Hypermodelling

Introducing Multi-dimensional Concern Reverse Engineering

Tim Frey
Otto-von-Guericke-University
Magdeburg, Germany
tim.frey@tim-frey.com

Veit Köppen
Otto-von-Guericke-University
Magdeburg, Germany
veit.koeppen@ovgu.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
gunter.saake@ovgu.de

ABSTRACT

In data warehouse systems you search for all items that have a desired combination of features, called dimensions. Similar to features of an item, concerns of software are scattered throughout a software system. So, a developer might ask the following questions: Which modules are belonging to a concern? Which concerns are appearing in a package? In this paper, we introduce multi-dimensional concern reverse engineering supported by Data Warehouse technology. This approach enables to search for code fragments like it can be done for artifacts in the warehouse. We show that a transformation process from source code to the Data Warehouse is possible. Consequently, a developer can fasten up searches and perform source code analysis more easily.

Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

Keywords

Separation of Concerns; Data Warehouse; Hypermodelling

1. INTRODUCTION

One of the most important principles in software engineering is separation of concerns (SOC) [7]. This principle addresses the goal that modules have a primary and only responsibility. The term concern can be defined as a logical matter of interest in a software system [27]. Often programmers use or develop libraries, encoding functionality of hardware devices. Such functionality is a typical instance of a concern. Developers are confronted with the task to encapsulate functionality of hardware devices into modules or to compose new functionality through the use of libraries. Hence, we focus on the generic definition of concern as any matter of interest and see hardware functionality as a sample instance of this term.

In currently applied programming paradigms, it is not possible to separate all concerns into their own modules. Therefore, a module normally encodes multiple functionalities at the same time [21]. Different approaches enhance the capabilities of programming languages and enable a better separation [3,17]. However, a program is a complex system and concerns represent a multi-dimensional space [4]. Therefore, it is hard to understand how concerns are intertwined and related with each other; and still, a lot of modules weave the functionality of various concerns together. Consequently, a developer faces the challenge that modules represent multiple aspects of functionality at the same time. It is also occurring that functionality is encoded in multiple modules at the same time. The association of a source code fragment with multiple concerns creates an access and navigation problem in software analysis; concerns cannot be used for navigation. Neither can software be viewed or visualized from the concern perspective.

Concern tools allow developers to associate concerns with code elements [25]. Other tools enable query operations on code [6,14,28]. It seems, there is a gap between query and concern detection tools. Hence, no queries for fragments belonging to one or another concern at the same time can be done. Such queries can be used to uncover modules that are belonging to different concerns. Additionally, query tools slow down with queries, addressing relations of program elements to each other. This results from the used primary hierarchy; normally a program is viewed from packages down to classes and their members. Query tools operate on top of this structure. This slows queries down, because element references are not directly connected.[1]

We see the necessity to enable analysis of source code for internal hierarchies and relationships within the code and not only for the primary hierarchy. A multi-dimensional viewpoint towards software is needed. Thus, we need a technology that supports multi-dimensional viewpoints of program. Consequently, this kind of technology would enable to investigate concern relations in in new ways. Furthermore, the former named limitation of the query time can be speed up. A multi-dimensional approach allows it to directly query for different relations besides the primary hierarchy.

We introduce the Hypermodelling approach to overcome addressed limitations. Our approach enables multi-dimensional concern reverse engineering. Hypermodelling is basically the idea to adopt mechanisms from the area of Data Warehousing (DW) in combination with SOC to create a multi-dimensional viewpoint towards software. We see Online Analytical Processing (OLAP) [16] as an efficient method to analyze multi-dimensional data.

More and more hardware devices are programmed in high level languages. In this paper, we focus on generic ways how concerns are encoded in the Java language. Previous attempts have shown that it is not a trivial task to build a holistic multi-dimensional schema for Java [11]. Thus, we see a first step to determine the application ability to use DW technology in the creation of a DW cube. We test the application of our cube with a sample application that we load into the cube. Since loading code into this cube is a high effort, we identify several areas that profit from it. This is done with respect to costs and benefits.

Our contribution is to describe the benefit of DW technology for code analysis. Furthermore, the Hypermodelling approach to use

[1]For instance, in Eclipse references between classes are resolved by scanning the classpath and they are not connected statically.

a DW for code analysis is explained. We present a first cube and an exemplary report on it. This shows, DW technology makes it possible to query code from various viewpoints.

The paper is organized as follows: First, we describe different areas that could profit from the Hypermodelling approach. Afterwards, we briefly explain related research fields. Then, similarities between the different fields are shown and the Hypermodelling approach is presented. Following, we present the actual application to load source code into a DW. Reports demonstrate how measures are computed. Related research is compared and shown. Finally, we conclude and briefly describe future work paths.

2. MOTIVATION

Loading code into a DW comes with cost that should be paid only if convincing benefits can be obtained. Hence, we argue to apply DW technology to analyze source code in the following points.

2.1 Aggregation

Programmers spend most of their time exploring source code [24]. Thus, multiple attempts exist to improve the usage of Integrated Development Environment (IDE). This is done through support for concern analysis or enhancements for query operations [6,18,23,25,28]. A problem of the query tools is that when large code bases are queried; the performance drops down.[2] We develop Hypermodelling for Eclipse to allow developers to state multi-dimensional queries in the IDE for investigation of projects. The tool computes code slices based on element relations in the code. For instance, members of a slice are all classes that extend a specific class. The current implementation consists mainly of a query engine, responsible for parsing code and computing results. Thus, also the performance drops comparable to other tools. The queries have a multi-dimensional character through the inspiration from OLAP tools. One main application in DW is the usage of aggregates in queries. These allow us a fast computation and aggregated measures enable logical relations to be visualized through measures and indicators. For instance, the total number of employees in a certain district can be computed this way. Both dimensions, employee and district, can be aggregated against each other. Hence, it would be desirable to speed up the queries through the usage of DW technology to add the possibility of aggregate inclusions in queries. Such a query could be: all classes of a project that are annotated by a specific annotation and that are extending a certain class. Additionally, aggregates for various dimensional combinations can be (pre-) computed. The advantage of Hypermodelling in comparison to other tools would not only be the superior multi-dimensional query possibility, but also a performance advantage.

2.2 Various viewpoints

Additionally, various tools within an IDE are applied to investigate code from various perspectives. Future IDEs will use data from code and associated processes, leverage this data, and support synergy in data and functionality. The data of various tools will be used within other tools that identify a new meaning and therefore create a benefit. Furthermore, a great challenge is to use the immense amount of data and turn it into useful knowledge [29]. Process data are different kinds of facts, associated with code. They tie code elements or their combination with other data together. Such facts can be; the association of code elements belonging to a certain task, or test outcomes and associated tested code fragments. Hence, an integrating technology, scaling for large amounts of data, is needed. The desired representation of various facts and their association with code must be possible in an extensible way and allow (re)composition to enable new usage scenarios.

2.3 Hierarchies

In commonly applied languages the object dimension is considered as the dominant mean to separate concerns. Further viewpoints have been proposed to advance software development [4,27]. Even though, the exploration of code is normally done by a primary hierarchy. This hierarchy is from project down to the package structure containing classes and their members. Other hierarchies like call or method hierarchies are supported, but the very main hierarchy is still used most [20]. New mechanisms to separate concerns are supported by IDE extensions, offering new viewpoints on the code [5]. Thus, a technology supporting various viewpoints towards code capable of representing multiple hierarchies is needed for advanced analysis.

2.4 Dynamic artifact extraction

Related to code visualizations is the representation of software through models. Models can be seen as various projections of a program and the related information that is used within the development process. Thus, software is a multi-dimensional space, visualized via various model planes that are used for projections. Therefore, we formulate the goal: create a holistic model that can be used to create models dynamically [1].The dynamic creation of models out of a holistic model probably happens with transformations out of the central model. If a model represents a case like a class diagram it is just a subset of the code that is visualized. In the case of sequence diagram it is just data that is associated with various source code elements. Thus, a technology is needed to support the extraction of various facts out of the source model easily. Hence, again the need for a technology, allowing regarding software from various viewpoints excels.

2.5 Integration

Mining software archives deals with automated extraction, collection and abstraction of data generated in the development process [12]. Thereby, data are extracted out of various systems and aligned in a format for analysis [9,26]. Thus, to alter or switch to another perspective for the analysis means to alter the custom built extraction mechanism. A tool extracting facts at an aggregation level of interest would help to concentrate on the primary task to analyze the data and not to extract the data. Data of the development process are normally associated with the source code of an investigated program. Thus, a model of source code is needed that can be used to integrate data from various sources at one central point. This can be used to extract easily desired data from a central point.

2.6 Indicator Associations

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 that indicators, as also the code structure, are needed to analyze the way of machine operates. 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 DW is

[2]For instance JQuery [28].

computations of indicators. Therefore, a first step towards analyzing code structure and indicators, is the realization of multi-dimensional model for software within a DW. This can then be used to associate indicators with source code elements.

3. HYPERMODELLING

The Hypermodelling idea, to combine OLAP and SOC, covers research areas that are normally not viewed together. We briefly introduce the concepts required for their combination and we explain Hypermodelling via an example.

3.1 Separation of Concerns

Programming language designers have developed numerous mechanisms for SOC at the source code level. In this paper we focus mainly on Metadata annotations [3,10], classes, methods and Aspects [8,17].

Annotations are elements that can be used as meta information for classes and their members like fields or methods. Normally, this information is used to enhance annotated elements with certain capabilities [3]. For example such annotations can influence how an object graph of an application is wired.

Aspects are program fragments that are used to enable programmers to code crosscutting concerns in their own modules [8,17]. Crosscutting concerns represent functionality of a program that is not clearly belonging to one module and would be scattered within various modules. Aspects enable grouping such kind of functionality together and define places where the functionality is applied.

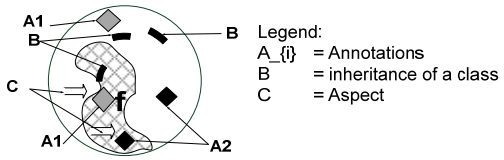

Figure 1. A Software System

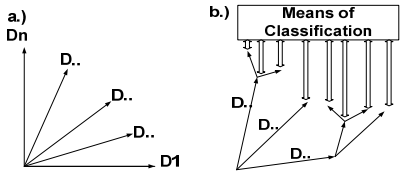

Figure 2. Fragment space

Figure 1 shows an abstract visualization of a programs source code. The circle symbolizes the code. The symbols represent occurring elements in the code. Figure 1 also shows a fragment of source code *f*. This fragment is affected by multiple concerns at the same time. It has different annotations, is affected by an aspect, and extends a certain class. All this concerns, affecting the element can be determined by parsing the code.

The generalized view of the software fragment can be seen in Figure 2a. A fragment belongs to different concerns at the same time; named as D1 to Dn. In Figure 2b it is shown that these concerns can have hierarchies, like inheritance or package structures. The large wide arrows and the arbor express that these concerns and the sorting of the corresponding source code fragments into hierarchies can be done automatically.

Examples for concern associations in Figure 2 are parent classes or interfaces. A fragment is associated with various other classes that represent D1 to Dn. The parent classes are located within their own hierarchies: They have parent classes themselves or they are located within a package structure what is expressed as hierarchy. Likewise, annotations can be used for a fragment. These are also located within packages. A hierarchy can be used to visualize this relation of a code fragment with annotations and the corresponding hierarchy. Generally, every mean to apply SOC can be used to determine the association of a fragment with concerns. We kept the graphic generic to be not limited to a specific kind of concern associations. All concern associations are known and can be resolved automatically out of the language structure. This way, the various associations of a code fragment with concerns can be done.

3.2 Data Warehousing

DW and OLAP are used to analyze multi-dimensional data. The main data structure for DW is the data cube [16], which we depict in Figure 3 as a relational schema. This structure consists as one possibility out of a relational schema on top of which an OLAP-Cube is designed [16].[3]

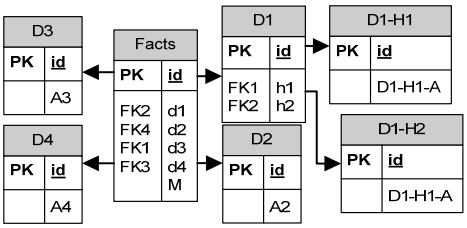

Figure 3. Snowflake schema for data Cube

We present a Snowflake schema of a DW in Figure 3. The fact table references multiple dimensional tables (D1-D4). A row in the fact table is associated with measures (M). Such measures are often called indicators. They are associated with combinations of dimensions. This means a row in the fact table is a connector of different points in dimensions and measures. Dimensions can have attributes (A1-A4) and also references to other tables in hierarchical levels (h1: D1-H1, h2: D1-H2).

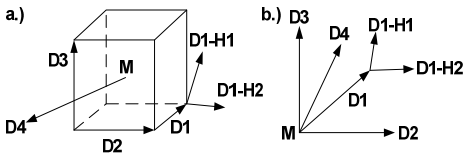

Figure 4. OLAP Cube

A data cube, on an abstract level, shown in Figure 4 visualizes a multi-dimensional structure. There, the association of the measure (M) with points in the different dimensions is visualized. A hierarchy of D1 is shown that can be used to structure members of a dimension in hierarchical levels. Finally, queries are executed in the data cube. Selections, filtering of dimensions as well as navigation between different dimensional hierarchies (called Roll-up, Drill-down, or Drill-across) are selected to determine corresponding measures. On main advantage of OLAP is that aggregations of measures are available and the navigation path is

[3]Since OLAP doesn't imply a relational schema, it is presented here anyways to ease comprehension, because the scope of this paper is to show similarities to software (re-)engineering.

easily and efficiently accessible. This enables us at the same time to generate reports on actual and aggregated data. Note; data mining techniques could also be applied within DW.

3.3 Hypermodelling

In Hypermodelling a source code fragment, for example a class or a method, can be associated with multiple concerns at the same time. All fragments that are associated with a specific concern are members of a slice belonging to this concern. Thus, concerns are quite similar to dimensions in the context of OLAP cubes. This leads to the technique to use OLAP similar methods to query for fragments, belonging to one or more concern. We use the dimensions of an OLAP cube as concerns. For us, the association of a class with concerns, like its parent classes, is the same like the associations of measures in OLAP with dimensions.

```
1: @Entity
2: @Deprecated
3: class Customer{
4:          @Deprecated
5:          Customer(){
6:          …}
7: …}
8: class CustomerDAO extends DaoSupport
9:
10:         @SuppressWarnings(“deprecation”)
11:         Customer createCustomer(){
12:                   return new Customer();
13:         }     ...}
```

Listing 1. Example for annotated source code

Table 1. Concerns of Listing 1 in a table

Element / Dimensions	extends DaoSupport	@Entity	@Deprecated	@SupressWarnings
Customer		X	X	
Customer.Customer()			X	
CustomerDAO	X			
CustomerDAO. createCustomer()				X

We present an example in Listing 1 and Table 1. Listing 1 shows Java source code. A Customer class and a Data-Access-Object (DAO) are shown. The CustomerDAO class extends a helper class (DaoSupport) for table access. This is commonly done when frameworks are used. Table 1 shows rows that are representing source code fragments and columns representing concerns. The “X” indicates that a fragment belongs to a concern. For example the constructor of the class Customer is marked @Deprecated. Likewise the rest of the table-listing associations can be done.

In a nutshell, the main idea is to use the code fragments itself as fact table and allow queries on this “table”. Furthermore, also hierarchies exist in source code. A class, for instance, is member within a hierarchical package structure. This way, source code shows huge similarities to DW data structures.

A sample query for Table 1 could be; all fragments that belong to @Entity and to @Deprecated. This leads to the result of the Customer class. Hence, queries can be used to determine code fragments, belonging to different concerns at the same time.

4. Data Warehouse-application evaluation

We load source code into a DW to evaluate the possibility of the application of actual OLAP technology. We use in our example the Microsoft Analysis Services Server Version 2008 R2. We lean the application partly on the previous shown sample and load annotations and inheritance. The loading is done via ETL (Extract-Transform and Load). For this we introduce an intermediate layer, the ETL-Relational source code schema. Afterwards, we explain the cube structure.[4] Furthermore, we visualize exemplary queries and show a DW report.

4.1 The relational schema for ETL

As data source for the cube, we develop a relational schema for annotations and inheritance. The schema is inspired by the Eclipse internal Java model[5]. A reason to lean the model towards the Eclipse model is to be open for a portability of the Hypermodelling technique into the Eclipse IDE.

Nevertheless, the model of Eclipse is actually not build as a model for a relational database. Therefore, modifications have to be applied to create a similar model to a relational database. Thus, our relational model can just be seen as inspired by Eclipse and not as a one to one mapping. We perceive that some relations in the Eclipse model are based on the Java language specification logic. These differ compared to logical viewpoints of a programmer. For example, instances of annotations are not linked with the definition of the annotation type itself. This means that the occurrence of an annotation is actually not an instance of its type definition. This kind of “logical” gaps also makes a challenge for the transformation 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, presented in Figure 5, shows the relational representation, whereby all fact tables are emphasized in grey. The fact tables are the source for the associations of the 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 for inheritance.

In Java primitive (e.g., integer and boolean) and complex (Classes, Enums, Annotations, Interfaces) types occur in source code. These are realized through AbstractType, ComplexType and TypeClassification table. The TypeClassification indicates the kind of type. The AbstractType is defined to have the possibility of 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.[6] However, a complex type can also have additional properties in contrast to a primitive. Generally, a complex type is defined in a file that is, again, belonging to a package which is furthermore a member in a taxonomic package

[4] The shown graph is leaned on the visualization in [13].

[5] http://www.eclipse.org/jdt

[6] AbstractType enables to extend the model. Further associations between model elements can be done; e.g., method parameters that can be primitive or complex types.

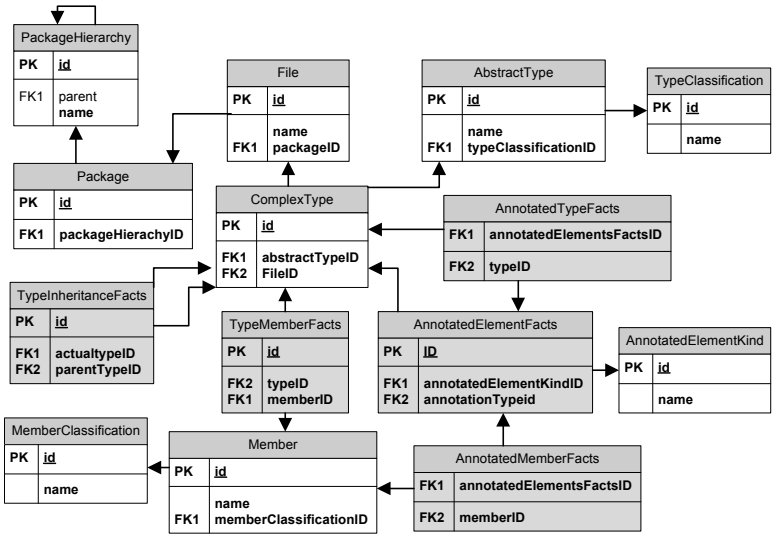

Figure 5. Relational schema

hierarchy. Often different package roots exist in a project that contains the same packages. Thus, a file is not a direct member in a package hierarchy and the bonding to the Package table is done it between.

As a matter of fact, a complex type can have multiple members. Such members are fields and methods. Like shown, this relation is realized though a fact table.[7] 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 the different member kinds and mind their different properties. Implementation of interfaces and inheritance of classes are realized through the TypeInheritanceFacts table. The approach is taken to avoid a self reference of the ComplexType table, because a type can implement multiple interfaces.

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

4.2 The cube structure

In Figure 6, we present a visualization of a cube on top of the relational schema. The dimensions are the nodes and the hierarchies are shown by their connection.

The first dimension type represents elements, having an association to a complex type (named Type in the figure). They are located at the button, beneath the facts (Program nodes). They are affected elements by inheritance or annotations. Thus, they are "passive" elements in a relation. Since Members and Types can be annotated, both of them are directly connected with a fact table. In this way, we enable the possibility to sum up Member Annotations at Type dimension or to compute with the ones appearing at a Type.

[7]The approach of a fact table in between was chosen to credit reality; often methods are moved between complex types.

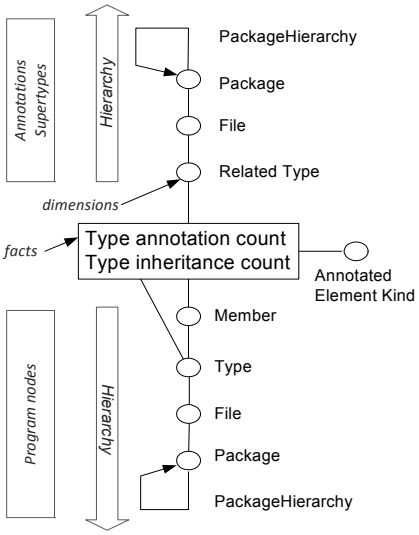

Figure 6. The Hypermodelling Cube

The second type of dimensions is the AnnotatedElementKind . We state this explicitly, because it is only used together with annotations and not with inheritance facts. The Member dimension is arranged parallel to the Type dimension. The reason is the hierarchic relation between members and types. Through that relation facts, associated with a member, can be aggregated at type level. Thus, the AnnotatedElementKind dimension is introduced to enable queries at type and also at member level. We visualize this vertical to the others to emphasize that it can be used to discriminate other dimensions.

The third type of dimension is the "related" dimension that is located at the top (Annotations, Supertypes). This represents "active" parts of a relation. Active means the annotation or the supertype itself.

Finally, two measures, consisting of occurrence counts, connect different dimension types. We show that the dimensions on top and bottom are ordered in a hierarchy. Therefore, queries can use hierarchical structures that occur in the code.

Table 2. The source data of the report in Figure 9

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			

				Repository	Transactional	SuppressWarnings	PersistenceContext
EntityManagerClinic	Member	Field	em				1
		Method	getVets		1	1	
	Type			1	1		

Figure 7. Association measures at types and members

	org.springframework	java.lang
	org.springframework.samples.petclinic	
	Clinic	Object
EntityManagerClinic	1	1

Figure 8. Inheritance measures sample

4.3 The filled cube

The discussed relational schema and cube are filled with real Java data. As data, to support verification, we choose a sample application that is available publicly. The elected petclinic application is a demonstration of the capabilities of the spring framework that is applied widely in the industry. 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 of 31 Java files, containing application logic. It is making use of declarative transaction management, database access, and aspect oriented programming paradigms. [9]

In order to load the data, a parser for the Eclipse IDE is developed that inserts directly Java source code into the relational schema. Out of the filled schema the cube is processed.

4.3.1 Sample source code in the cube

In the following, we present sample queries containing results about a source code excerpt.

```
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()  {
```

Listing 2. Excerpt source code of the demo project

In Listing 2, we show an excerpt of a class of the org.springframework.sample.petclinic.jpa package.Corresponding query results are presented in Figure 7 and 8. In Figure 8 we present the inheritance measure. As it can be seen, the insertion also added the Object class as ancestor, since every class inherits in Java from this class. Additionally, the package hierarchy is shown to indicate that drilldowns over various hierarchy levels are possible. Generally, this enables queries in the style; all classes in a package, extending a class of another package (java.lang) and implementing an interface of another (org.springframework).

We show annotated elements in Figure 7. Likewise, the hierarchical structure of dimensions is shown on the left. The EntityMangerClinic has annotations at type and also at member level. Members can be divided into fields and methods. The count of one indicates that an annotation exists. Now, a query can determine the fragments that are fields and are marked as @PersistenceContext at the same time; the result is *em*. Clearly, dimensions like class name could be even used to discriminate further. Even a combination with the inheritance facts is possible. Divers concerns can be used to navigate through code.

In the following a report that uses the annotation measures is presented. We restrict ourselves to this example; however, further reports are possible.

4.3.2 A sample report

Table 2 shows the source data of our exemplary report in Figure 9. We present it to enable the verification of our report. We verify our loading technique of code by counting the occurrences of annotations in the sample loaded code that is publicly available. In Figure 9 a sample report visualizes the distribution of annotations that occur in the packages of the analyzed project. We use data of Table 2 as source. 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 where they are defined. For example, the previous shown Annotation Transactional is in the

[9] The petclinic application is described at http://static.springsource.org/docs/petclinic.html and can be downloaded at http://hypermodelling.com

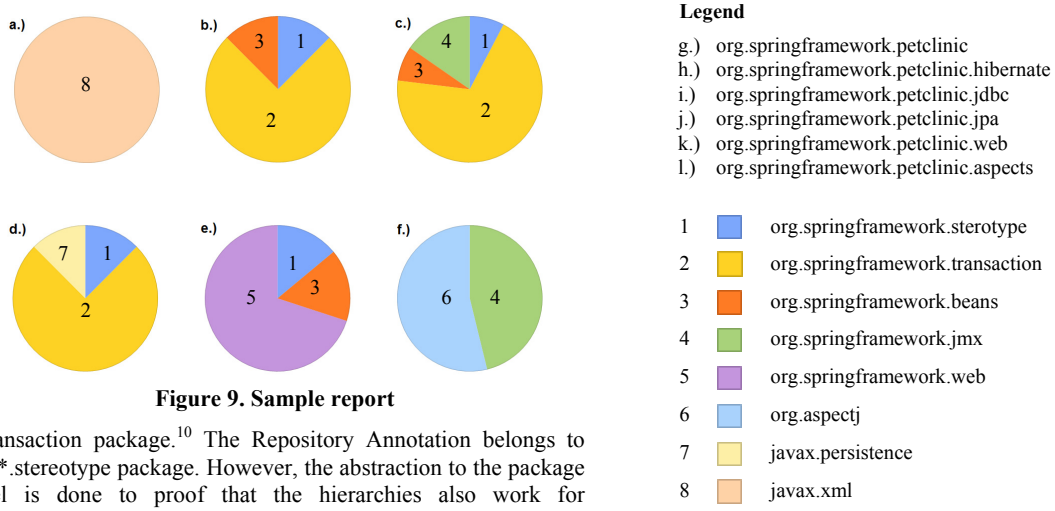

Figure 9. Sample report

*.transaction package.[10] The Repository Annotation belongs to the *.stereotype package. However, the abstraction to the package level is done to proof that the hierarchies also work for annotations and to present a report on a generalized level. Further, the annotated elements are generalized to the package level.

4.3.3 Interpretation of the report

The report shows, two different dimensions and their hierarchies can be related to each other. So it is possible to see code not in one primary perspective anymore and to create reports from various perspectives. Also, the indicators can be computed for different perspectives, like for classes or packages. It is now possible to see code also from the viewpoint of annotations and their packages.

We also use the very main application of DW technology, aggregations. Aggregations show the total occurrences with the two dimensional hierarchies and enable drill downs over various dimensions.

The visualization shows that the occurrence of dimensional relations can be depicted, easily. This makes it possible to visualize various relations just with a few clicks. This is a huge advantage in comparison to the current state of the art in software visualization. There, the data to create such reports would have been collected through custom parsers and then a custom visualization would have been applied. Now, it is possible to create reports on the fly.

An advanced application for such reports is also to browse the code from various perspectives and interpret the results. The report shown in Figure 9 can be used to interpret the responsibility of various packages in the project. It excels that package *a* has just xml annotations and no other ones. The reason is that the package represents the domain model of the application, as it can be verified by 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.* 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 persistency 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.

With such reports a developer can now do a query for classes and packages that contain annotations out of the persistence package *.persistence and also contain ones out of the transactional package. The result set is all classes in package d.. This is quite useful to get the required code fragments that are members of transactions and implemented with a specific persistence technology. Without Hypermodelling it would have been a longer and more complex search; through Hypermodelling concern oriented navigation is now possible.

All together, it seems straight forward to see what a package is used for within the application, just out of the annotations used in a package. Thus, there is the assumption that this is maybe the case in other applications as well. This needs to be investigated deeper and is out of the scope of this paper. However, this very first application, to load data into the cube shows the possibility to regard source code from one another viewpoint already created an interesting observation; annotations might be used to indicate the meaning of a package. Hence, browsing and measuring code from various perspectives seems very interesting for further investigation scenarios.

5. Related Work

The idea of Hypermodelling is, aside from the motivating, related technology, related to other approaches that try to integrate code and enable it for systematic inspection. Following we present related approaches and compare it to the idea to store code in a multi-dimensional model.

Storing source code in databases [14] 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 differs to the approach proposed in this paper. Hypermodelling uses DW technology to do the query and the relational model is just used to be the source for the cubes. Hence, Hypermodelling can improve the current approaches through the usage of aggregations at various levels.

It seems that in the area of repository mining [12] the idea of having a central access for the source code is gaining attention. The usage of Business Intelligence like means to support the

[10]The wildcard '*' is used to indicate parent packages. Their name was stripped for readability issues and the package names are unique by the last name in this sample.

mining process is already mentioned. Since Business Intelligence is often used as a term for all the associated technologies, like OLAP or DW this can be seen related. Software Intelligence (SI) [15] 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. Also, the idea of using actual DW technology is not respected.

A source code search infrastructure name Sourcerer [2] based on a relational model with four tables enables searches of relations in code. The relations 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 the schema is quite to understand. Also, not so many self joins of a table are needed to calculate the relations. But, since the main application of Hypermodelling is to use multi-dimensional cubes and support aggregations it is also superior this way. Through the multi-dimensional model no self joins are needed and the relations can be determined faster.

In [22] a method to control a software development project with metrics is described. There, a Metrics Warehouse is mentioned. Since the measures calculated in the paper represent somehow metrics of source code this seems first to be related. But the metrics described are economic project figures, not representing code metrics. Therefore, the approach is supporting the business user and not the developer, what is one goal of Hypermodelling.

6. Conclusions and future Work

We presented various areas where the appliance of DW technology for source code analysis would be an advantage. We solved the problem in reverse engineering that concerns can't be used for navigation, visualization or as perspective through the Hypermodelling approach. Furthermore, we actually demonstrated the possibility to use DW technology, showing a first multi-dimensional model and a report. Now, source code can be navigated and also reported easily from various positions. Also, aggregations can be done. Nearby, it excelled that the meaning of the responsibility of packages can be indicated by annotations. The comparison to related work showed: the approach to use a multi-dimensional model is probably superior in determining relations in code and faster than normal SQL queries. It would be interesting to use the approach as code search engine.

Like described in the Section 2, various areas can benefit from the approach. We see the first step in investigating the portability of the DW technology directly into the IDE. This way, various query tools, as described in Section 2, can be accelerated. The model needs to be extended for other relations in code and associated with other facts, like process data and indicators. Also, the extraction of data and the application of mining with the DW are possible. We see the next step to extract information, already loaded in a schema, and use it for mining.

Nevertheless, only a small part of structures occurring in source code is currently used to demonstrate the actual application of the technology. For instance, we use annotations that are widespread in new Java programs, but older programs do not use them. We need more cubes to enable advanced possibilities in code investigation. These cubes should respect other facts as inheritance and annotations in their multi-dimensional model. This will represent a larger part of the Java language and enable advanced queries. It will also be capable to inspect further applications. Furthermore, it can be investigated if queries can be used to create software models dynamically. Furthermore, a holistic model in the DW can be used by various tools in the IDE to sample their data for different viewpoints.

Currently, no historical versions of software are used within our model. Since historical data is a key application in classic DW, this can be an interesting field in the future how to load and organize historical software versions.

Moreover, we are confident that a DW can play an integrative role in software design. We believe it can be used to plan different concerns, like it is done in strategic enterprise management [19]. This comes out of the reason that the multi-dimensionality and responsibility of modules can be respected through our approach. The incursion is; concerns can be planned on an abstract level with measures. For instance, new functionality for a package is planned with concern combinations that are associated with a measure. Then, the concrete implementation is done in the classes of the package. Finally, a report compares the planned measure with the actual implementation level.

Another idea is to associate other indicators than counting with source code. An advanced trail can be energy analysis. Energy gain or loss can be associated as indicators with source code. Additionally, call hierarchies and execution probabilities can be also associated with elements in source code. Through the central analysis point, correlations of the various elements can maybe be determined. This can be used to expose energy loss spots in software, what would be especially interesting when software operates machines. It would be an advantage to know the exact regions of code that are most often executed and consume the most energy at same time. Exposing such spots can lead to build energy-optimized code.

Lastly, modern software systems contain millions lines of code. In the case we want to handle this amount of data a technology is needed that is designed to handle large scale data sets. However, all the code search engines share currently the same limitation: little structure information is used to explore the source code. Additionally, a dominant viewpoint towards the programs is always the perspective of project-packages-classes. The internal structures in the code itself are not respected. One advantage to use DW technology is that the multi-dimensionality of source code can be respected this way. We believe that the approach can be used to build better source code engines in the future that can be queried in a multi-dimensional way.

7. REFERENCES

[1] C. Atkinson, D. Stoll. Orthographic Modelling Environment. FASE'08/ETAPS'08 Proceedings of the Theory and practice of software,11th international conference on Fundamental approaches to software engineering. pages 93-96. Springer. 2008

[2] K. S. Bajracharya, J. Ossher, C.V. 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), Belgium, 2010

[3] J. A. Bloch. Metadata Facility for the Java Programming Language. 2004.

[4] 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).pages 1-5. ACM. 2005.

[5] A. Colyer, A. Clement, M. Kersten. Aspect-oriented programming with ajdt. In Proceedings of AAOS 2003:

Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003. Springer. 2003.
[6] B. de Alwis, G. C. Murphy, M. P. Robillard. A Comparative Study of Three Program Exploration Tools.pages 103-112 ACM. 2007
[7] E. W. Dijkstra. Selected Writings on Computing: A Personal Perspective. On the role of scientific thought. Springer. 1982
[8] R. E. Filman, T. Elrad, S. Clarke, M. Aksit Aspect-Oriented Software Development. Addison-Wesley Professional; 1st edition. 2004
[9] M. Fischer, M. Pinzger, H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. ICSM'03. pages 23-32. Merlin. 2003
[10] M. Fowler, Domain-Specific Languages, Addison-Wesley, Netherlands. 2010
[11] T. Frey. Vorschlag Hypermodelling: Data Warehousing für Quelltext. 23rd GI Workshop on Foundations of Databases. CEUR-WS. pages 55-60. Austria. 2011
[12] M. W. Godfrey, A. E. Hassan, J. D. Herbsleb et al.. Future of mining software archives: A roundtable. IEEE. 2009
[13] M. Golfarelli, D. Maio, S. Rizzi. The Dimensional Fact Model: A Conceptual Model For Data Warehouses. In International Journal of Cooperative Information Systems, 7, pages 215-247. 1998.
[14] E. Hajiyev, M. Verbaere, O. de Moor. Codequest: scalable source code queries with datalog. ECOOP'06: proceedings. pages 2-27 Springer. 2006
[15] A. E. Hassan, T. Xie.Software Intelligence: Future of Mining Software Engineering Data. In Proceedings of FSE/SDP Workshop on the Future of Software Engineering Research. Santa Fe. 2010
[16] W. H. Inmon. Building the Data Warehouse. 4th ed., J.Wiley & Sons, New York. USA. 2005.
[17] G. Kiczales, E. Hilsdale, et al..An Overview of AspectJ. European Conference on Object-Oriented Programming (ECOOP). pages 327-353.Springer. 2001
[18] I. Majid and M. P. Robillard. NaCIN - An Eclipse Plug-In for Program Navigation-based Concern Inference. In Proceedings of the Eclipse Technology Exchange at OOPSLA.ACM. 2005
[19] M. C. Meier, W. Sinzig, P. Mertens. Enterprise Management with SAP SEM/Business Analytics. 2nd Edition, Springer. Berlin. 2005
[20] G. C. Murphy, M. Kersten, L. Findlater. How Are Java Software Developers Using the Eclipse IDE?. IEEE. 2006
[21] H. Ossher, P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In Proceedings of the Symposium on Software Architectures and Component Technology. Kluwer. 2001.
[22] C. R. Pandian. Software metrics: a guide to planning, analysis and application. Auerbach Publications. 2003
[23] J. H. Pfeiffer, A. Sardos, J. R. Gurd. Complex code querying and navigation for aspectj. Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange. ACM. 2005
[24] M. P. Robillard, W. Coelho, G. C. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. pages 889-903, IEEE. USA. 2004
[25] M. P. Robillard, F. Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In Proceedings of the 2005 OOPSLA Workshop on Eclipse technology eXchange. ACM. 2005
[26] J. Sliwerski, T. Zimmermann, A. Zeller. Don't Program on Fridays! How to Locate Fix-Inducing Changes. In Proceedings of the 7th Workshop Software Reengineering, 2005
[27] S. M. Sutton Jr., P. Tarr. Aspect-oriented design needs concern modeling. AOSD workshop on Aspect-Oriented Design. 2002.
[28] K. D. Volder. Jquery: A generic code browser with a declarative configuration language. In P. V. Hentenryck, editor, PADL, volume 3819 of Lecture Notes in Computer Science, pages 88–102. Springer, 2006
[29] A. Zeller. The future of programming environments: Integration, synergy, and assistance. Future of Software Engineering, IEEE, USA, 2007