package types	
		CFDR	GFDR	Pearson	Sig. (2 tailed)	Pearson	Sig. (2 tailed)
1	82	2,31	1,08	-0,019	0,865	0,615	<0,001
2	82	1,77	0,91	0,06	0,595	0,648	<0,001
3	82	1,43	0,76	0,126	0,258	0,647	<0,001
4	82	1,2	0,66	0,189	0,089	0,675	<0,001
5	79	1,07	0,62	0,236	0,037	0,682	<0,001
6	76	0,98	0,59	0,284	0,013	0,717	<0,001
7	76	0,9	0,56	0,334	0,003	0,734	<0,001
8	74	0,86	0,53	0,358	0,002	0,736	<0,001
9	72	0,8	0,5	0,402	<0,001	0,752	<0,001
10	70	0,77	0,47	0,422	<0,001	0,69	<0,001
11	69	0,73	0,43	0,45	<0,001	0,762	<0,001

Figure A.27. Averages Field Distributions, Restrictions, Package Taxonomies



A.5.6. Method Parameters Features

Before, we investigated field feature distributions in packages. Here, we do a similar investigation for method parameters. Research about categorization argues for different kind of features of

category elements that are used within different contexts to categorize those. Methods are used differently to fields within a program.

In general we assume that method parameters are also features of elements within a "package as category". Since fields are mostly defined to be private variables, methods are often public and occur regularly in interfaces. Therefore, we are careful with our assumptions, since method parameters are something that is used totally different in comparison to fields. However, to verify MA 5, we assume that we get similar findings then before.

Hence, we compute the shared feature indicators for method parameters in the same way like we did it for fields. We compute if a type of a method parameter is occurring at method parameters of other types. Finally, we result in the complete same indicators as for fields, just with the difference that the investigated feature are now method parameter types and not field types and more. Thus, when we speak of features in this section, we refer, when not said explicitly, to the features of method parameters.

In order to get similar findings, we assume that method parameters are more likely to occur in multiple types inside a package then on the outside. This means that we will find a higher CFDR then a GFDR. We assume this for the averages as well as in the cases without a restriction and a restriction of at least one shared parameter. We assume the package hierarchies have an influence to the distribution as well.

In addition to the prior assumptions, we also regard if we can apply abstractions, to the shared feature, in the case for method parameters. We do so, because in a program much more method parameters appear compared to fields. Our goal is to determine if the packages wherein the parameter types are defined, has an influence to the results. In special, we assume that it will make the results fuzzier.

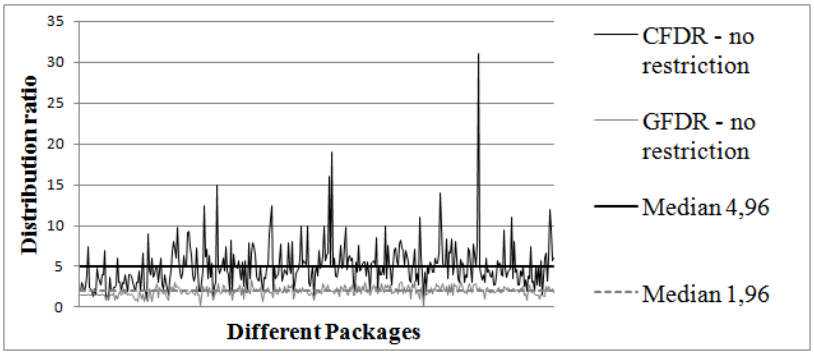
In the following, we repeat similar investigations like for fields for method parameters and refer to the corresponding chapters.

Method Parameter Package Distribution

First we repeat the investigation (Section A.5.3) to treat all packages as orthogonal. We compute the feature occurrence for method parameter type features and end up in 358 different packages with the total amount of 3453 types in the packages.

We present the feature distribution, without a restriction to the at least shared types as CFDR and GFDR in Figure A.28. Similar to the case of the fields (see Figure A.22), we see that the CFDR value is nearly all times ahead of the GFDR value and likewise is its average. The main difference to the type features is that the averages appear at a higher level.

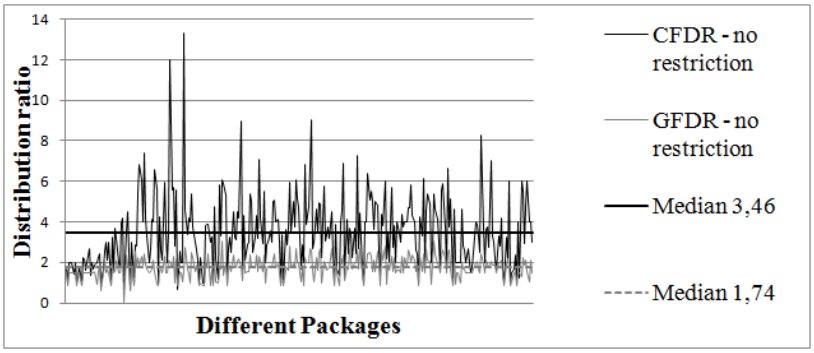
Figure A.28. CFDR and GFDR of Method Parameter Feature



We restrict our query to reveal only method parameters types that are at least occurring in two types within a package and get the result of 305 remaining packages. We show the visualization of the result set in Figure A.29. There, we see again a similar chart to the prior one, but with a certain decrease of the averages. However, the values are still above the field case in Figure A.25.

Therefore, it seems that the feature occurrence for method parameter types is more likely to appear in the same package then in all method parameters of the project.

Figure A.29. CFDR and GFDR of Method Parameter Feature, Occurring twice



Growing Restrictions

Like in prior sections, we present growing restrictions of the at least required occurrence of a feature in types. Table A.18 shows the average values and correlations. We visualize the CFDR and GFDR in Figure A.30. The comparative values for fields can be found in Table A.15. Additionally, we populated the data of Figure A.26 and Table A.18 in the visualization in Figure A.30. We show and discuss the data in the following.

Table A.18. Method Parameter Indicators, orthogonal Packages

Feature occurrence in at least X Elements within the package	Remaining packages (cases)	Average/Mean		Correlation of CFDR to amount of package types		Correlation of GFDR to amount of package types	
		CFDR	GFDR	Pearson	Sig. (2 tailed)	Pearson	Sig. (2 tailed)
1	358	4,96	1,96	0,026	0,617	0,591	<0,001
2	305	3,46	1,74	0,283	<0,001	0,632	<0,001
3	253	2,85	1,65	0,434	<0,001	0,661	<0,001
4	207	2,63	1,58	0,444	<0,001	0,725	<0,001
5	175	2,45	1,53	0,455	<0,001	0,639	<0,001
6	159	2,28	1,5	0,465	<0,001	0,681	<0,001
7	144	2,12	1,42	0,475	<0,001	0,661	<0,001
8	120	2,07	1,41	0,45	<0,001	0,644	<0,001
9	109	1,95	1,4	0,496	<0,001	0,621	<0,001
10	98	1,85	1,36	0,504	<0,001	0,62	<0,001
11	91	1,8	1,34	0,624	<0,001	0,624	<0,001

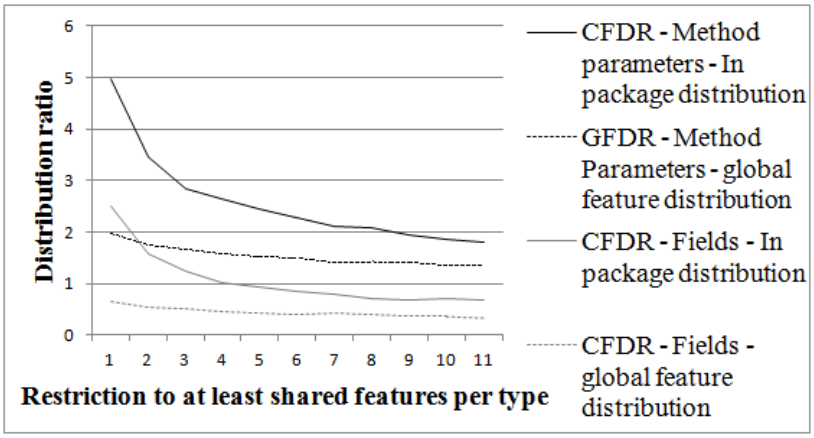
In Figure A.30, we see that the data for method parameters gets arranged at a higher level then for fields. We also recognize that the field feature distribution line is decreasing with a lower gradient then the one for method parameter features. We assume that the higher values of CFDR and GFDR for method parameters result of the reason that method parameters exceed field types in numbers. In general, we account the visualization as similar to the case of fields. However, the main difference to the field features correlations is that the shared features CFDR rate seems now to have a correlation on the package size, what seemed not to be the case for field features. We assume the reason for this is the higher amount of method parameters compared to fields.

We also recognize that the correlations between CFDR, GFDR and package size seems to get more similar with growing restrictions. We interpret this as indicator that the likelihood to find types

that share all features in a package gets similar to find a type that shares all features at the outside. Therefore, growing restrictions seem here not to sharpen the boundaries of a package, when we credit method parameters as criterion to be used of grouping types into packages. Anyway, this are just guesses and we see the necessity for more detailed investigations in the future.

We conclude that of a small set of restrictions and the higher CFDR than GFDR for such cases indicates that method parameters serve for such cases as a criterion that gets used to group types into packages together.

Figure A.30. Averages of Method Parameter Distributions, growing Restrictions



Package Hierarchies

We do the same investigation like before, but use the package hierarchy, like we did it for field features in Table A.17. Like there, we restrict the query results to at least 25 appearing types within a hierarchy to be certain that our analysis does not credit lonely packages and focuses directly on hierarchies. We show our findings in Table A.19.

We visualize the difference to the prior findings how the average CFDR and CFDR develops in Figure A.31. Additionally, we populate additionally prior averages without hierarchies from Table A.18. We see that the GFDR and CFDR value are much higher when the hierarchy is included then without the package hierarchy. Furthermore, the decrease of the CFDR with hierarchies has a reduced gradient in contrast to the CFDR. We consider the origin of this to be a containment of more types that share features within the same hierarchy, what results in a slower decrease.

However, much more interesting is if the correlations change in comparison to the prior section without hierarchies. We see that the correlation that we revealed between the CFDR and package size before, now is less intense. This makes us wonder in the first place, because we did not find similar values to the case for pure packages without hierarchies. Therefore, it seems that method parameter types are one criterion for arranging packages in hierarchies. A logical explanation for this finding an often applied practice in programming be: Interfaces for classes are defined in other packages then their class implementation. Commonly, there is a package that defines the interfaces (e.g. org.services) and then there is an *.impl (e.g. org.services.impl) package arranged lower in the hierarchy, containing the implementation. However, the GFDR is like in all cases before correlating to the package size.

We tested this assumption and looked into the project and found for instance 16 packages that are just named *.impl wherein classes are defined that implement interfaces at a higher level. 16 is a high number if just 84 package hierarchy nodes are inspected. We see this ass support for our interpretation. However, more details investigations would be needed to be certain.

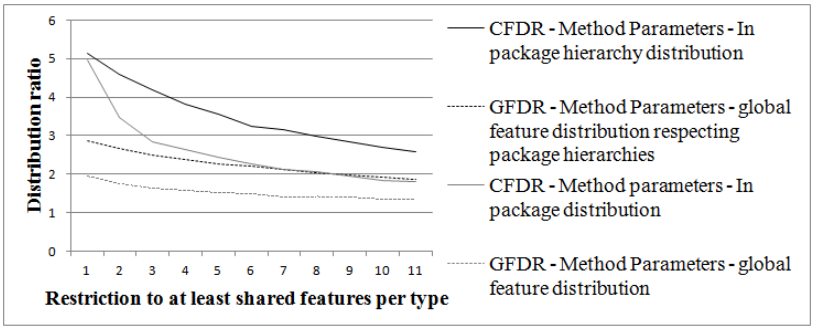
In any case, further research needs to dig into the specific details how the hierarchy exactly influences the independence and to which degrees interfaces are responsible. Right now, we have

strong indications that package hierarchies play a role for method parameter types. Even more, they seem to play a larger role for method parameter types, like for fields.

Table A.19. Method Parameter Indicators, Package Hierarchies

Feature occurrence in at least X Elements within the package	Remaining packages (cases)	Average/Mean		Correlation of CFDR to amount of package types		Correlation of GFDR to amount of package types	
		CFDR	GFDR	Pearson	Sig. (2 tailed)	Pearson	Sig. (2 tailed)
1	84	5,14	2,88	-0,017	0,876	0,681	<0,001
2	84	4,6	2,66	0,038	0,731	0,724	<0,001
3	84	4,18	2,5	0,082	0,458	0,704	<0,001
4	84	3,81	2,37	0,12	0,278	0,494	<0,001
5	84	3,55	2,27	0,148	0,178	0,68	<0,001
6	84	3,23	2,2	0,175	0,111	0,677	<0,001
7	84	3,15	2,12	0,205	0,061	0,682	<0,001
8	84	2,99	2,05	0,231	0,034	0,691	<0,001
9	84	2,83	1,99	0,247	0,024	0,699	<0,001
10	84	2,69	1,93	0,268	0,014	0,698	<0,001
11	84	2,58	1,87	0,291	0,007	0,71	<0,001

Figure A.31. Averages of Method Parameter Distributions, growing Restrictions, Package Hierarchies



Method Parameter Types Package Abstraction

However, a method parameters occur in higher numbers then fields. In detail, the investigated project contains 63772 method parameters and 13694 fields. Thus, we wonder if we still get similar results, when we abstract the method parameters type feature to their packages. Hence, we do the same investigation, like in the last section, with a restriction to 25 types, and compute the indicators based on the parameters type package. For instance, a method contains the parameter org.bookkeeping.Account and we abstract it to the package org.bookkepping and drop the specific type. Another example are java.util.Collection and java.util.ArrayList that both result in java.util. This abstraction to the package level sizes the different kinds of parameter fields types down to 139 different investigated packages. Therefore, we have now 139 different parameter type features left, compared to 63772 that were used in the prior sections.

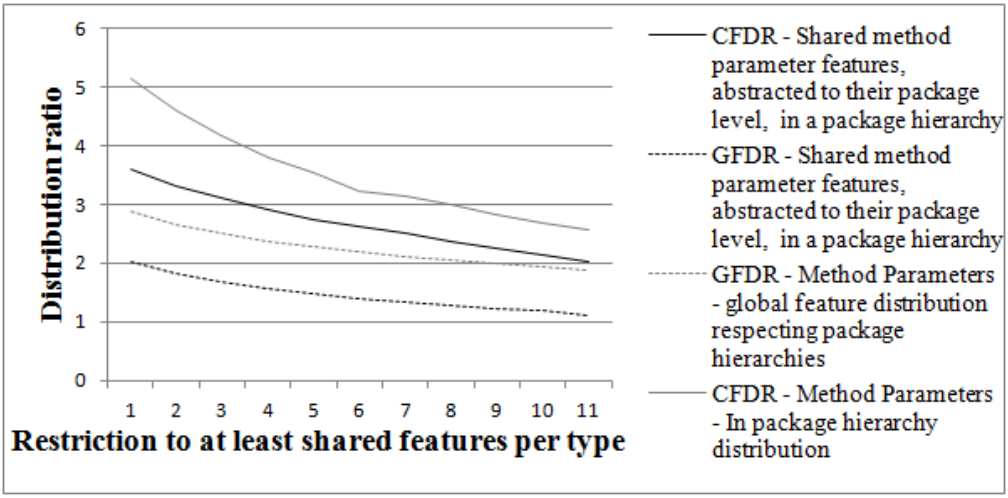
We report the result of our computations. We see in Table A.20 and Figure A.32 that the GFDR and CFDR averages are arranged at a lower level compared to the average that was computed for of method parameters without abstraction in the last section (see Table A.19, which GFDR and CFDR is also populated in Figure A.32). We consider this drop logical, because the amount of different types dropped also through the reduction to packages. However, we still have a measurable effect between the CFDR and GFDR average values. Therefore, we can tell that it seems more likely that

method parameters are defined to be out of the same package and occur more times in a package then outside. The correlations and significance tests are similar to before and indicate again that we cannot get an indication of a relation between the package hierarchy size and the CFDR. Again, the correlation between the package size and the GFDR seems to exist.

Table A.20. Method Parameters to Package Abstraction, Package Hierarchies

Feature occurrence in at least X Elements within the package	Remaining packages (cases)	Average/Mean		Correlation of CFDR to amount of package types		Correlation of GFDR to amount of package types	
		CFDR	GFDR	Pearson	Sig. (2 tailed)	Pearson	Sig. (2 tailed)
1	84	3,59	2,01	-0,022	0,843	0,57	0,002
2	84	3,32	1,82	0,027	0,808	0,634	<0,001
3	84	3,1	1,68	0,064	0,563	0,641	<0,001
4	84	2,92	1,57	0,093	0,403	0,642	<0,001
5	84	2,75	1,47	0,117	0,289	0,634	<0,001
6	84	2,63	1,4	0,134	0,223	0,631	<0,001
7	84	2,5	1,33	0,158	0,15	0,634	<0,001
8	84	2,36	1,27	0,187	0,088	0,645	<0,001
9	84	2,26	1,22	0,204	0,062	0,665	<0,001
10	84	2,14	1,18	0,223	0,043	0,683	<0,001
11	84	2,03	1,11	0,241	0,028	0,68	<0,001

Figure A.32. Averages of Method Parameter Features Packages Distributions, growing Restrictions, Package Hierarchies



Impact on MA5

All over, we conclude that we get findings for annotations, method parameters and fields to be used criteria to assemble types in packages and package hierarchies. This way, we verify M5. Furthermore, even the packages of method parameters seem to be a criterion. Additionally, the computed averages and statistics differed for the diverse feature kinds. Therefore, we assume that the different feature kinds seem to be used in different terms to structure a program into packages and their hierarchy.

A.6. Main Findings, Discussion and Hypermodelling

First, we get into the main findings of this chapter. Then, we discuss potential arguments against our investigation. Finally, we relate this chapter to the evaluation of Hypermodelling.

A.6.1. Main Findings

We provided a new theory that categorization plays a role in programming activities. In special, we illuminated that research about categorization follows similar structures like separation of concerns. We described the similarities of several mechanisms to separate concerns and categorization presented a complete program comprehension theory based on categorization. Different statistics showed first support for the theory that categories manifest in code structure. In detail, we found support for five different assumptions that arranging types into packages and their hierarchies is similar to categorizing elements into categories.

We are confident to say that annotations with a semantic meaning do not get distributed randomly within a project. Specifically, we have first support that they correlate to the semantic meaning of packages. Additionally, we presented first evidence that fields of classes are comparable to features of category elements. Certain field combinations occur more likely within a package than outside. We assume: The types of fields are one criterion to be used to group types together into packages. Similar results as for packages were revealed for package hierarchies. Package hierarchies have no scope meaning in programming, but we find indications that they mater for type arrangements therein. This is the case for method parameters as well as for fields. Hence, method parameters types and field types are more likely to occur in types of the same package or package hierarchy than in the whole project. In case of methods parameters, we were even capable to abstract the used types the used packages and still got significant results.

All over, different structures of a program and the distribution of elements therein shows indications that categorization is applied by developers and manifests in code structure. All the different perspectives indicate that categorization is used to structure different elements in the programming process. Therefore, we infer and predict that categorization is done for other elements in programming as well. As a matter of fact, our investigation represents a first evaluation, and we see the necessity for further and detailed evaluations how categorization is applied during programming. Our findings report the clear need for detailed investigations in the future.

A.6.2. Discussion

However, one may argue that the investigated amount of data is too small or that the statistic values of their interpretation are not valid enough to argument for our theory that categorization is an essential process in software development. He could go on and argue that intertwining of concerns does not mean that humans do not try to separate the concerns.

We argue against this point and mention that cross-categorization already provides examples where the context influences the chosen category. However, we do not argue that separation of concerns does not work for us humans. There can clearly be examples provided where a clear separation is excellent. Nevertheless, for many cases in programming it is difficult to come up with good mechanisms to realize the separation. In most cases, separation is not sharp. We think, it is natural for humans to try to separate concerns, but intertwining and the recognition of the right category of an intertwined object is a critical case that often appears. Hence, the more important question is: What are the right means to separate and to compose these modules? It is not important to discuss whether separation of concerns is natural or not; it is more important to see it from the point of comprehension and which means of separation should be used in which case.

He could come up with other examples where certain programming mechanisms are used differently from our comparison. We mention that our comparison shows well known cases and already similarities excelled. However, our comparison is a first observation. Therefore, such challenging cases may exist. Again, we see the necessity for more detailed discussions in a larger forum.

Another way to threaten our theory maybe to argue against the citied literature, because we use this literature to derive our comprehension model. For instance, one my refer to [218], which contains the origin of our conclusion that effective programmers have different investigation strategies compared to others. The described study in the original paper is not based on a large number of programmers. Therefore, the assumption that the investigation strategy can be associated with concerns is wrong.

We argue against this falsification, because of the fact that categories can change the behavior [44]. Since an investigation strategy is a kind of behavior, we argue for the accuracy of our conclusion, again.

Likewise, he could argue against our cited psychological studies. He could say that there is no complete model of the human mind available and there are only theories of how it works. This way, only empirically supported research can be considered as delivering indisputable facts. Therefore the different theories, exemplary, classic and prototype and also hybrid ones may be wrong. This could affect the comparison of these theories with the means of separating concerns. We argue that these theories are based on empiric research and parts of them have been verified. Our comparison was only on a very basic level and already showed similarities. Also, the comparison between the theories and means of separating concerns is only a small part of the model. Even if all of these theories are proved wrong, only a small part of our model would be corrupted. Additionally, our first findings in the evaluation reported similar results to a psychological study.

Furthermore, one may use the psychological findings and theories to challenge our model. He can mention that the creation of categories just out of the mere composition of different elements is not right. We respond and induce the fact that psychological research appears to go in the direction of cognitive concepts that are used draw inferences about the combination of categories [176]. We state that compositions and grouping of source code elements are a such concept. This means that categorization and grouping within source code is one instantiation of such a concept. This way, our model holds is valid for the specific case of composition.

Finally, one could address our first evaluation. It could be accounted as to trivial or having to less cases. However, we response that the current results indicate that it seems more likely to share features in a package then on the outside. For the future, we see the need to develop a detailed method and to do an advanced investigation. Therefore, we response that future research needs to get further into these issues.

A.6.3. Hypermodelling

This chapter shows that categorization is similar to separation of concerns. Thereby, we underlined the theoretic approach of a new theory by an intensive study of code. The whole study was done by computing the necessary indicators with Hypermodelling. In general, the possibilities of Hypermodelling made it possible to do this investigation. For instance, it was possible to compute the different indicators with and without limitations to have comparative values at hand. A manual code investigation would have been much more effort. Therefore, this chapter underlines, again, the analytic capabilities of Hypermodelling and its suitability for code analysis scenarios.

However, we conclude from the research about categorization that a programmer associates a source code fragment with different concerns and derives functionality and understanding from different features. Thereby, a programmer uses different perspectives, out of multi-category association and different category kinds, to interpret the meaning of a code fragment. The perspective in which the developer studies the code fragment changes, depending on his task. The resulting claim is that concern detection and analysis should be capable to support multiple perspectives and not be limited to one. Hypermodelling capabilities match these requirements.

Additionally, categories are not defined sharp. Every time, the perspective and categories change, varying elements are associated with them. Humans classify other elements to be part of the category once they alter their perspective. In order to support perspective changes, a technology is needed that can handle multiple perspectives at once. Hypermodelling is purely multi-dimensional, what enables support perspective changes. For instance, Hypermodelling can be used to compute category members actively. The whole investigation in this chapter was done with Hypermodelling. Hence, the same computations can be applied in a programming scenario to compute actively the category members. We imagine that the elements outside a package that share many features with the elements in a package can be computed with Hypermodelling. This helps to reveal similar elements, belonging logically together. Therefore, we are confident an active computing analytic technology, like Hypermodelling, will be needed in such scenarios.

A.7. Related Work

Our model is located in the field of program comprehension and describing cognitive aspects in software engineering [244, 200, 79]. In [50, 51] a model is presented, based on the idea of problem domain reconstruction via top-down hypotheses. Thereby, prior knowledge is used and the verification of hypotheses is done with beacons. Programmers use these beacons to understand the way of operation of a code fragment [208]. Our model differs from that approach, because it is founded on categories and concerns. We are convinced that beacons represent similar constructs to what we called field-, annotation- or method parameter features. This way, we connect those beacons to features, categories and concerns, what have not been done in prior work. Furthermore, our model also differs, because it considers the influence of the context in which source code is studied for comprehension.

In [160], the comprehension is divided into a base of knowledge, a mental model and an assimilation process. The knowledge base contains the experience of a programmer. The mental model links the representation in the mind and the implementation of the program. The assimilation of a program is reached by applying the knowledge base to the program. Again this model does not differ very much from our approach. For instance, basic categories are comparable with the experience of a developer. Again, the main difference is that neither separation of concerns nor research on categorization is mentioned. Hence, logically, no study similar to ours is provided.

Research about a concern scheme, COSMOS [245, 248, 247, 63], can be seen related, since the conceptual space is modeled. COSMOS proposes a scheme with various kinds of concerns and rules how they can be associated with each other. The goal was to develop a method to map these to programming constructs. In contrast to our approach no background from psychology was used. Furthermore, no study was done and the scheme differs in general from our comprehension model.

In [215] the problem of semantic defects in a program is proposed to be solved by an ontology. Thereby, the problem is discussed how concepts in the mind of a programmer are mapped to program elements. This is related, because we come up with explanations how a developer comprehends a program. However, no study is done and no background about categorization is presented.

Altogether, we see the model and study not as alternative or as independent from other program comprehension theories. Related work shows already that our model does not stand orthogonal to previous research. Instead, our model links an approved research area in psychology with program comprehension. In contrast to previous comprehension models, we respect the idea that separation of concerns plays a role in program comprehension and relate program comprehension to psychology. Additionally, we present a code study that argues for the validity of our approach. Furthermore, we introduce the idea of concerns and basic concerns, which are based on their usage frequency and importance. The occurrence of multi-concern fragments is explained by cross categorization. Compositional concerns explain the creation of new concerns. Different category types and kinds (Section A.4.3 and Section A.4.4) introduce the idea of similarity to different means of applying separation of concerns. Former models did not take all these facts into consideration. Thus, future work needs to compare in detail program comprehension with categorization.

A.8. Summary and Conclusions

We showed that some constructs in programming are similar to research of categorization in psychology. Examples showed similarities of means to apply separation of concerns and categorization theories. Different categorization theories are similar to some mechanisms to separate concerns. Cross-categorization is comparable to intertwined code fragments representing various concerns simultaneously. Elements can be associated with multiple categories at the same time. Hence, it is natural that not all concerns in source code are separated, too.

We applied our findings of similarities between separation of concerns and categorization and developed a program comprehension model based on research of psychology. Different dimensions of concern comprehension were uncovered and visualized. Finally, a complete model with all the different aspects was presented. In short, the presented comprehension model is consisting out of

two different areas. First: The mental representation where concerns are structured in categories and mapped to source code. Second: A continuous study process that influences the mental model and the comprehension. Vice versa, the current state of comprehension influences the actions of the study process.

We verified the assumption that categorization plays a role in programming, by presenting and extensive study about source code. Thereby, we took advantage of the Hypermodelling approach and computed various indicators about source code. The findings gave indications that packages and the elements located therein share characteristics with categories. Furthermore, we revealed that the package hierarchy, even it has no functional meaning in programming, is likely one used criteria to group types (classes, interfaces, enums) into those. This indicates that categorization is used to structure different elements in the programming process. Therefore, we infer and predict that categorization is done for other elements in programming as well.

Lastly, we presented a discussion about the relation to Hypermodelling and reported our main findings. Thereby, we presented indications that a multi-dimensional perspective towards source code seems natural for humans. Especially the flexible computation capabilities of Hypermodelling can be useful in this context and underline the need for Hypermodelling from another perspective.

However, we see further work to be done. Even though, we did a preliminary evaluation that categorization plays a role in programming, we see the need for further and deeper verifications and evaluations. Also, we restricted our presented research to a few mechanisms to separate concerns and compared them with categorization. This was the case for the evaluation and also for the model and concern- category comparison. A more detailed and consistent comparison needs to be done. In detail, our comparison of categorization and separation of concerns needs to be verified through further investigations and improved through additional composition mechanisms. More quantitative research studies on manifestations of concerns in source code can determine and verify how categories are realized in code. The cited psychological studies have to be transferred and adapted to retrieve corresponding patterns in source code. A possible outcome of how categories manifest in code can be a new concern revealing technique.

On the other hand, we see new impulses through our model and study. Our model enables further research on the role of categorization in program comprehension. With the help of our model, several areas can be identified where additional research is needed. For instance, we assume a difference in comprehension of normal and of basic concerns. Investigations need to verify this assumption. Such investigations can discover methods of detecting such basic concerns automatically.

Finally, the model proposes a comprehension process. We believe that future program investigation tools and IDEs need to construct a category based knowledge model of a developer by using behavior tracking and mining of a developers knowledge out of source code. Hypermodelling seems suitable for this task, because showed an example to mine developer knowledge out of source code in a prior chapter.

Aside all this, different indicator computations of this chapter revealed that statistic computations are another use case for Hypermodelling. Therefore, the contribution of this chapter is not limited in the verification of the application of categorization in programming, but also shows that Hypermodelling is suitable to investigate statistic indicators. Future research can now determine which advanced statistic methods are especially interesting for source code investigations.

Appendix B. Data Sources and Computations

B.1. Lines of Code Sources

We assembled project data about lines of code by a web search to generate the chart in Section 1.3. When we finally recovered lines of code of a project, we searched for the release date of the project in the web. For instance, when we found out the 1,7 million lines of code of an F-22 Raptor jet on a webpage (http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code/0), we looked up in wikipedia when the official shipment started (2002). For this reason, we do not stick to the release years and lines of code as absolute and neither as totally correct. Furthermore, there is also the problem that some lines of code estimations may include unit tests and similar things. Additionally, some projects like Windows XP differed in various sources from 35 million to 40 million. Also, there is also the problem that we mixed up estimations of different experts, because we used different webpages as sources.

Therefore, the following table (Table B.1), listing the data sources of the lines of code, has to be seen as approximation. Further research would have to identify exact numbers of lines of code. In our work, we stick to the fact that modern code bases can be quite complex. If there are a few million more or less, it does not alter our assumption that code bases are quite complex. This way, we consider the approximation as valid for our work.

However, someone may be interested to get into more details and the sources from where we got the lines of code to dig into more details. Therefore, we provide the web addresses of the data origin.

Table B.1. Lines of code estimations and sources

Product	Release year	Million lines of code	Source
Win NT 3.1	1993	4,5	http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/
Win NT 3.5	1994	7,5	http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/
Win NT 3.51	1995	9,5	http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/
Win 95	1995	15	http://www.nytimes.com/2006/03/27/technology/27soft.html
NT 4.0	1996	11,5	http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/
Win 98	1998	18	http://www.nytimes.com/2006/03/27/technology/27soft.html
Windows 2000	1999	29	http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/
Windows XP	2001	40	http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/
F-22 Raptor	2002	1,7	http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code/0
Windows Server 2003	2003	50	http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/
Linux Kernel 2.6.0	2003	5,9	http://en.wikipedia.org/wiki/Linux_kernel
Debian 2.2	2003	55	http://www.dwheeler.com/sloc/
Average embedded device	2003	1	http://www.coverity.com/library/pdf/ControllingSoftwareComplexity.pdf
World of Warcraft	2004	5,5	http://www.gamespot.com/news/blizzard-outlines-massive-effort-behind-world-of-warcraft-6228615

Product	Release year	Million lines of code	Source
Open Office	2004	7	http://www.openoffice.org/tools/dev_docs/OOo_cws.html
Typical cellphone	2005	2	http://spectrum.ieee.org/computing/software/why-software-fails/0
Java JDK	2006	6,5	http://planet.jboss.org/post/java_is_finally_free_and_open
Max OS X	2006	86	http://www.engadget.com/2006/08/07/live-from-wwdc-2006-steve-jobs-keynote/
Win Vista	2007	50	http://www.nytimes.com/2006/03/27/technology/27soft.html
Car radio and navigation system	2009	20	http://www.motorauthority.com/news/1026505_modern-luxury-vehicles-claimed-to-feature-more-software-than-a-fighter-jet
Premium car	100	100	http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code/0
Phillips Magnet Ressoance Imaging Device	2010	8	http://www.architectingforum.org/whitepapers/SAF_WhitePaper_2010_11.pdf
Dreamliner Jumbo Jet	2011	6,5	http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code/0
Eclipse Juno Simultaneous Release	2012	55	http://eclipsehowl.wordpress.com/2012/06/26/juno-the-best-simultaneous-release-of-its-time/
Android	2012	12	http://www.itechwhiz.com/2011/03/google-android-as-top-most-smartphones.html
Linux 3.2	2012	15	http://en.wikipedia.org/wiki/Linux_kernel
Alfresco Content Management System	2012	2	http://www.ohloh.net
JBoss application server	2012	1,2	http://www.ohloh.net
Open JDK	2012	4,1	http://www.ohloh.net
The spring framework	2012	1,1	http://www.ohloh.net

1

B.2. Holistic Code Cube Computations

In order to compute all the relations in Java in a Data Warehouse cube, we present the holistic computation rules for future researchers in the following. The cube contains all the computations of Chapter 6 and assembles them together into a holistic computation structure for Java code.

[1]The whole data was retrieved at 6.7.2012

Dimensions / Measures	TypeMember Facts (A)	Field (B)	FieldTypes (C)	Field ComplexType (D)	Method (E)	Method Parameters (F)	Method Complex Type (G)	MethodCall Facts (H)	Type Member Facts (CalledMethod Hierarchy) (I)	TypeInheritance Facts (J)	AnnotatedMember Facts (K)	AnnotatedTypes Facts (L)
(1) Package Hierarchy	*Ref.: Package(packageHierarchyID – 2B)*	*M:N – TypeMemberFacts (A1)*	*M:N – Field Types (B1)*	*M:N – Field Types (C1)*	*M:N – TypeMemberFacts (A1)*	*M:N – Method (E1)*	*M:N –Method Parameters (F1)*	*M:N – Method (E1)*	*No relation cause it is another hierarchy and the grammar/ap would not be clear this way*	*M:N – TypeMemberFacts (A1)*	*M:N – TypeMemberFacts (A1)*	*Reference Package(packageHierarchyI D – L2)*
(2) Package	*Ref.: File(packageID 3B)*	*M:N – TypeMemberFacts (A2)*	*M:N – Field Types (B2)*	*M:N – Field Types (C2)*	*M:N – TypeMemberFacts (A2)*	*M:N – Method (E2)*	*M:N –Method Parameters (F2)*	*M:N – Method (E2)*		*M:N – TypeMemberFacts (A2)*	*M:N – TypeMemberFacts (A2)*	*Reference File(packageID L3)*
(3) File	*Ref.: Type (fileID – 5A)*	*M:N – TypeMemberFacts (A3)*	*M:N – Field Types (B3)*	*M:N – Field Types (C3)*	*M:N – TypeMemberFacts (A3)*	*M:N – Method (E3)*	*M:N –Method Parameters (F3)*	*M:N – Method (E3)*		*M:N – TypeMemberFacts (A3)*	*M:N – TypeMemberFacts (A3)*	*Reference Type (FileID – L5)*
(4) Type Classification	*Ref.: Type (typeClassificationID – 5A)*	*M:N – TypeMemberFacts (A4)*	*M:N – Field Types (B4)*	*M:N – Field Types (C4)*	*M:N – TypeMemberFacts (A4)*	*M:N – Method (E4)*	*M:N –Method Parameters (F4)*	*M:N – Method (E4)*		*M:N – TypeMemberFacts (A4)*	*M:N – TypeMemberFacts (A4)*	*Reference TypeI(typeClassificationID– L5)*
(5) Type	*Type (typeID)*	*M:N – TypeMemberFacts (A5)*	*M:N – Field Types (B5)*	*M:N – Field Types (C5)*	*M:N – TypeMemberFacts (A5)*	*M:N – Method (E5)*	*M:N –Method Parameters (F5)*	*M:N – Method (E4)*		*M:N – TypeMemberFacts (A5)*	*M:N – TypeMemberFacts (A5)*	*Type (typeID)*
(6) Member	*Member (memberID)*	*Ref.: Field (B8)*	*Ref.: Field(C8)*	*M:N – Field Types (C6)*	*Ref.: Method(memberID - E15)*	*M:N –Method Parameters (F6)*	*M:N – Field Types (F7)*	*Ref.: Method (memberID - H15)*		*M:N – TypeMemberFacts (A5)*	*Member (memberID)*	*M:N – TypeMemberFacts (A6)*
(7) Member Classification	*Ref.: Member(memberClassification ID – 6A)*	*Ref.: Member (B6)*	*Ref.: Member (C6)*	*M:N – Field Types (C7)*	*Ref.: Member (E6)*	*Ref.: Member (F6)*		*Ref.: Member (H6)*		*M:N – TypeMemberFacts (A5)*	*Reference Member (memberClassificationID – K6)*	*M:N – TypeMemberFacts (A7)*
Fields												
(8) Field	*M:N – Field (B8)*	*Field (id)*	*Field (id)*	*M:N – Field Types (C8)*						*M:N – TypeMemberFacts (A8)*	*M:N – TypeMemberFacts (A8)*	*M:N – TypeMemberFacts (A8)*
(9) Field - Abstract Type (Join with Type)	*M:N – Field (B9)*	*M:N – Field Types (C9)*	*AbstractType (abstract TypeID)*	*AbstractType (abstractTypeID)*						*M:N – TypeMemberFacts (A9)*	*M:N – TypeMemberFacts (A9)*	*M:N – TypeMemberFacts (A9)*
(10) Field – Type Classification	*M:N – Field (B10)*	*M:N – Field Types (C10)*	*Ref.: Field - Abstract Type(typeclass ification ID - C9)*	*Ref.: Field - Abstract Type(typeclass ification ID - D9)*						*M:N – TypeMemberFacts (A10)*	*M:N – TypeMemberFacts (A10)*	*M:N – TypeMemberFacts (A10)*
(11) Field –Type	*M:N – Field (B11)*	*M:N – Field (C11)*	*M:N – Field (D11)*	*Field Type (id)*						*M:N – TypeMemberFacts (A11)*	*M:N – TypeMemberFacts (A11)*	*M:N – TypeMemberFacts (A11)*
(12) Field – File	*M:N – Field (B11)*	*M:N – Field Types (C11)*	*M:N – Field Types (D12)*	*Ref.: Field Complex Type (fileID – D11)*						*M:N – TypeMemberFacts (A12)*	*M:N – TypeMemberFacts (A12)*	*M:N – TypeMemberFacts (A12)*
(13) Field - Package	*M:N – Field (B12)*	*M:N – Field Types (C12)*	*M:N – Field Types (D13)*	*Ref.: Field - File (packageID – D12)*						*M:N – TypeMemberFacts (A13)*	*M:N – TypeMemberFacts (A13)*	*M:N – TypeMemberFacts (A13)*
(14) Field - Package Hierarchy	*M:N – Field (B13)*	*M:N – Field Types(C13)*	*M:N – Field Types(D14)*	*Ref.: Field - Package (packageHierarchyID – D13)*						*M:N – TypeMemberFacts (A14)*	*M:N – TypeMemberFacts (A14)*	*M:N – TypeMemberFacts (A14)*

No relation because fields and methods are distinct members

no relation cause it is within the method

	TypeMember Facts (A)	Field (B)	FieldTypes (C)	Field ComplexType (D)	Method (E)	Method Parameters (F)	Method Complex Type (G)	MethodCall Facts (H)	Type Member Facts (CalledMethod Hierarchy) (I)	TypeInheritance Facts (J)	AnnotatedMember Facts (K)	AnnotatedTypes Facts (L)
Method parameters												
(15) Method	*M:N – Method (E15)*	*No relation because fields and methods are distinct members*			*Method (id)*	*Method (id)*	*M:N – Method Parameters (F15)*	*Method (caller)*		*M:N – TypeMemberFacts (A15)*	*M:N – TypeMemberFacts (A15)*	*M:N – TypeMemberFacts (A15)*
(16) Method Parameters	*M:N – Method (E16)*				*M:N – Method Parameters (F16)*	*Method Parameters (id)*	*M:N – Method Parameters (F16)*	*M:N – Method (E16)*		*M:N – TypeMemberFacts (A16)*	*M:N – TypeMemberFacts (A16)*	*M:N – TypeMemberFacts (A16)*
(17) Method - Abstract Type (Join with Type)	*M:N – Method (E17)*				*M:N – Method Parameters (F17)*	*AbsractType (abstractTypeID)*	*AbstractType (abstractTypeID)*	*M:N – Method (E17)*		*M:N – TypeMemberFacts (A17)*	*M:N – TypeMemberFacts (A17)*	*M:N – TypeMemberFacts (A17)*
(18) Method - Type Classification	*M:N – Method (E18)*				*M:N – Method Parameters (F18)*	*Reference Method Parameters - Abstract Type (typeclassificationID – F17)*	*Ref.: Method - Abstract Type (typeclassification ID – G17)*	*M:N – Method (E18)*		*M:N – TypeMemberFacts (A18)*	*M:N – TypeMemberFacts (A18)*	*M:N – TypeMemberFacts (A18)*
(19) Method – Complex Type	*M:N – Method (E19)*				*M:N – Method Parameters (F19)*	*Reference Method Parameters - Abstract Type (typeclassficationID – F17)*	*Ref.: Method - Abstract Type (typeID– G17)*	*M:N – Method (E19)*	*No relation cause it is another hierarchy and the granularity would not be clear this way*	*M:N – TypeMemberFacts (A19)*	*M:N – TypeMemberFacts (A19)*	*M:N – TypeMemberFacts (A19)*
(20) Method – File	*M:N – Method (E20)*				*M:N – Method Parameters (F20)*	*M:N – Method ComplexType (G20)*	*Ref.: Method-ComplexType (fileID – G19)*	*M:N – Method (E20)*		*M:N – TypeMemberFacts (A20)*	*M:N – TypeMemberFacts (A20)*	*M:N – TypeMemberFacts (A20)*
(21) Method- Package	*M:N – Method (E21)*				*M:N – Method Parameters (F21)*	*M:N – Method ComplexType (G21)*	*Ref.: Method - File (packageID – G20)*	*M:N – Method (E21)*		*M:N – TypeMemberFacts (A21)*	*M:N – TypeMemberFacts (A21)*	*M:N – TypeMemberFacts (A21)*
(22) Method - Package Hierachy	*M:N – Method (E22)*				*M:N – Method Parameters (F22)*	*M:N – Method ComplexType (G22)*	*Ref.: Method - Package (packageHierarchyID – G21)*	*M:N – Method (E22)*		*M:N – TypeMemberFacts (A22)*	*M:N – TypeMemberFacts (A22)*	*M:N – TypeMemberFacts (A22)*
Method calls												
(23) Called Method	*M:N – Method (E23)*	*No relations because they are distinct members to fields*			*M:N – Method Calls (H23)*	*M:N – Method (E23)*	*M:N – Method Parameters(F23)*	*Method (called)*	*Method (memberID)*	*M:N – TypeMemberFacts (A23)*	*M:N – TypeMemberFacts (A23)*	*M:N – TypeMemberFacts (A23)*
(24) Called Method - Type	*M:N – Method (E24)*				*M:N – Method Calls (H24)*	*M:N – Method (E24)*	*M:N – Method Parameters (F24)*	*M:N – TypeMemberFacts (CalledMethodHierarchy) (I24)*	*Type (typeID)*	*M:N – TypeMemberFacts (A24)*	*M:N – TypeMemberFacts (A24)*	*M:N – TypeMemberFacts (A24)*
(25) Called Method – Type Classification	*M:N – Method (E25)*				*M:N – Method Calls (H25)*	*M:N – Method (E25)*	*M:N – Method Parameters (F25)*	*M:N – TypeMemberFacts (CalledMethodHierarchy) (I25)*	*Ref.: Called Method- Type (typeclassificationID– I24)*	*M:N – TypeMemberFacts (A25)*	*M:N – TypeMemberFacts (A25)*	*M:N – TypeMemberFacts (A25)*
(26) Called Method – File	*M:N – Method (E26)*				*M:N – Method Calls (H26)*	*M:N – Method (E26)*	*M:N – Method Parameters (F26)*	*M:N – TypeMemberFacts (CalledMethodHierarchy) (I26)*	*Ref.: Called Method- Type (fileID – I24)*	*M:N – TypeMemberFacts (A26)*	*M:N – TypeMemberFacts (A26)*	*M:N – TypeMemberFacts (A26)*
(27) Called Method - Package	*M:N – Method (E27)*				*M:N – Method Calls (H27)*	*M:N – Method (E27)*	*M:N – Method Parameters (F27)*	*M:N – TypeMemberFacts (CalledMethodHierarchy) (I27)*	*Ref.: Called Method - File (packageID – I26)*	*M:N – TypeMemberFacts (A27)*	*M:N – TypeMemberFacts (A27)*	*M:N – TypeMemberFacts (A27)*
(28) Called Method - Package Hierachy	*M:N – Method (E28)*				*M:N - Method Calls (H28)*	*M:N – Method (E28)*	*M:N – Method Parameters (F28)*	*M:N – TypeMemberFacts (CalledMethodHierarchy) (I28)*	*Ref.: Called Method - Package (packageHierarchyID – I27)*	*M:N – TypeMemberFacts (A27)*	*M:N – TypeMemberFacts (A27)*	*M:N – TypeMemberFacts (A27)*

Inheritance	TypeMember Facts (A)	Field (B)	FieldTypes (C)	Field ComplexType (D)	Method (E)	Method Parameters (F)	Method Complex Type (G)	MethodCall Facts (H)	Type Member Facts (CalledMethod Hierarchy) (I)	TypeInheritance Facts (J)	AnnotatedMember Facts (K)	AnnotatedTypes Facts (L)
(29) Parent - Type	*M:N – Method (E29)*	*M:N – TypeMemberFacts (A29)*	*M:N – Field (B29)*	*M:N – Field Types (C29)*	*M:N – TypeMemberFacts (A29)*	*M:N – Method (E29)*	*M:N –Method Parameters (F29)*	*M:N – Method (E29)*		*Type (parentID)*	*M:N – TypeMemberFacts (A29)*	*M:N – TypeMemberFacts (A29)*
(30) Called Method – Type Classification	*M:N – Method (E30)*	*M:N – TypeMemberFacts (A30)*	*M:N – Field (B30)*	*M:N – Field Types (C30)*	*M:N – TypeMemberFacts (A30)*	*M:N – Method (E30)*	*M:N –Method Parameters (F30)*	*M:N – Method (E30)*	*No relation cause it is another hierarchy and the granularity would not be clear this way.*	*Ref.: Parent Type (typeclassification ID – 129)*	*M:N – TypeMemberFacts (30)*	*M:N – TypeMemberFacts (30)*
(31) Parent – File	*M:N – Method (E31)*	*M:N – TypeMemberFacts*	*M:N – Field (B31)*	*M:N – Field Types (C31)*	*M:N – TypeMemberFacts*	*M:N – Method (E31)*	*M:N –Method Parameters (F31)*	*M:N – Method (E31)*		*Ref.: Parent- Type (fileID – 129)*	*M:N – TypeMemberFacts (A31)*	*M:N – TypeMemberFacts (A31)*
(32) Called Method - Package	*M:N – Method (E32)*	*M:N – TypeMemberFacts (A32)*	*M:N – Field (B32)*	*M:N – Field Types (C32)*	*M:N – TypeMemberFacts (A32)*	*M:N – Method (E32)*	*M:N –Method Parameters (F32)*	*M:N – Method (E32)*		*Ref.: Parent - File (packageID – 131)*	*M:N – TypeMemberFacts (A32)*	*M:N – TypeMemberFacts (A32)*
(33) Called Method - Package Hierarchy	*M:N – Method (E33)*	*M:N – TypeMemberFacts (A33)*	*M:N – Field (B33)*	*M:N – Field Types (C33)*	*M:N – TypeMemberFacts (A33)*	*M:N – Method (E32)*	*M:N –Method Parameters (F32)*	*M:N – Method (E33)*		*Ref.: Parent - Package (packageHierarchyID – 132)*	*M:N – TypeMemberFacts (A33)*	*M:N – TypeMemberFacts (A33)*
Member Annotation												
(34) Type Annotation - Type	*M:N – Method (E34)*	*M:N – TypeMemberFacts (A34)*	*M:N – Field (B34)*	*M:N – Field Types (C34)*	*M:N – TypeMemberFacts (A34)*	*M:N – Method (E34)*	*M:N –Method Parameters (F34)*	*M:N – Method (E34)*		*M:N – TypeMemberFacts (A34)*	*Member Annotation - Type (annotationTypeID)*	*M:N – TypeMemberFacts (A34)*
(35) Type Annotation – File	*M:N – Method (E35)*	*M:N – TypeMemberFacts (A35)*	*M:N – Field (B35)*	*M:N – Field Types (C35)*	*M:N – TypeMemberFacts (A35)*	*M:N – Method (E35)*	*M:N –Method Parameters (F35)*	*M:N – Method (E35)*	*No relation cause it is another hierarchy and the granularity would not be clear this way.*	*M:N – TypeMemberFacts (A35)*	*Reference Member Annotation- TypeFileID – K34)*	*M:N – TypeMemberFacts (A35)*
(36) Type Annotation - Package	*M:N – Method (E36)*	*M:N – TypeMemberFacts (A36)*	*M:N – Field (B36)*	*M:N – Field Types (C36)*	*M:N – TypeMemberFacts (A36)*	*M:N – Method (E36)*	*M:N –Method Parameters (F36)*	*M:N – Method (E36)*		*M:N – TypeMemberFacts (A36)*	*Reference Member Annotation- File(packageID K35)*	*M:N – TypeMemberFacts (A36)*
(37) Type Annotation - Package Hierarchy	*M:N – Method (E37)*	*M:N – TypeMemberFacts (A37)*	*M:N – Field (B37)*	*M:N – Field Types (C37)*	*M:N – TypeMemberFacts (A37)*	*M:N – Method (E37)*	*M:N –Method Parameters (F37)*	*M:N – Method (E37)*		*M:N – TypeMemberFacts (A37)*	*Reference Member Annotation - Package(packageHierarchyID – K36)*	*M:N – TypeMemberFacts (A37)*
Type Annotation												
(38) Member Annotation - Type	*M:N – Method (E38)*	*M:N – TypeMemberFacts (A38)*	*M:N – Field (B38)*	*M:N – Field Types (C38)*	*M:N – TypeMemberFacts (A38)*	*M:N – Method (E38)*	*M:N –Method Parameters (F38)*	*M:N – Method (E38)*		*M:N – TypeMemberFacts (A38)*	*M:N – TypeMemberFacts (A38)*	*Type Annotation - Type (annotationTypeID)*
(39) Member Annotation – File	*M:N – Method (E39)*	*M:N – TypeMemberFacts (A39)*	*M:N – Field (B39)*	*M:N – Field Types (C39)*	*M:N – TypeMemberFacts (A39)*	*M:N – Method (E39)*	*M:N –Method Parameters (F39)*	*M:N – Method (E39)*	*No relation cause it is another hierarchy and the granularity would not be clear this way.*	*M:N – TypeMemberFacts (A39)*	*M:N – TypeMemberFacts (A39)*	*Reference Type Annotation- TypeFileID – L38)*
(40) Member Annotation - Package	*M:N – Method (E40)*	*M:N – TypeMemberFacts (A40)*	*M:N – Field (B40)*	*M:N – Field Types (C40)*	*M:N – TypeMemberFacts (A40)*	*M:N – Method (E40)*	*M:N –Method Parameters (F40)*	*M:N – Method (E40)*		*M:N – TypeMemberFacts (A40)*	*M:N – TypeMemberFacts (A40)*	*Reference Type Annotation- FilepackageID L39)*
(41) Member Annotation - Package Hierarchy	*M:N – Method (E41)*	*M:N – TypeMemberFacts (A41)*	*M:N – Field (B41)*	*M:N – Field Types (C41)*	*M:N – TypeMemberFacts (A41)*	*M:N – Method (E41)*	*M:N –Method Parameters (F41)*	*M:N – Method (E41)*		*M:N – TypeMemberFacts (A41)*	*M:N – TypeMemberFacts (A41)*	*Reference Type Annotation - PackageHierarchyID – L40)*

References

Sorted by primary authors. References without known authors first. Hyperlinks checked at 7.11.2012

[1] *SEON - A Family of Software Evolution ONtologies*. http://www.se-on.org/. Software Evolution and Architecture Lab. University of Zürich.

[2] *Eclipse Code Recommenders*. http://www.eclipse.org/recommenders/. The Eclipse Foundation. 2012.

[3] *MDX Language Reference*. http://msdn.microsoft.com/en-us/library/ms145595.aspx. Microsoft Corporation. 2012.

[4] *XMLforAnalysis.com*. http://xmlforanalysis.com. Simba Technologies.

[5] *Microsoft SQL Server 2012 Homepage*. http://technet.microsoft.com/en-us/sqlserver. Microsoft Corporation.

[6] *Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces*. http://www.research.ibm.com/hyperspace/. IBM.

[7] *Eclipse Modeling Project*. http://www.eclipse.org/modeling. The Eclipse Foundation.

[8] *OMG MDA Homepage*. http://www.omg.org/mda/. Object Management Group, Inc..

[9] *MetaObjectFacility (MOF) Specification*. http://www.omg.org/mof/. Object Management Group, Inc..

[10] *XML Schema*. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/. W3C.

[11] *XHTML 1.1 XML Schema Definition*. http://www.w3.org/TR/xhtml11/xhtml11_schema.html. W3C.

[12] *OSGi Alliance Homepage*. http://www.osgi.org/. OSGi™ Alliance.

[13] *OSGi Service Platform Core Specification*. http://www.osgi.org/download/r4v41/r4.core.pdf. 2007.

[14] *Eclipse Equinox Homepage*. http://www.eclipse.org/equinox/. The Eclipse Foundation.

[15] *The AspectJ Programming Guide*. http://www.eclipse.org/aspectj/doc/released/progguide/index.html. Xerox Corporation.

[16] *AspectJ Homepage*. http://www.eclipse.org/aspectj/. The Eclipse Foundation.

[17] *Aspect-Oriented Research on Composition Filters homepage*. http://trese.cs.utwente.nl/oldhtml/composition_filters/. Department of Computer Science of the University of Twente, The Netherlands..

[18] *Java Platform, Enterprise Edition (Java EE) Technical Documentation*. http://docs.oracle.com/javaee/. Oracle.

[19] *Java Persistence API*. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html. Oracle.

[20] *pentaho™open source business intelligence*. http://www.pentaho.com/. Pentaho Corporation.

[21] *JQuery a query-based code browser*. http://jquery.cs.ubc.ca/index.htm. The University of British Columbia Vancouver, Canada .

[22] *Homepage of the main Spring Framework manufacturer*. http://www.springsource.org.

[23] *Eclipse Java development tools (JDT) Overview*. http://www.eclipse.org/jdt/overview.php. Eclipse Foundation.

[24] *AJDT: AspectJ Development Tools*. http://www.eclipse.org/ajdt/demos/VisualiserDemo.html. Eclipse Foundation.

[25] *Spring Tool Suite*. http://www.springsource.org/sts. SpringSource.

[26] *Mylyn/Architecture*. http://wiki.eclipse.org/Mylyn/Architecture. The Eclipse Foundation.

[27] *AJDT: AspectJ Development Tools*. http://www.eclipse.org/ajdt/. The Eclipse Foundation.

[28] *JDT Core Component*. http://www.eclipse.org/jdt/core/index.php. Eclipse Foundation.

[29] *semmle / code*. http://semmle.com/semmlecode/. Semmle. Semmle Limited.

[30] *NDepend Homepage*. http://www.ndepend.com/. SMACCHIA.COM.

[31] *JavaDepend /Jarchitect Homepage*. http://www.javadepend.com. http://jarchitect.com/.

[32] M. Aksit and L. Bergmans. *Obstacles in object-oriented software development*. OOPSLA '92 conference proceedings on Object-oriented programming systems, languages, and applications. ACM. 1992. 341-358.

[33] A. Andresen. *Komponentenbasierte Softwareentwicklung mit MDA, UML 2 und XML*. 978-3446229150. Hanser Verlag. 2004.

[34] S. Apel. *The role of features and aspects in software development: similarities, differences, and synergetic potential. PhD thesis*. Otto von Guericke University, Magdeburg, Germany. 2007.

[35] S. L. Armstrong, L. R. Gleitman, H. Gleitman. *What some concepts might not be*. Cognition, Volume 13, Issue 3. 1983. 263-308.

[36] C. Aschwanden and M. Crosby. *Code scanning patterns in program comprehension*. In Proceedings of the 39th Hawaii International Conference on System Sciences. 2006.

[37] C. Atkinson, D. Brenner, P. Bostan et al.. *Modeling Components and Component-Based Systems in KobrA*. The Common Component Modeling Example. Springer. 2008. 54-84.

[38] C. Atkinson and D. Stoll. *An Environment for the Orthographic Modeling of Workflow Components*. Proceedings of the Prozessinnovationen mit Unternehmenssoftware (PRIMIUM) Subconference at the Multikonferenz Wirtschaftsinformatik (MKWI). ceur-ws.org. 2008.

[39] C. Atkinson and D. Stoll. *Orthographic Modeling Environment*. Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering. Springer. 2008. 93-96.

[40] C. Atkinson, J. Bayer, C. Bunse. *Component-based Product Line Engineering with UML*. 978-0201737912. Addison-Wesley. 2001.

[41] S. Bajracharya, J. Ossher, C. Lopes. *Searching API Usage Examples in Code Repositories with Sourcerer API Search*. Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation. ACM. 2010. 5-8.

[42] K. S. Bajracharya, J. Ossher, C. Lopes. *Sourcerer - An Infrastructure for Large-scale Collection and Analysis of Open-source Code*. Third International Workshop on Academic Software Development Tools and Techniques (WASDeTT-3). 2010.

[43] P. F. Baldi, C. Lopes, E. J. Linstead et al.. *A theory of aspects as latent topics*. Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. 2008. 543-562 .

[44] L. W. Barsalou. *Ad hoc categories*. Memory and cognition, Volume 11. 1983. 211-227.

[45] M. Bennicke, F. Steinbrückner, M. Radicke et al.. *Das sd&m Software Cockpit: Architektur und Erfahrungen*. Workshop on Applied Program Analysis 2007, 37. Jahrestagung der Gesellschaft für Informatik e.V. (GI). Springer. 2007. 254-260.

[46] J. A. Bloch. *JSR-000175 A Metadata Facility for the Java Programming Language*. http://jcp.org/ja/jsr/detail?id=175. http://jcp.org/aboutJava/communityprocess/final/jsr175/index.html. Sun Microsystems. 2004.

[47] G. Booch, R. A. Maksimchuk, M. W. Engel et al.. *Object-oriented analysis and design with applications*. 978-0805353402. Addison-Wesley. 1993.

[48] A. Bragdon, S. P. Reiss, R. Zeleznik et al.. *Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments*. Proceedings of the 32nd International Conference on Software Engineering (ICSE). ACM. 2010. 455-464 .

[49] A. Bragdon, S. P. Reiss, R. Zeleznik et al.. *Code Bubbles Research Demonstration*. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE). ACM. 2010. 293-296.

[50] R. Brooks. *Towards a theory of the comprehension of computer programs*. International Journal of Man-Machine Studies, Volume 18. 1983. 543-554.

[51] R. Brooks. *Using a Behavioral Theory of Program Comprehension in Software Engineering*. Proceedings of the 3rd international conference on Software engineering (ICSE). IEEE. 1978. 196-201.

[52] L. R. Brooks, G. R. Norman, S.W. Allen. *Role of specific similarity in a medical diagnostic task*. Journal of Experimental Psychology General, Volume 120. 1991. 278-287.

[53] W. J. Brown, R. C. Malveau, H. W. McCormick et al.. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 978-0471197133. Addison Wesley. 1998.

[54] M. Bruch, M. Mezini, M. Monperrus. *Mining subclassing directives to improve framework reuse*. Proceedings of 7th IEEE Working Conference on Mining Software Repositories. IEEE. 2010. 141–150.

[55] J.S. Bruner, J. Goodnow, G. Austin. *A study of thinking*. 978-0887386565. Wiley. 1986.

[56] F. Budinsky, D. Steinberg, R. Ellersick et al.. *Eclipse modeling framework: a developer's guide*. 978-0131425422. Addison-Wesley. 2003.

[57] D. Bulos and S. Forsman. *Getting Started with ADAPT*. http://www.symcorp.com/downloads/ADAPT_white_paper.pdf. Symmetry Corporation. 1998.

[58] S. Cameron. *Microsoft® SQL Server® 2008 Analysis Services Step by Step*. 978-0735626201. mitp Professional. 2009.

[59] M. Chapman. *Making AspectJ development easier with AJDT*. http://www.infoq.com/articles/aspectj-with-ajdt. InfoQ.com. 2006.

[60] R Chern. *Masterthesis: Reducing Remodularization Complexity Through Modular-Objective Decoupling*. http://www.cs.ubc.ca/grads/resources/thesis/Nov08/Chern_Rick.pdf. The University of British Columbia, Vancouver Canada. 2008.

[61] J-D. Choi and A. Zeller. *Isolating Failure-Inducing Thread Schedules*. Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '02) . ACM. 2002. 210-220.

[62] K. Chow and D. Notkin. *Semi-automatic update of applications in response to library changes*. Proceedings of the International Conference on Software Maintenance. IEEE. 1996.

[63] W. Chung, W. Harrison, V. Kruskal et al.. *Working with Implicit Concerns in the Concern Manipulation Environment*. AOSD ’05 Workshop on Linking Aspect Technology and Evolution (LATE). 2005. .

[64] M. Ciolkowski, J. Heidrich, F. Simon et al.. *Empirical results from using custom-made software project control centers in industrial environments*. Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. ACM. 2008. 243-252.

[65] S. Clarke, W. Harrison, H. Ossher et al.. *Subject-oriented design: towards improved alignment of requirements, design, and code*. Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM. 1999. 325-339.

[66] C. Clayberg and D. Rubel. *Eclipse: building commercial-quality plug-ins*. 978-0321228475. Addison-Wesley. 2004.

[67] B. Cleary and C. Exton. *Assisting Concept Assignment Using Probabilistic Classification and Cognitive Mapping*. In Proceedings of the 2nd International Workshop on Supporting Knowledge Collaboration in Software Development (KSCD 2006) collocated with the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). 2006.

[68] B. Cleary and C. Exton. *Assisting Concept Location in Software Comprehension*. 19th Annual Workshop of the Psychology of Programming Interest Group. 2007.

[69] A. Clement, A. Colyer, M. Kersten. *Aspect-Oriented Programming with AJDT*. Workshop on Analysis of Aspect-Oriented Software at ECOOP ‘03. Springer. 2003. 1-6.

[70] M. Clifton. *What is a Framework?*. http://www.codeproject.com/KB/architecture/WhatIsAFramework.aspx. The Code Project.

[71] E. F. Codd, S. B. Codd, C.T. Salley. *Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate*. Codd and Associates. http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf. Wiley. 1993.

[72] A. Colyer, A. Clement, G. Harley et al.. *Eclipse AspectJ:*. 0-321-24587-3. Addison-Wesley. 2004.

[73] M. E. Crosby, J. Scholtz, S. Wiedenbeck. *The roles beacons play in comprehension for novice and expert programmers*. In Proceedings of the 14th Workshop of the Psychology of Programming Interest Group. 2002. 58-73.

[74] B. de Alwis, G.C. Murphy, M. P. Robillard. *A Comparative Study of Three Program Exploration Tools*. 15th IEEE International Conference on Program Comprehension (ICPC '07). IEEE. 103-112.

[75] B. de Alwis and G. C. Murphy. *Answering Conceptual Queries with Ferret*. ICSE '08 Proceedings of the 30th international conference on Software engineering. ACM. 2008. 21-30.

[76] B. de Alwis. *Supporting Conceptual Queries Over Integrated Sources of Program Information. Ph.D. thesis*. https://dspace.library.ubc.ca/dspace/bitstream/2429/695/1/ubc_2008_spring_dealwis_brian.pdf. The University of British Columbia, Vancouver Canada. 2008.

[77] K. De Volder. *JQuery: A Generic Code Browser with a Declarative Configuration Language*. PADL'06 Proceedings of the 8th international conference on Practical Aspects of Declarative Languages. Springer. 2006. 88–102.

[78] R. Deline and K. Rowan. *Code canvas: Zooming towards better development environments*. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. ACM. 2010. 207-210.

[79] F. Detienne. *Software Design– Cognitive Aspects*. 978-1852332532. Springer. 2002.

[80] E. W. Dijkstra. *On the role of scientific thought*. Selected Writings on Computing: A Personal Perspective. Springer. 1982. 60-66.

[81] R. Dumke. *Software Engineering: Eine Einführung für Informatiker und Ingenieure: Systeme, Erfahrungen, Methoden, Tools*. 978-3528353551. Vieweg+Teubner Verlag. 2003.

[82] T. Elrad, M. Aksit, G. Kiczales et al.. *Discussing aspects of AOP*. Communications of the ACM, Volume 44 Issue 10. ACM. 2001. 33-38.

[83] A. Endres and D. Rombach. *A Handbook of Software and Systems Engineering, Empirical Observations, Laws, and Theories, Person Education*. 978-0321154200. Addison Wesley. 2003.

[84] M. E. Fayad, R. E. Johnson, D. C. Schmidt. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. 978-0471248750. John Wiley and Sons. 1999.

[85] J. Feigenspan, C. Kästner, S. Apel et al.. *Do Background Colors Improve Program Comprehension in the #ifdef Hell?*. Empirical Software Engineering. Springer. 2012.

[86] S. Few. *Information Dashboard Design: The Effective Visual communication of Data*. 978-0596100162. O'Reilly Media. 2006.

[87] R. E. Filman, T. Elrad, S. Clarke et al.. *Aspect-Oriented Software Development*. 978-0321219763. Addison-Wesley. 2004.

[88] M. Fischer, M. Pinzger, H. Gall. *Populating a Release History Database from Version Control and Bug Tracking Systems*. ICSM '03 Proceedings of the International Conference on Software Maintenance. IEEE. 2003. 23-32.

[89] G. Fischer and J. Wolff von Gudenberg. *Programmieren in Java 1. 5: Ein kompaktes, interaktives Tutorial*. 978-3540231349. Springer. 2005.

[90] B. Fluri, M. Wuersch, M. Pinzger et al.. *Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction*. IEEE Transactions on Software Engineering , Volume 33 Issue 11. IEEE. 2007. 725-743.

[91] B. Fluri, M. Würsch, M. Pinzger et al.. *Change distilling: Tree differencing for fine-grained source code change extraction*. IEEE Transactions on Software Engineering (TSE), Volume 33, Issue 11.. IEEE. 2007. 725-743.

[92] M. Fontoura. *Dimension Templates: Multi-dimensional separation of concerns in UML*. http://fontoura.org/papers/mdsoc99.pdf. 1999.

[93] M. Fowler. *Patterns of enterprise application architecture*. 978-0321127426. Addison-Wesley. 2003.

[94] T. Frey, M. Gelhausen, G. Saake. *Categorization of concerns: a categorical program comprehension model*. Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools. ACM. 2011. 73-82.

[95] T. Frey and M. Gelhausen. *Strawberries are nuts*. Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering. ACM. 2011. 49.

[96] T. Frey, M. Gelhausen, H. Sorgatz et al.. *On the Role of Human Thought – Towards A Categorial Concern Comprehension*. Proceedings of the 2nd workshop on Free composition @ onward! 2011. ACM. 2011. 1-6.

[97] T. Frey. *Hypermodelling for Drag and Drop Concern Queries*. Proceedings of Software Engineering 2010 (SE2012). Gesellschaft für Informatik (GI) (Berlin). Springer. 2012. 107-118.

[98] T. Frey and V. Köppen. *Hypermodelling Live - OLAP for Code Clone Recommendation*. Proceedings of Baltic DB & IS 2012. Tenth International Baltic Conference on Databases and Information Systems . CEUR-WS.org. 2012. 37-44.

[99] T. Frey and M. Gräf. *Data-Warehouse-Infrastruktur zur Codeanalyse*. JavaSPEKTRUM 6/2012. SIGS DATACOM GmbH. 2012.

[100] T. Frey. *Hypermodelling - A Data Warehouse Approach for Software Analysis*. Magdeburger Informatik Tage (MIT). University of Magdeburg. 2012. 9-17.

[101] T. Frey, V. Köppen, G. Saake. *Hypermodelling - Introducing Multi-dimensional Concern Reverse Engineering*. Proceedings of the 2nd International ACM/GI Workshop on Digital Engineering (IWDE). University of Magdeburg. 2011. 58-66.

[102] T. Frey. *Vorschlag Hypermodelling: Data Warehousing für Quelltext*. Proceedings of the 23rd GI Workshop on Foundations of Databases. CEUR-WS. 2011. 55-60.

[103] T. Frey and M. Gräf. *Hypermodelling Reporting: Towards Cockpits for Code Structure*. Proceedings of Theory and Practice of Computer Science - 39th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM). Springer. 2013.

[104] T. Frey and V. Köppen. *Exploring Software Variance with Hypermodelling - An exemplary approach*. In 5. Arbeitstagung Programmiersprachen (ATPS'12). im Rahmen der Software Engineering 2012 (SE2012). Gesellschaft für Informatik (GI). Springer. 2012. 121-140.

[105] T. Fritz, J. Ou, G. C. Murphy et al.. *A degree-of-knowledge model to capture source code familiarity*. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE). ACM. 2010. 385-394.

[106] E. Gamma, R. Helm, R. E. Johnson et al.. *Design Patterns*. 978-0201633610. Addison-Wesley. 1995.

[107] M. W. Godfrey, A. E. Hassan, J. D. Herbsleb et al.. *Future of mining software archives: A roundtable*. IEEE Software, Volume 26 Issue 1. IEEE. 2009. 67-70 .

[108] M. Golfarelli, D. Maio, S. Rizzi. *The Dimensional Fact Model: A Conceptual Model For Data Warehouses*. International Journal of Cooperative Information Systems, Volume 7. 1998. 215-247.

[109] J. M. Gómez, C. Rautenstrauch, P. Cissek. *Einführung in Business Intelligence mit SAP NetWeaver 7.0*. 783-540795360. Springer . 2008.

[110] M. Grand. *Patterns in Java: a catalog of reusable design patterns illustrated with UML - Volume I*. 978-0471227298. John Wiley and Sons. 2002.

[111] S. G. Tzafestas, R. Vichnevetsky, U. Grude. *Java ist eine Sprache: Java lesen, schreiben und ausführen - Eine präzise und verständliche Einführung*. 978-3528059149. Vieweg +Teubner. 2005.

[112] V. Gruhn, D. Pieper, C. Röttgers. *MDA: Effektives Softwareengineering mit UML2 und Eclipse*. 978-3540287445. Springer. 2006. 530.

[113] R. E. Hajiyev, M. Verbaere, O. de Moor. *Codequest: scalable source code queries with datalog*. ECOOP'06 Proceedings of the 20th European conference on Object-Oriented Programming. Springer. 2006. 2-27.

[114] J. Han, M. Kamber, J. Pei. *Data Mining: Concepts and Techniques*. 978-0123814791. Morgan Kaufmann. 2011.

[115] J. Hannemann and G. Kiczales. *Design Pattern Implementation in Java and AspectJ*. OOPSLA '02 Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM. 2002. 161-173.

[116] U. Hannig. *Knowledge Management und Business Intelligence*. 978-3540428046. Springer. 2002.

[117] W. Harrison and H. Ossher. *Subject-oriented programming: a critique of pure objects*. Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. ACM. 1993. 411-428.

[118] A. E. Hassan and T. Xie. *Future of Mining Software Engineering Data*. Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM. 2010. 161-166 .

[119] J. Heidrich, J. Münch, J. Wickenkamp. *Practical Guidelines for Introducing Software Cockpits in Industry*. Proceedings of the 5th Software Measurement European Forum. 2009. 49-64.

[120] J. Heidrich, J. Münch, A. Wickenkamp. *Usage Scenarios for Measurement-based Project Control*. Proceedings of the 3rd Software Measurement European Forum. 2006. 47-60.

[121] A. Hemrajani. *Agile Java Entwicklung mit Spring, Hibernate und Eclipse*. 978-3826616969. mitp. 2007.

[122] M. Hennig and H. Seeberger. *Einführung in den "Extension Point" Mechanismus von Eclipse*. JavaSPEKTRUM 1. 2008. sigs-datacom. 2008.

[123] A. Hevner, S. March, J. Park et al.. *Design Science in Information Systems Research*. MIS Quarterly, Volume 28 Issue 1. 2004. 75-105.

[124] D. Hovemeyer and W. Pugh. *Finding bugs is easy*. ACM SIGPLAN Notices, Volume 39 Issue 12 . ACM. 2004. 92-106.

[125] O. Hummel. *Semantic Component Retrival in Software Engineering*. PhD Thesis. Fakultät für Mathematik und Informatik, Universität Mannheim. 2008.

[126] O. Hummel, W. Janjic, C. Atkinson. *Code Conjurer: Pulling Reusable Software out of Thin Air*. IEEE Software, Volume 25, Issue 5. IEEE. 2008. 45-52 .

[127] W. L. Hursch and C. Lopes. *Separation of Concerns*. ftp://ftp.ccs.neu.edu/pub/people/lieber/crista/techrep95/separation.pdf. College of Computer Science, Northeastern University Boston, MA 02115, USA. 1995.

[128] W. H. Inmon. *Building the Data Warehouse*. 978-0764599446. Springer. 2005.

[129] J. Jeske, B. Brehmer, F. Menge et al.. *Aspektorientierte Programmierung Überblick über Techniken und Werkzeuge*. 3-939469-23-8. Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam. 2006.

[130] R. Johnson. *Expert One-on-One J2EE Design and Development*. 978-0764543852. Wrox. 2002.

[131] E. Juergens, F. Deissenboeck, B. Hummel. *CloneDetective - A Workbench for Clone Detection Research*. Proceedings of the 31st International Conference on Software Engineering . ACM. 2009. 603-606.

[132] S. H. Kaisler. *Software paradigms*. 978-0471483472. John Wiley and Sons. 2005.

[133] C. Kaner and W. P. Bond. *Software Engineering Metrics: What Do They Measure and How Do We Know?*. 10th International Software Metrics Symposium. http://www.kaner.com/pdfs/metrics2004.pdf. 2004.

[134] A. S. Kaplan and G. L. Murphy. *Category Learning With Minimal Prior Knowledge*. Journal of Experimental Psychology: Learning, Memory, and Cognition, Volume 26, Issue 4. 2000. 829-846.

[135] K. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. Otto von Guericke University, Magdeburg, Germany. 2010.

[136] D. Kehn. *Extend Eclipse's Java Development Tools*. http://www.ibm.com/developerworks/opensource/library/os-ecjdt/. IBM. 2003.

[137] H.-G. Kemper, W. Mehanna, C. Unger. *Business Intelligence- Grundlagen und praktische Anwendungen: : Eine Einführung in die IT-basierte Managementunterstützung*. 978-3834802750. Vieweg+Teubner Verlag. 2004.

[138] M. Kempka. *Plug-in-Design für Eclipse*. http://it-republik.de/jaxenter/artikel/Plug-in-Design-fuer-Eclipse-1150.html. JAXenter. 2007.

[139] M. Kersten. *Focusing knowledge work with task context. PhD thesis*. http://www.cs.ubc.ca/grads/resources/thesis/Nov07/Kersten_Mik.pdf. Department of Computer Science University of British Columbia, Canada. 2007.

[140] G. Kiczales, E. Hilsdale, J. Hugunin et al.. *An Overview of AspectJ*. ECOOP '01 Proceedings of the 15th European Conference on Object-Oriented Programming. Springer. 2001. 327-353.

[141] G. Kiczales and M. Mezini. *Separation of Concerns with Procedures, Annotations, Advice and Pointcuts*. ECOOP'05 Proceedings of the 19th European conference on Object-Oriented Programming. Springer. 2005. 195-213.

[142] G. Kiczales, J. Lamping, A. Mendhekar et al.. *Aspect-oriented programming*. ECOOP'97 Proceedings of the 11th European conference on Object-Oriented Programming. Springer. 1997. 220-242.

[143] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. 978-0471200246. John Wiley and Sons. 2002.

[144] D. Klein and G. Murphy. *Paper has been my ruin: conceptual relations of polysemous senses*. Journal of Memory and Language, Volume 47, Issue 4. 2002. 548-570.

[145] A. J. Ko, B. Myers, M. J. Coblenz et al.. *An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks*. IEEE Transactions on Software Engineering, Volume 32. IEEE. 2006. 971-987.

[146] V. Köppen, G. Saake, K.-U. Sattler. *Data Warehouse Technologien: Technische Grundlagen*. 978-3826691614. mitp Professional. 2012.

[147] S. Ladd, D. Davison, S. Devijver et al.. *Expert Spring MVC and Web Flow*. 978-1590595848. Apress. 2006.

[148] B. Lahres and G. Rayman. *Praxisbuch Objektorientierung. Von den Grundlagen zur Umsetzung*. 978-3898426244. Galileo Press. 2006.

[149] B. Lahres and G. Rayman. *Objektorientierte Programmierung*. 978-3836214018. Galileo Computing. 2009.

[150] G. Lakoff. *Women, fire, and dangerous things: What categories reveal about the mind*. 978-0226468044. University of Chicago Press. 1987.

[151] B. Landau. *Will the real grandmother please stand up? - The psychological reality of dual meaning representations*. Journal of Psycholinguistic Research, Volume 11. Kluwer. 1982. 47-62.

[152] L. Langit, K. S. Goff, D. Mauri et al.. *Smart Business Intelligence Solutions with Microsoft SQL Server® 2008*. 978-0735625808. Microsoft Press. 2009.

[153] S. Larndorfer, R. Ramler, C. Buchwiser. *Dashboards, Cockpits und Projektleitstände: Herausforderung "Messsysteme für die Softwareentwicklung"*. http://sigs-datacom.de/sd/publications/pub_article_show.htm?&AID=2596&Table=sd_article. OBJEKTspektrum. SIGS-DATACOM GmbH. 2009.

[154] S. Larndorfer and R. Ramler. *TestCockpit: Business Intelligence for Test Management*. Proceedings of the Work in Progress Session in connection with 33rd EUROMICRO Conference on Software Engineering and Advanced Applications. 2007.

[155] S. Larndorfer, R. Ramler, C. Buchwiser. *Experiences and Results from Establishing a Software Cockpit at BMD Systemhaus*. Software Engineering and Advanced Applications, Euromicro Conference, 35th Euromicro Conference on Software Engineering and Advanced Applications. IEEE. 2009. 188-194.

[156] W. Lehner. *Datenbanktechnologie für Data-Warehouse-Systeme.*. 978-3898641777. Dpunkt Verlag. 2003.

[157] O. Lemos, S. Bajracharya, J. Ossher et al.. *CodeGenie: Using Test-Cases to Search and Reuse Source Code*. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM. 2007. 525-526 .

[158] O. A. L. Lemos, S. Bajracharya, J. Ossher. *CodeGenie:: a tool for test-driven source code search*. Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. ACM. 2007. 917-918.

[159] C. Lengauer, C. Consel, M. Odersky et al.. *Domain-specific program generation: international seminar, Dagstuhl Castle, Germany, March 23-28, 2003: revised papers*. 978-3540221197. Springer. 2004.

[160] S. Letovsky. *Cognitive Processes in Program Comprehension*. Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers. Ablex Publishing Corp. 1986. 58-79.

[161] S. Letovsky and E. Soloway. *Delocalized plans and program comprehension*. IEEE Software, Volume 3 Issue 3. IEEE. 1986. 41–49.

[162] B.P. Lientz and E.B. Swanson. *Software Maintenance Management*. 978-0201042054. Addison-Wesley. 1980.

[163] D.C. Littman, J. Pinto, S. Letovsky et al.. *Mental Models and Software Maintenance*. In Empirical Studies of Programmers: First Workshop. Ablex Publishing Corp.. 1986. 80-98.

[164] G. Mak. *Spring Recipes: A Problem-Solution Approach*. 978-1590599792. Apress. 2008.

[165] R. L. Manning. *AspectJ in Action*. 1-930110-93-6. Manning Publications. 2003.

[166] L. Markle. *JQuery - A tool for combining query results and a framework for building code perspectives. Masterthesis*. http://www.cs.ubc.ca/grads/resources/thesis/Nov08/Markle_Lloyd.pdf. The University of British Columbia, Vancouver Canada. 2008.

[167] L. Markle and K. De Volder. *JQueryScapes: Customizable Java code perspectives*. 7th International Conference on Aspect-Oriented Software Development - Demonstration. http://jquery.cs.ubc.ca/papers/aosd-jquery-demo-2008.pdf. 2008.

[168] A. Marowka. *A Study of the Usability of Multicore Threading Tools*. International Journal of Software Engineering and Its Applications Volume 4, Issue 3. 2010.

[169] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. 978-0135974445. Prentice Hall. 2002.

[170] C. Martin. *Stability*. http://www.objectmentor.com/resources/articles/stability.pdf. Objectmentor. 1997.

[171] C. Martin. *Granularity*. http://www.objectmentor.com/resources/articles/granularity.pdf. Objectmentor. 1996.

[172] C. Martin. *Agile Principles, Patterns, and Practices in C#*. 978-0131857254. Prentice Hall. 2006.

[173] C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 978-0132350884. Prentice Hall. 2008.

[174] D. McCandless. *Information is Beautiful*. 978-000729466. HarperCollins. 2010.

[175] M. McCloskey and S. Glucksberg. *Natural categories: Well-defined or fuzzy sets?*. Memory and Cognition, Volume 6. 1978. 462-472.

[176] D. L Medin and E. J. Heit. *Categorization*. In D. Rumelhart and B. Martin Handbook of cognition and perception. Academic Press. 1999.

[177] D.L. Medin, E.B. Lynch, K. O. Solomon. *Are there kinds of concepts?*. Annual Review Psychology, Volume 51. 2000. 121-147.

[178] M. C. Meier, W. Sinzig, P. Mertens. *Enterprise Management with SAP SEM/Business Analytics*. 978-3540002536. Springer. 2003.

[179] J. Memmert. *Designing with Cosmos*. Workshop Aspect Oriented Design, AOSD 2002. 2002.

[180] R. Merker. *Programmieren lernen mit Java*. 978-3834800688. Springer. 2006.

[181] C. B. Mervis and M. A. Crisafi. *Order of acquisition of subordinate-, basic-, and superodinate-level categories*. Child Development, Volume 53. 1982. 258-266.

[182] R. Miles. *AspectJ Cookbook* . 978-0596006549. O'Reilly. 2005.

[183] J. Münch, J. Heidrich, F. Simon et al.. *Soft-Pit: Ganzheitliche Projekt-Leitstände zur ingenieurmäßigen Software-Projektdurchführung*. Bundesministerium für Bildung und Forschung -BMBF-: Forschungsoffensive "Software Engineering 2006": Statuskonferenz 26.-28. Juni 2006, Leipzig. http://publica.fraunhofer.de/documents/N-48314.html. 2006.

[184] J. Münch and J. Heidrich. *Software project control centers: concepts and approaches*. Journal of Systems and Software, Volume 70, Issues 1–2. 2003. 3–19.

[185] J. Mundy, W. Thornthwaite, R. Kimball. *The Microsoft Data Warehouse Toolkit: With SQL Server 2008 R2 and the Microsoft Business Intelligence Toolset*. 978-0470640388. John Wiley and Sons. 2011.

[186] G. L. Murphy. *The Big Book of Concepts*. 978-0262632997. MIT Press. 2002.

[187] G.L. Murphy. *Comprehending complex concepts*. Cognitive Science, Volume 12. . 1988. 529–562.

[188] G. L. Murphy. *Causes of taxonomic sorting by adults: A test of the thematic- to taxonomic shift*. Psychonomic Bulletin and Review, Volume 8. 2001. 834–83.

[189] G. C. Murphy, M. Kersten, L. Findlater. *How Are Java Software Developers Using the Eclipse IDE?*. IEEE Software, Volume 23 Issue 4. IEEE. 2006.

[190] F. Navrade. *Strategische Planung mit Data-warehouse-systemen*. 978-3834910349. Gabler Verlag. 2008.

[191] B. Neate, W. Irwin, N. I. Churcher. *CodeRank: A New Family of Software Metrics*. ASWEC: Australian Software Engineering Conference. IEEE. 2006. 369-378.

[192] S. P. Nguyen and G. L. Murphy. *An apple is more than a fruit: Crossclassification in children's concepts*. Child Development, Volume 74, Issue 6. 2003. 1783–1806.

[193] S. Nguyen. *Cross-classification and category representation in children's concepts*. Developmental Psychology, Volume 43. 2007. 719-731.

[194] A. Nguyen, T. T. Nguyen, G., Jr. Wilson et al.. *A graph-based approach to API usage adaptation*. Proceedings of the ACM international conference on Object oriented programming systems languages and applications. ACM. 2010. 302-321.

[195] B. Nora, G. Said, A. Fadila. *A Comparative Classification of Aspect Mining Approaches*. Journal of Computer Science, Volume 2, Issue 4. Science Publications. 2006. 322-325.

[196] M. Odersky, L. Spoon, B. Venners. *Programming in Scala*. 978-0981531601. mitp Professional. 2008.

[197] H. Ossher, W. Harrison, F. Budinsky et al.. *Subject-Oriented Programming: Supporting Decentralized Development of Objects*. 1994.

[198] H. Ossher and P. Tarr. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer. 2000.

[199] K. Ostermann. *Reasoning about aspects with common sense*. Proceedings of the 7th international conference on Aspect-oriented software development. ACM. 2008. 48–59.

[200] M. P. O'Brien. *Software Comprehension – A Review and Research Direction*. Department of Computer Science and Information Systems University of Limerick, Ireland, Technical Report. 2003.

[201] O. Panchenko, J. Karstens, H. Plattner et al.. *Precise and Scalable Querying of Syntactical Source Code Patterns Using Sample Code Snippets and a Databasee*. 19th International Conference on Program Comprehension. IEEE. 2011. 41-50.

[202] O. Panchenko. *In-Memory Database Support for Source Code Search and Analytics*. Proceedings of the 2011 18th Working Conference on Reverse Engineering. IEEE. 2011. 421-424.

[203] O. Panchenko and H. Plattner. *Source Code Analytics*. Proceedings of the 15th IASTED International Conference on Software Engineering and Applications . 2011.

[204] C. R. Pandian. *Software metrics: a guide to planning, analysis and application*. 978-0849316616. Auerbach Publications. 2003.

[205] D. L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules*. Communications of the ACM Volume 15 Issue 12 . ACM. 1972. 1053-1058.

[206] T. B. Pedersen and C. S. Jensen. *Multidimensional Database Technology*. IEEE Computer, Volume 34 Issue 12. IEEE. 2001. 40-46.

[207] T. Pender. *UML Bible*. 978-0764526046. John Wiley and Sons. 2003.

[208] N. Pennington. *Comprehension Strategies in Programming*. In Empirical Studies of Programmers: Second Workshop. Ablex Publishing Corp.. 1987. 100-113 .

[209] N. Pennington. *Stimulus structures and mental representations in expert comprehension of computer programs*. Cognitive Psychology, Volume 19. 1987. 295–341.

[210] G. Pomberger and W. Pree. *Software Engineering: Architektur-design und Prozessorientierung*. 978-3446224292. Hanser Verlag. 2004.

[211] W. Pree. *Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design*. Proceedings of the 8th European Conference on Object-Oriented Programming. 3-540-58202-9. Springer. 1994.

[212] B. Pugh and A. Loskutov. *Findbugs homepage*. http://findbugs.sourceforge.net/. Sourceforge.net.

[213] V. Rajlich and N. Wilde. *The role of concepts in program comprehension*. Proceedings of the 10th International Workshop on Program Comprehension. IEEE. 2002. 271.

[214] R. Ramler, W. Beer, C. Klammer et al.. *Concept, Implementation and Evaluation of a Web-Based Software Cockpit*. Software Engineering and Advanced Applications, 36th EUROMICRO Conference. IEEE. 2010. 385 - 392.

[215] D. Ratiu and F. Deissenboeck. *How Programs Represent Reality (and how they don't)*. Proceedings of the 13th Working Conference on Reverse Engineering. IEEE. 2006. 83 - 92 .

[216] P. Rechenberg. *Informatik-Handbuch*. 978-3446401853. Hanser Verlag. 2006.

[217] T. Risberg, R. Evans, P. Tung et al.. *Developing a Spring Framework MVC application step-by-step 2.5*. http://static.springframework.org/docs/Spring-MVC-step-by-step/. Springsource.

[218] M.P. Robillard, W. Coelho, G. C. Murphy. *How Effective Developers Investigate Source Code: An Exploratory Study*. IEEE Transactions on Software Engineering, Volume 30 Issue 12. IEEE. 2004. 889-903.

[219] M.P. Robillard and F. Weigand-Warr. *ConcernMapper: simple view-based separation of scattered concerns*. Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange. ACM. 2005.

[220] M. P. Robillard and G. C. Murphy. *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies*. In Proceedings of the 24th International Conference on Software Engineering. ACM. 2002. 406-416.

[221] E. Rosch and C. B. Mervis. *Family resemblances: Studies in the internal structure of categories*. Cognitive Psychology, Volume 7. 1975. 573-605.

[222] B. H. Ross and G. L. Murphy. *Food for Thought: Cross-Classification and Category Organization in a Complex Real-World Domain*. Cognitive Psychology, Volume 38. 1999. 495–553.

[223] C. K. Roy, J. R. Cordy, R. Koschke. *Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach*. Science of Computer Programming, Volume 74 Issue 7. Elsevier. 2009. 470-495.

[224] M. Russo and A. Ferrari. *The Many-to-Many Revolution 2.0*. http://www.sqlbi.com/articles/many2many/. http://www.sqlbi.com/wp-content/uploads/The_Many-to-Many_Revolution_2.0.pdf. sqlbi. 2011.

[225] N. Rutar, C. B. Almazan, J. S. Foster. *A Comparison of Bug Finding Tools for Java*. Proceedings of the 15th International Symposium on Software Reliability Engineering. IEEE. 2004. 245-256.

[226] S. Sah and V. G. Vaidya. *A Review of Parallelization Tools and Introduction to Easypar*. International Journal of Computer Applications (0975 – 8887) , Volume 56. No.12. 2012.

[227] T. Savage, M. Revelle, D. Poshyvanyk. *FLAT^3: Feature Location and Textual Tracing Tool*. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. ACM. 2010. 255-258.

[228] H.-A Schmid. *Systematic Framework Design by Generalization*. Communications of the ACM, Volume 40 Issue 10. ACM. 1997. 48-51.

[229] U. Schneider and D. Werner. *Taschenbuch der Informatik*. 978-3446407541. Hanser Verlag. 2007.

[230] H. Schrödl. *Business Intelligence mit Microsoft SQL Server 2005. BI-Projekte erfolgreich umsetzen*. 978-3446404632. Hanser Verlag. 2006.

[231] H. Seeberger. *Erste Schritte mit OSGi*. http://it-republik.de/jaxenter/artikel/Erste-Schritte-mit-OSGi-2077.html. JAXenter. 2008.

[232] H. Seeberger. *Erste Schritte mit OSGi - Teil II*. http://it-republik.de/jaxenter/artikel/Erste-Schritte-mit-OSGi---Teil-II-2078.html. JAXenter. 2008.

[233] T.M. Shaft and I. Vessey. *The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension*. Information Systems Research, Volume 6 no. 3. 1995. 286-299.

[234] B. Shneiderman and R. Mayer. *Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results*. International Journal of Parallel Programming, Volume 8 Issue 3. 1979. 219 - 238.

[235] J. Sliwerski, T. Zimmermann, A. Zeller. *Don't Program on Fridays! How to Locate Fix-Inducing Changes*. Proceedings of the 7th Workshop Software Reengineering. 2005. .

[236] S. A. Sloman. *Categorical Inference Is Not a Tree: The Myth of Inheritance Hierarchies*. Cognitive Psychology, Volume 35. 1998. 1–33.

[237] E. E. Smith and D.L. Medin. *Categories and concepts*. 978-0674102750. Harvard University Press. 1981.

[238] E. E. Smith, D. N. Osherson, L. J. Rips et al.. *Combining prototypes: A selective modification model*. Cognitive Science, Volume 12. 1988. 485–527.

[239] J. D. Smith, M. J. Murray, J. P. Minda. *Straight talk about linear separability*. Journal of Experimental Psychology: Learning, Memory and Cognition, Volume 23. 1997. 659-680.

[240] E. Soloway and K. Ehrlich. *Empirical Studies of Programming Knowledge*. IEEE Transactions on Software Engineering, Volume 10 Issue 5. IEEE. 1984. 595-609.

[241] I. Sommerville. *Software Engineering*. 978-0137035151. Addison Wesley. 2010.

[242] T. Stahl, M. Völter, S. Efftinge et al.. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 978-3898644488. Dpunkt Verlag. 2007.

[243] M. Stengel, M. Frisch, S. Apel et al.. *View infinity: a zoomable interface for feature-oriented software development*. 33rd International Conference on Software Engineering (ICSE). ACM. 2011. 1031-1033.

[244] M.-A. Storey. *Theories, Methods and Tools in Program Comprehension: Past, Present, and Future*. Proceedings of the 13th International Workshop on Program Comprehension. IEEE. 2005. 181-191.

[245] S. M. Sutton, Jr. and I. Rouvellou. *Modeling of software concerns in Cosmos*. Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM. 2002. 127-133.

[246] S. M. Sutton, Jr. and P. Tarr. *Aspect-oriented design needs concern modeling*. Workshop on Aspect-Oriented Design 1st International Conference on Aspect-Oriented Software Development. 2002.

[247] S. M. Sutton, Jr.. *Early stage concern modeling*. Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 1st International Conference on Aspect- Oriented Software Development. 2002.

[248] S. M. Sutton, Jr. and I. Rouvellou. *Concern Space Modeling in Cosmos*. http:// www.research.ibm.com/AEM/pubs/cosmos_oopsla2001_poster_abstr.pdf. 2001.

[249] C. Szyperski, D. Gruntz, S. Murer. *Component Software*. 978-0201745726. Pearson Education. 2002.

[250] P. Tarr, W. Ossher, W. Harrison et al.. *N degrees of separation: multi-dimensional separation of concerns*. Proceedings of the 21st international conference on Software engineering (ICSE). ACM. 1999. 107-119 .

[251] A. Teyseyre and M. Campo. *An Overview of 3D Software Visualization*. IEEE Transactions on Visualization and Computer Graphics, Volume 15, Issue 1. IEEE. 2009. 87-105 .

[252] M. Thiele, W. Lehner, D. Habich. *Data-Warehousing 3.0 - Die Rolle von Data-Warehouse-Systemen auf Basis von In-Memory Technologie*. Innnovative Unternehmensanwendungen mit In-Memory Data Management, IMDM. Springer. 2011. 57-68.

[253] E. Thomsen. *OLAP Solutions: Building Multidimensional Information Systems*. Wiley. 1997.

[254] T. Thüm, C. Kästner, F. Benduhn et al.. *FeatureIDE: An Extensible Framework for Feature-Oriented Software Development*. Science of Computer Programming. 2012.

[255] M. F. Verda, G. L. Murphy, B.H. Ross. *Influence of multiple categories on the prediction of unknown properties*. Memory and Cognition, Volume 33, Issue 3. 2005. 479-487.

[256] M. Völter. *Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development*. http://www.voelter.de/data/pub/ModelsAndAspects.pdf. 2005.

[257] A. von Mayrhauser and A.M. Vans. *Program Comprehension During Software Maintenance and Evolution*. Computer, Volume 28 Issue 8. IEEE. 1995. 44-55.

[258] D. Weeratunge, X. Zhang, S. Jagannathan. *Analyzing multicore dumps to facilitate concurrency bug reproduction*. Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS). ACM. 2010. 155-166.

[259] J. Weilbach and M. Herger. *SAP XApps and the Composite Application Framework*. 978-1592290482. SAP Press. 2005.

[260] B. Westfechtel. *Models and tools for managing development processes*. 978-3540667568. Springer. 1999.

[261] M. Würsch, G. Ghezzi, M. Hert et al.. *SEON: a pyramid of ontologies for software evolution and its applications*. Computing, Volume 94, Issue 11. 2012. 857-885..

[262] G. Wütherich, N. Hartmann, B. Kolb et al.. *Die OSGI Service Platform-Eine Einführung mit Eclipse Equinox*. 978-3898644570. dpunkt Verlag. 2008.

[263] T. Yamashina, H. Uwano, K. Fushida et al.. *A real-time code clone detection tool for software maintenance.*. Technical Report NAIST-IS-TR2007011. Graduate School of Information Science, Nara Institute of Science and Technology. 2008.

[264] T. Yamauchi and A. B. Markman. *Inference Using Categories*. Journal of Experimental Psychology: Learning, Memory, and Cognition, Volume 26, No. 3. 2000. 776-795.

[265] E. Yourdon, Y. Press, L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design: Fundamentals of a Discipline of Computer Programme and Systems Design*. 978-0138544713. Wiley. 1979.

[266] A. Zeller. *The Future of Programming Environments: Integration, Synergy, and Assistance*. Proceedings of Future of Software Engineering (FOSE '07) . IEEE. 2007. 316-325.

[267] A. Zeller. *Yesterday, my program worked. Today, it does not. Why?*. Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE). Springer. 1999. 253-267.

[268] A. Zeller and R. Hildebrandt. *Simplifying and Isolating Failure-Inducing Input*. IEEE Transactions on Software Engineering , Volume 28 Issue 2. IEEE. 2002. 183-200.

[269] A. Zeller. *Automated Debugging: Are We Close?*. IEEE Computer, Volume 34 Issue 11. IEEE. 2001. 26-31.