Exploring Software Variance with Hypermodelling An exemplary approach

Tim Frey, Veit Köppen

Institut für Technische und Betriebliche Informationssysteme
Otto-von-Guericke University
Magdeburg, Germany
tim.frey@tim-frey.com; veit.koeppen@iti.cs.uni-magdeburg.de

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. 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 paper, we demonstrate that Hypermodelling is suitable to explore software variance. We present reports based on real application data of one project example to reveal multiple facts about the software variance. We show visualizations at different granularity levels. This supports our theory that Hypermodelling can be used to explore software variance in an easy way.

1 Introduction

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 [Sc97]. Without loss of generality, we restrict ourselves to Java as programming language in our examples. 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 [BMM10]. 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 [BMM10, GHH09, ZZ11]. Source code mining is about extracting facts of source code and associated artifacts to analyze them via machine learning algorithms [BMM10]. 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 develop the Hypermodelling approach, see for instance [Fr11, FKS11, Fr12]. Hypermodelling utilizes Data Warehouse (DW) technologies to enable and infrastructure an analysis of source code with Online Analytical Processing (OLAP) queries [HKP11]. This enables fast and flexible investigations of source code through queries without having a complex fact-extract scenario.

In our former work on Hypermodelling we visualized rudimentary queries on annotations and inheritance [FKS11]. In this paper, 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 at different granularity levels. Hence, our contribution is to demonstrate that Hypermodelling can be used to do investigations on software variance. Our reports give first clues what types of queries can be composed with Hypermodelling. This supports our hypothesis that DW mechanisms can be beneficial for software investigations.

First, we give some background information about Hypermodelling and our used exemplarily used project. In the following, we present and discuss various reports. Then, we describe related approaches and point out their similarity to the usage of our Hypermodelling approach. Lastly, we draw conclusions and give an outlook to future work.

2 Background

In this section, we recap the most important facts of Hypermodelling. Then, we describe details about the project that served as base to generate reports with our Hypermodelling approach.

2.1 Hypermodelling

Hypermodelling is developed on the idea to program analysis and DW. A more detailed description is available in our former work [Fr11, FKS11, Fr12].[1] DW systems are an integrative component in business computing [In05]. They are used to assemble data of different sources together. The integrated data are arranged into multi-dimensional data structures, i.e. so called cubes, which serve as base for queries [HKP11]. The queries allow aggregating different relations and hierarchies that occur in the data. For instance, the revenues for a specific salesman in a specific area can be computed for a given time period. Thereby, this query aggregates the region, the salesman and the time in relation

[1]Http://hypermodelling.com

to revenue indicator. Likewise, hierarchies can be abstracted. For instance, the region of the salesman can be split into continents, countries and counties and the aggregates are associated with the distinct revenues for those. Likewise, this can be done for other hierarchies, like customer group, year, or department. Generally, the idea is that different aggregations enable detailed investigations.

With Hypermodelling we introduce the idea that programming elements, like annotations or classes, are similar to the data that is used within a DW. For instance, classes are defined within a package hierarchy. Annotations are associated with classes and their members. They are also defined in their own package. This is like the association of a salesman to a region, time period, and revenues. The hierarchies in code are similar to hierarchies of region or time. All together, we load source code into a DW and realize the associations of classes, their inheritance, packages, and annotations as a DW cube. This enables us to built queries on top of this cube and it creates the possibility to compute different aggregations for code. For this paper, we combined the cubes that were presented in our former work [FKS11] and built queries on it. The results of queries on this cube are the reports that we present in this paper.

2.2 The Alfresco Project

In order to have valid and real world application data, we downloaded a head revision of the Alfresco project[2] 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 DW. 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 to exemplarily present reports with the Hypermodelling approach. Note, other projects could be used similarly, e.g., [FKS11].

3 Hypermodelling Inheriting Variance

In the following, we present different reports based on the loaded project. 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.

[2] http://www.alfresco.com

3.1 Inheritance Overview

We present the amount of total inherited types and distinct inherited types in Table 1. Types that are 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 1: Which kind of type is used as a supertype

	Total inherited types	Distinct inherited types	Implementation ratio (Total/Distinct)
Superclasses	2,798	377	7.42
Superinterfaces	1,435	625	2.3
Total	4,233	1,002	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 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.[3] In total, most of the implementing types are classes, followed by interfaces, and enumerations. Likewise, this is the same for the distinct inherited types.

Table 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

The implementation ratio is very interesting in this table. We assume indicators are led by classes, but we were 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 1 and Table 2 that the implementation ratio of classes exceeds interfaces. This means: Classes seem to have a higher variance then interfaces.

3.2 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.

3.2.1 Source Data

Table 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

[3]We leaved this out to avoid a complex anything with anything table.

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 DW 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 3: Exemplary source data of inheritance at the package level

Package hierarchy	**Total inherited types**	**Distinct inherited types**	**Implementation ratio (Total/Distinct)**
org.alfresco	1,972	808	2.441
java.lang	1,404	11	127.64
org.springframework	214	40	5.35
junit.framework	184	2	92.0
org.antlr	87	6	14.5
Total	4,233	1,002	4.22
Without org.alfresco	2,261	194	11:65
Without org.alfresco and java.lang	857	183	4.68

3.2.2 Dependency on Total Packages

In Figure 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.

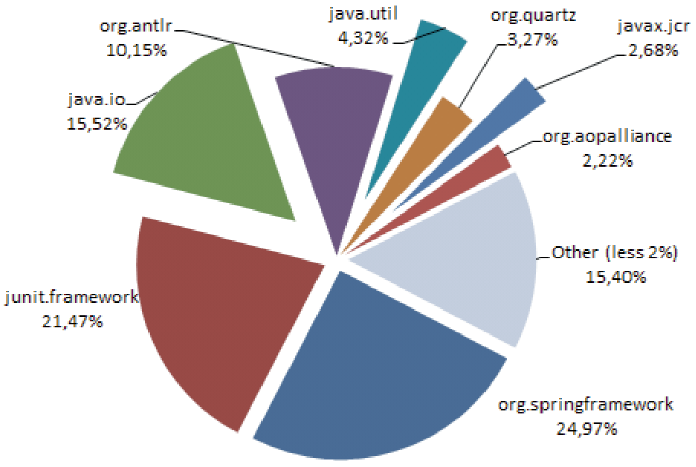

Figure 1: Dependency on total used types

3.2.3 Dependency on Distinct Packages

Figure 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.

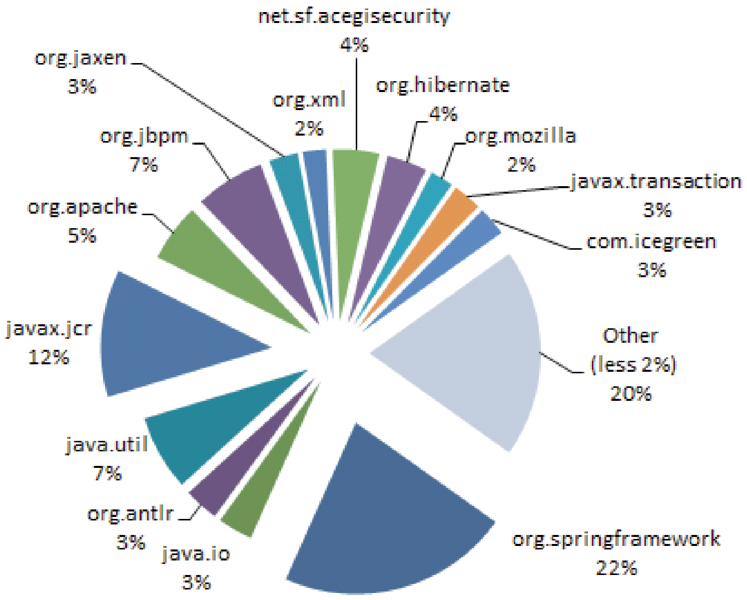

Figure 2: Dependency on distinct types

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 1. For instance, the Junit framework that is extensively used does not occur, because not many different types of it seem to be used.

3.2.4 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 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 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.[4]

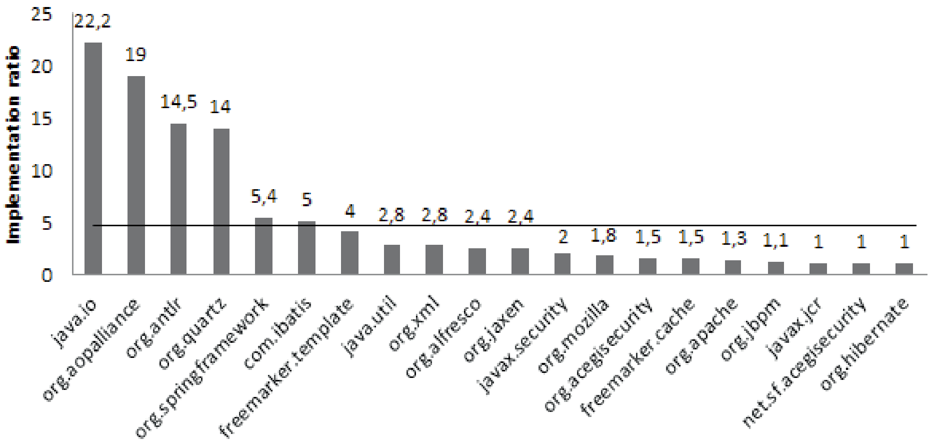

Figure 3: Implementation ratio for packages

However, like all other figures, the data of Figure 3 has been retrieved with a simple multi-dimensional expression (MDX-Queries).[5] The query language is declarative and especially created to query multi-dimensional data easily. In order to give an impression about the used query language, we present exemplarily the query that served as foundation to generate Figure 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’])
```

Listing 1: The query of Figure 3

[4]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.
[5]For further information about the MDX language, see the language reference: http://msdn.microsoft.com/en-us/library/ms145595.aspx and the XMLA standard: http://xmlforanalysis.com

In short, the query in Listing 1 computes dynamically the indicator *[Implementation Ratio]*. The quotient is computed by the aggregation of the *[Measures].[Type Inheritance Count])* as dividend and the distinct count of the amount of *[Measures].[Type Inheritance Count])* measures as divisor. The *SELECT* applies the computation to different packages of the corresponding cube (*FROM [Inheritance-Annotations-Cube]*). Lastly, all other packages, except the *org.alfresco* ones, are excluded as inheritors from the computation (*WHERE ([Package Hierarchy].[Parent].&[‘org.alfresco’])*).

3.3 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 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.[6]

Table 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	ApplicationContextAware	580	26
	InitializingBean	578	56
	AbstractLifecycleBean	351	42
	ApplicationListener	175	7
	… in total 40 types	… in total 2,374	… in total 214
junit.framework	TestCase	1326	170
	TestSuite	2	14
	Total 2 types	Total ,1328	Total: 184

3.3.1 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 5. The column distinct inherited types 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.

[6]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.

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.

Table 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
		setAuthenticationService	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

3.3.2 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[7]. 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."[8]

This means that the override annotation can be used, but there is no compulsion to do so. Thus, @Overwride marked methods override a method for sure, but overriding can also take place without this annotation. In fact, modern development environments, like Eclipse[9] remind programmers through messages and markers to use the annotation.

[7] http://jcp.org/ja/jsr/detail?id=175
[8] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Override.html
[9] http://eclipse.org

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.

3.3.3 The Report Filtered by @Override

We present the modified query result for the @Override marked methods of children of TestCase and InitializingBean in Table 6. 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 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[10]. TestCase[11] defines the two @Override marked methods. Hence, InitializingBean extenders override methods out of the supertypes *supertype*.

Table 6: Excerpt of the most used method names with @Override

Supertype package	**Supertype**	**Methodname**	**Distinct inherited types (In this case: 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
junit.framework	TestCase	setUp	95
		tearDown	53
		Total: 2	Total distinct types that got an @Override Annotation: 113

3.3.4 An Additional Type-Method Report Filtered by @Override

Surprised, by the previous results of overridden methods, we depict further inherited types that have a huge amount of overridden methods. Our choice for the concrete types is made out of their name, indicating what they are used for. Additionally, we investigate

[10] http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/beans/factory/InitializingBean.html
[11] http://www.junit.org/apidocs/junit/framework/TestCase.html

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.

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".

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.

Table 7: Common inherited types and @Override marked methods in children

Supertype	**Methodname**	**Overriding Types**	**Exists in Supertype?**	**Additional methods in supertype**
Serializable	toString	42	no	
	equals	33	no	
	hashCode	32	no	
	getInviteeEmail	2	no	
	getInviteeFirstName	2	no	
	getInviteeLastName	2	no	
	clone	2	no	
	getInvitationType	2	no	
	initialiseHandler	1	no	
Transaction Listener Adapter	afterCommit	21	yes	flush
	afterRollback	11	yes	
	beforeCommit	6	yes	beforeCompletion
	equals	4	no	
	hashCode	3	no	
	toString	1	no	
Entity Lookup Callback DAOAdaptor	updateValue	5	yes	deleteByValue
	findByValue	5	yes	
	getValueKey	5	yes	
	deleteByKey	2	yes	

3.3.4 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 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 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 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 4a 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 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 4c. 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 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 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 4c 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.

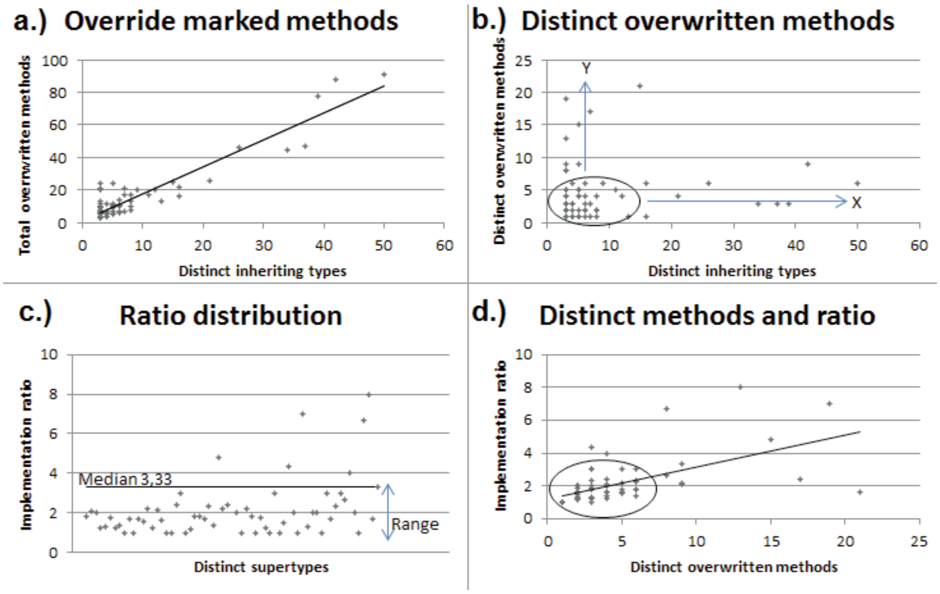

Figure 4: Relations of different inheritance indicators

4 Related Work

Related to the idea of Hypermodelling is the area of source code mining [GHH09]. Our approach to study software variance based on subclassing is especially near the idea to mine subclassing directives from existing code [BMM10]. Mining subclassing directives is about to uncover which methods are most used by extenders of classes to create a recommender system for developers to assist them in the integrated development environment. In order to generate these directives, the code is parsed and facts get extracted into a binary matrix by static analysis. Then, an algorithm is applied to generate directives. Our Hypermodelling approach overcomes the limitation to do a static analysis to uncover facts. Facts can be revealed via a query from the DW. Furthermore, we show that Hypermodelling is not limited to one fixed fact set: Plenty of facts can be extracted and visualized. Therefore, we already show that there exist more complex subclassing directives. For instance, methods that extend the *Serializable* interface often override three different methods of the object class. The method presented in the mining subclasses paper [BMM10] would not uncover this information. Moreover, our reports showed that different kinds of slices can be built to investigate source code and to visualize them to gain more knowledge. However, we do not see Hypermodelling in competition to such traditional mining approaches. Rather, Hypermodelling as integrative component to access the different facts about source code.

The usage of Business Intelligence is already mentioned in the literature. Since Business Intelligence is often used as a term for all associated technologies, like OLAP or DW this can be seen related. Software Intelligence (SI) [HX10] 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 how DW technology can be used. Lastly, no reports about code are shown and mentioned.

In [Pa03] a method to control a software development project with metrics is described. There, a Metrics Warehouse is mentioned. Since the measures computed in the paper represent somehow metrics of source code this seems first to be related. However, the metrics described are economic project figures, not representing code metrics. Even though, we think the idea to steer a project based on reports similar to the ones presented in this paper can be an interesting trail for the future.

5 Conclusions and Future Work

We presented different reports based on the Alfresco application. We explored different types of parent – children relations of software variance. We showed a difference in the variance of classes and interfaces at the exemplarily project level. Through drill downs we depicted variance differences of frameworks. Furthermore, we depicted two different figures to express the dependency on a framework. Investigations about the method variance of subclasses illuminated the fact: The methods of subclasses exceed probably a pure relation to supertypes methods. Commonly, there exists an undercurrent that methods of a supertypes subclasses share method names of another supertype. Thereby, we also demonstrated that Hypermodelling reports can be discriminated with dynamic factors, like metadata annotations, to enhance results. We showed diagrams that supported the following: The indicators, used within the paper to draw conclusions about inheritance variance seem to be linked. Lastly, we presented related work that described research that has potentially synergy effects with Hypermodelling.

However, the main intention of this paper is to present how the Hypermodelling approach can be applied to investigate software variance. We described at various points that we adjusted the queries. Thus, the key figure is that all statistics shown in this paper can be redone with queries. Those can easily be adjusted to the specific need. The diversity of the reports showed that we were capable to use the Hypermodelling approach to reveal facts about software variance that would be complex and expensive to be uncovered otherwise. The diversity of the reports was only possible because Hypermodelling is designed to uncover such facts with queries. All together, we see the flexibility of Hypermodelling as support that Hypermodelling can advance source code mining.

We see future work divided into two different trails. First, we think it is necessary to advance the Hypermodelling approach itself. That means that more facts and different cubes need to be built to enable other investigation of source code. Currently, we are planning extensions like caller-callee relationships. Secondly, we see the need to load a larger code base and different projects into the same warehouse to intensify and evaluate our investigations. Thereby, we see the necessity to derive standard reports for software variance. In order to create those, we see the presented reports as origin for further developments.

Nonetheless, Hypermodelling is mainly about the infrastructure to explore source code with queries. Therefore, we feel the emerging need to work together with source code mining experts and widen the possibilities of source code mining.

References

[Sc97] H.-A. Schmid. Systematic Framework Design by Generalization. Communications of the ACM, Vol.40/No.10, Oct. 97, pp. 48-51

[BMM10] M. Bruch, M. Mezini, and M. Monperrus, "Mining subclassing directives to improve framework reuse," in Proc. Working Conference on Mining Software Repositories. IEEE Computer Society, 2010, pp. 141–150.

[GHH09] M. W. Godfrey, A. E. Hassan, J. D. Herbsleb et al.. Future of mining software archives: A roundtable. IEEE. 2009

[ZZ11] 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

[Fr11] T. Frey. Vorschlag Hypermodelling: Data Warehousing für Quelltext. 23rd GI Workshop on Foundations of Databases. CEUR-WS, pages. 2011. pp.55-60.

[FKS11] 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. http://hypermodelling.com

[Fr12] T. Frey, Hypermodelling for Drag and Drop Concern Queries. Proceedings of Software Engineering 2010 (SE2012). Gesellschaft für Informatik (GI), Berlin, Germany. 2012

[In05] W. H. Inmon. Building the Data Warehouse. 4th ed., J.Wiley & Sons, New York. USA. 2005.

[HX10] 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

[Pa03] C. R. Pandian. Software metrics: a guide to planning, analysis and application. Auerbach Publications. 2003

[HKP11] J. Han, M. Kamber, J. Pei. Data Mining: Concepts and Techniques. Morgan Kaufmann. 3rd edition. 2011.

SE 12 SOFTWARE ENGINEERING

Zertifizierung und modellgetriebene Entwicklung sicherer Software (ZeMoSS)

ZeMoSS-Workshop: Zertifizierung und modellgetriebene Entwicklung sicherer Software

Michaela Huhn[1] Stefan Gerken[2] Carsten Rudolph[3]
[1] Institut für Informatik, Technische Universität Clausthal
[2] IC MOL RA R&D, Siemens AG, Braunschweig
[3] Fraunhofer Institut für Sichere Informationstechnologie (SIT), Darmstadt

Mit dem vielfältigen Einsatz softwaregesteuerter Produkte und Infrastrukturen in unserem Alltag wachsen die Software-Qualitätsanforderungen, sowohl bezüglich der funktionalen Sicherheit als auch bezüglich der Informationssicherheit. In der Luft- und Raumfahrt, der Energieerzeugung und im Schienenverkehr, aber auch in der Medizintechnik, der Automobiltechnik und bei mobilen Systemen sind Zertifizierung und der Nachweis der Sicherheit kritischer Systeme und softwarespezifische Sicherheitsnormen international etabliert und bindend. Zwei aktuelle, domänenübergreifende Herausforderungen bei der Entwicklung sicherer Software werden im Workshop adressiert:

- Modellgetriebene Entwicklung eingebetteter Software wird in der Industrie immer wichtiger und in ihren Grundlagen für höhere Sicherheitsanforderungsstufen seit langem in Sicherheitsnormen als dringend empfohlen klassifiziert. Da Normen aber immer nur die etablierten Regeln der Technik darstellen, entsteht für den Hersteller mit jedem Schritt hin zum erweiterten Einsatz modellgetriebener Methoden und Werkzeuge die Herausforderung, dass diese im Zertifizierungsprozess neu akzeptiert werden müssen, selbst wenn noch keine normativen Aussagen zu ihnen vorliegen.
- Durch die zunehmende Vernetzung kritischer Infrastrukturen und die Anbindung mobiler Endgeräte entstehen neue Risiken aus der wechselseitigen Abhängigkeit von Informationssicherheit und funktionaler Sicherheit. Hier sind eine Integration von Safety- und Security-Prozess und neue Methoden gefragt, die eine verbindende Behandlung von funktionaler Sicherheit und Informationssicherheit in der Risikoanalyse, der Entwicklung und beim Sicherheitsnachweis unterstützen.

Der ZeMoSS-Workshop soll den Austausch über offene Fragen und Lösungsansätze zu diesen Herausforderungen domänenübergreifend zwischen Teilnehmern aus Industrie und Forschung fördern.

Die Einladung zur Einreichung von Beiträgen führte zu sieben Einreichungen. Jede Einreichung wurde von mindestens drei Programmkomiteemitgliedern, assistiert durch externe Gutachter, begutachtet. Das Programmkomitee nahm fünf Beiträge zur Veröffentlichung im Tagungsband an, ein weiterer, nachträglich eingereichter Beitrag konnte für einen Vortrag zugelassen werden. Zusätzlich konnten wir mit Prof. Dr. Jens Braband, einen ausgewiesenen Experten für Sicherheit in der Bahntechnik, eine Key Note gewinnen:

Jens Braband: Security und Safety in der Eisenbahnsignaltechnik

Am Beispiel der Eisenbahnsignaltechnik wird die Wechselwirkung der Produkteigenschaften Safety und Security motiviert und erläutert. Dabei wird sowohl die Normenlandschaft beleuchtet als auch Beispiele für Anwendungen gezeigt. Es werden Erfahrungen bei der Erstellung eines Schutzprofils nach Common Criteria diskutiert.

Die folgenden Beiträge wurden angenommen:

- Kristian Beckers und Stephan Faßbender: Supporting the Context Establishment according to ISO 27005 using Patterns
- Marcus Mews und Steffen Helke: Towards Static Modular Software Verification
- Frank Ortmeier, Simon Struck und Michael Lipaczewski: Using model-based analysis in certification of critical software-intensive systems
- Patrick Werner, Stefan Gerken und Michaela Huhn: GSN$_M$-Edit: Ein modellgetriebener Editor für modulare GSN-Argumentationen
- Christian Wessel, Thorsten Humberg, Sven Wenzel und Jan Jürjens: Frühzeitige modellbasierte Risikoanalyse für mobile, verteilte Anwendungen

Wir möchten allen Beteiligten für ihren Beitrag zum Gelingen des ZeMoSS-Workshops danken: Unser Dank gilt den Autoren und dem eingeladenen Vortragenden für ihre Beiträge zum Programm. Wir danken den Programmkomiteemitgliedern und externen Gutachtern. Sie haben die Einreichungen in kurzer Zeit begutachtet und den Autoren nützliches Feedback gegeben. Das Programmkomitee, geleitet durch Michaela Huhn, Stefan Gerken und Carsten Rudolph, bestand aus:

Jan Jürjens	Technische Universität Dortmund
Volkmar Lotz	SAP AG
Marco Hauri	Ascom Systec AG
Hardi Hungar	Offis, Oldenburg
Stephan Katzenbeißer	Technische Universität Darmstadt
Lothar Pfeifer	Esterel Technologies
Heiko Saalbach	Movares Deutschland GmbH
Bernhard Schätz	fortiss, München

Schließlich möchten wir den lokalen Organisatoren der SE'12 für ihre Unterstützung im gesamten Prozess danken.

Wir hoffen, dass der ZeMoSS-Workshop für alle Teilnehmenden ein interessantes und stimulierendes Ereignis wird, aus dem sich neue Perspektiven für das Gebiet ergeben.

Februar 2012

Michaela Huhn (TU Clausthal)
Stefen Gerken (Siemens AG)
Carsten Rudolph (Fraunhofer SIT)
PC Chairs ZeMoSS'12