Modeling History

to Understand Software Evolution

Inauguraldissertation der
Philosophisch-naturwissenschaftlichen
Fakultät der Universität Bern

vorgelegt von

Tudor Gîrba

von Rumänien

Leiter der Arbeit:

Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz

Institut für Informatik und
angewandte Mathematik

Inauguraldissertation der Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern

vorgelegt von

Tudor Gîrba

von Rumänien

Modeling History

to Understand Software Evolution

Leiter der Arbeit:

Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz

Institut für Informatik und
angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 14.11.2005

Der Dekan:

Prof. Dr. P. Messerli

Acknowledgments

The work presented here goes well beyond the words and pictures, and into the very intimate corners of my existence. This work only holds what can be represented, while the real value lies somewhere in between me and the people that surrounded me.

I am grateful to Stéphane Ducasse and to Oscar Nierstrasz for giving me the opportunity to work at Software Composition Group based only on my engagement that I will be an "even better student." Stef, you were always energetic and supported me in so many ways. Oscar, you gave me the balance I needed at difficult times. Thank you both for being more than my supervisors.

I thank Harald Gall for promptly accepting to be an official reviewer, as well as for the good time we had at various meetings. I thank Gerhard Jăger for accepting to chair the examination.

Radu Marinescu was my first contact with research, yet it was not only research that he introduced me to. We had great and fruitful times together. Radu, I am grateful for many things, but most of all for the time we spent around a black liquid.

Michele Lanza pushed and promoted me with every occasion he could find. Michele, thank you for all the discussions, projects and the cool trips. I hope they will never stop.

Roel Wuyts encouraged me all along the way, and together with his wife, Inge, shared their house with me during my first days in Berne. Roel and Inge, thank you for making me feel like at home in a foreign country.

Jean-Marie Favre was always fun to work with. Jean-Marie, thank you for intro-

ducing me to the different facets of the meta world.

I thank Serge Demeyer for offering to review this dissertation. Serge, I was surprised and I was delighted.

Many thanks go to the Software Composition Group members. I thank Orla Greevy for trusting me and for reviewing this dissertation. I thank Gabriela Arévalo for the patience of sharing her incredible skills of dealing with the curse of the modern man: the bureaucracy. I thank Alex Bergel for not being tired of playing chess, Markus Gälli for the small-but-not-short-talks, Marcus Denker for always being calm and supportive, Laura Ponisio for encouraging me to eat healthier, Matthias Rieger for the best movie discussions, and Juan Carlos Cruz for the nice parties. Thanks go to Sander Tichelaar for never being upset with my bad jokes, Adrian Lienhard and Nathanael Schärli for showing me the Swiss way, and Therese Schmid for her kindness throughout the years.

Much of this work came out from the collaboration with my diploma students: Thomas Bühler, Adrian Kuhn, Mircea Lungu, Daniel Raţiu, Mauricio Seeberger. Daniel was my first diploma student and he had the patience of transforming my indecision from the beginning into something constructive. Mircea always found a way to make me exercise other perspectives. Thomas was a hard worker. Adrian and Mauricio made a great and joyful team together. Thank you all for treating me as your peer.

I am grateful to my Hilfsassistants – Niklaus Haldiman, Stefan Ott and Stefan Reichhart – for making my life so easy during the ESE lecture.

I thank the people at the LOOSE Research Group: Iulian Dragoş, Rãzvan Filipescu, Cristina Marinescu, Petru Mihancea, Mircea Trifu. I also thank the people that I met throughout my computer related existence, and that influenced me in one way or another: Bobby Bacs, Dan Badea, Dan Corneanu, Ştefan Dicu, Danny Dig, Bogdan Hodorog, Radu Jurcã, Gerry Kirschner, Rãzvan Pocaznoi, Adi Pop, Adi Trifu.

Much of me roots in the years of high-school, and for that I thank: Ciprian Ghirdan, Sorin Kertesz, Şerban Filip, Bogdan Martin, Cosmin Mocanu, Sergiu Ţent. I thank Andrei Mitraşcã for not being tired of talking nonsense with me, and I thank Codruţ and Bianca Morariu for being the best “finuţi” me and Oana have.

I thank Rãzvan and Silvia Tudor for the many visits to Bern, and I hope these visits will not stop.

Sorin and Camelia Ciocan took me in their home when I was alone. Thank you for being the friends you are for both me and Oana.

Nothing would have been possible were it not for my beloved parents. I thank you for being patient with me when I did not understand, and for believing in me when you did not understand. Thank you for keeping me safe and happy.

I thank Sorin, Corina and Ovidiu for taking care of my parents.

I will never forget the countless trips to and from Budapest together with Adina and Claudiu Pantea. Thank you for the support and cheer.

Aura and Ionicã Popa made me part of their family as if this was the most natural thing of all. Thank you for being the great in-laws that you are.

The joy of these years came from the love of my wife, and if I had success it was due to her unconditional support. Oana, my thanks to you extend beyond reasons, although there are plenty of reasons. This work is dedicated to you and your smile.

October 23, 2005
Tudor Gîrba

To Oana

Abstract

Over the past three decades, more and more research has been spent on understanding software evolution. The development and spread of versioning systems made valuable data available for study. Indeed, versioning systems provide rich information for analyzing software evolution, but it is exactly the richness of the information that raises the problem. The more versions we consider, the more data we have at hand. The more data we have at hand, the more techniques we need to employ to analyze it. The more techniques we need, the more generic the infrastructure should be.

The approaches developed so far rely on *ad-hoc* models, or on too *specific* meta-models, and thus, it is difficult to reuse or compare their results. We argue for the need of an explicit and generic meta-model for allowing the expression and combination of software evolution analyses. We review the state-of-the-art in software evolution analysis and we conclude that:

> *To provide a generic meta-model for expressing software evolution analyses, we need to recognize the evolution as an explicit phenomenon and model it as a first class entity.*

Our solution is to encapsulate the evolution in the explicit notion of *history* as a sequence of *versions*, and to build a meta-model around these notions: Hismo. To show the usefulness of our meta-model we exercise its different characteristics by building several reverse engineering applications.

This dissertation offers a meta-model for software evolution analysis yet, the concepts of history and version do not necessarily depend on software. We show how the concept of history can be generalized and how we can obtain our meta-model by transformations applied on structural meta-models. As a consequence, our approach of modeling evolution is not restricted to software analysis, but can be applied to other fields as well.

Table of Contents

List of Figures

Chapter 1

Introduction

The coherence of a trip is given by the clearness of the goal.

During the 1970's it became more and more apparent that keeping track of software evolution was important, at least for very pragmatic purposes such as undoing changes. Early versioning systems like the Source Code Control System (SCCS) made it possible to record the successive versions of software products [Rochkind, 1975]. At the same time, text-based delta algorithms were developed [Hunt and McIlroy, 1976] for understanding where, when and what changes appeared in the system. Some basic services were also added to model extra or meta information such as who changed files and why. However only very rudimentary models were used to represent this information – typically a few unstructured lines of text to be inserted in a log file.

While versioning systems recorded the history of each source file independently, configuration management systems attempted to record the history of software products as a collection of versions of source files. This arose research interests on the topic of configuration management during 1980's and 1990's, but the emphasis was still on controlling and recording software evolution.

The importance of *modeling and analyzing* software evolution was pioneered in the early 1970's with the work of Lehman [Lehman and Belady, 1985], yet, it was only in recent years that extensive research has been carried out on exploiting the

wealth of information residing in versioning repositories for purposes like reverse engineering or cost estimation. Problems like software aging [Parnas, 1994] and code decay [Eick *et al.*, 2001] gained increasing recognition both in academia and in industry.

The research effort spent on the issue of software evolution shows the importance of the domain. Various valuable approaches have been proposed to analyze aspects of software evolution for purposes like identifying driving forces in software evolution [Buckley *et al.*, 2005; Capiluppi *et al.*, 2004; Lehman and Belady, 1985], like prediction [Hassan and Holt, 2004; Zimmermann *et al.*, 2004], or like reverse engineering [Ball and Eick, 1996; Collberg *et al.*, 2003; Fischer *et al.*, 2003b; Lanza and Ducasse, 2002; Van Rysselberghe and Demeyer, 2004; Wu *et al.*, 2004b]. However, if we are to understand software evolution as a whole, we need means to combine and compare the results of the different analyses.

1.1 The Problem of Meta-Modeling Software Evolution

To combine or compare analyses we need a common meta-model, because a common meta-model allow for a uniform expression of analyses.

Short Incursion into Meta-Modeling

To *model* is to create a *representation* of a *subject* so that reasonings can be formulated about the subject. The model is a simplification of the subject, and its purpose is to answer some particular questions aimed towards the subject [Bézivin and Gerbé, 2001]. In other words, the model holds the elements of the subject that are relevant for the purpose of the reasoning. For example, a road map is a model of the actual physical space, and its purpose is to answer questions regarding the route to be taken. The same map is completely useless, when one would want to answer the question of where is the maximum precipitation point.

Several classes of models may exist for a given subject, each of these classes of models being targeted for some particular reasonings. Hence, not all models represent the same elements of the subject. That is to say, a model is built according to a description of what can go into a class of models: the meta-model [Seidewitz, 2003]. For example, we can have several types of maps representing the same

physical space. Each of these types of maps will represent characteristics of the physical space based on a specification: a road map might only show circles to represent places and lines to represent roads, while a precipitation map will use colors to represent the amount of precipitation in a region.

To understand a model one must understand its meta-model. A particular reasoning is dependent on what can be expressed on the respective underlying model, that is, it is dependent on the meta-model. Given an unknown reasoning, the basic step towards understanding it is to understand the meta-model.

In our map example, we need to understand the map to understand the reasonings built on it, and for understanding the map we need to understand what is represented in the map. For example, someone looks at the map to find the path from A to B, and she decides that the next step is to get to C. If we want to question why she chose C, we need to first understand the notation of the map and then to check if indeed C is the right choice.

The Problem of Meta-Modeling Software Evolution

Current approaches to analyze software evolution typically focus on only some traits of software evolution, and they either rely on *ad-hoc* models (*i.e.,* models that are not described by an explicit meta-model), or their meta-models are specific to the goals of the supported analysis.

A meta-model describes the way the domain can be represented by the model, that is, it provides bricks for the analysis. An explicit meta-model allows for understanding those bricks. Understanding the bricks allows for the comparison of the analyses built on top. Without an explicit meta-model, the comparison and combination of results and techniques becomes difficult.

Many approaches rely on the meta-model provided by the versioning system, and often no semantic units are represented (*e.g.,* packages, classes or methods), and there is no information about what exactly changed in a system. For example, it is difficult to identify classes which did not add any method recently, but changed their internal implementation.

There is no *explicit entity* to which to assign the properties characterizing how the entity evolved, and because of that, it is difficult to combine the evolution information with the version information. For example, we would like to know whether a large class was also changed a lot.

Problem Statement:

> *Because current approaches to analyze software evolution rely on ad-hoc models or on too specific meta-models, it is difficult to combine or compare their results.*

The more versions we consider, the more data we have at hand. The more data we have at hand, the more difficult it is to analyze it. The more data we have, the more techniques we need to employ on understanding this data. The more techniques we need, the more generic the infrastructure (either theoretical or physical) should be.

Research Question:

> *How can we build a generic meta-model that enables the expression and combination of software evolution analyses?*

1.2 Our Approach in a Nutshell

Thesis:

> *To provide a generic meta-model for expressing software evolution analyses, we need to recognize the evolution as an explicit phenomenon and model it as a first class entity.*

In this dissertation we propose Hismo as an explicit meta-model for software evolution. Hismo is centered around the notion of *history* as a sequence of *versions*.

Figure 1.1 (p.5) displays the core of our meta-model displaying three entities: *History*, *Version* and *Snapshot*. The Snapshot is a placeholder for the entities whose evolution is under study (*e.g.,* class, file). A History holds a set of Versions, where the Version adds the notion of time to the Snapshot.

With this scheme, time information is added on top of the structural information, that is, the structural information can exist without any reference to history, but can still be manipulated in the historical context. In other words, Hismo can be built on top of any snapshot meta-model without interfering with the existing meta-model. In this way, with Hismo we can reuse analyses at the structural level and extend them in the context of evolution analysis.

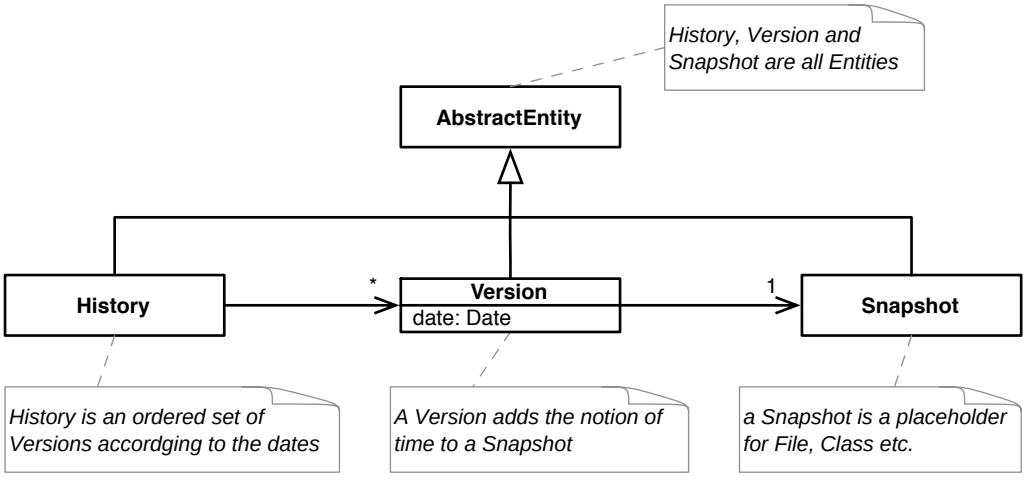


Figure 1.1: Details of the relationship between the History, the Version and the Snapshot.

History and Version are generic entities, and they need to be specialized for particular Snapshot entities. That is, for each Snapshot entity we have a corresponding History and Version entity. Building the historical representation for different Snapshot entities is a process that depends only on the type of the entities. Because of this property, we can generalize our approach of obtaining Hismo through a transformation applied to the snapshot meta-model. In this way, Hismo is not necessarily tied to software evolution analysis, but it is applicable to any snapshot meta-model.

To show the expressiveness of Hismo we describe how several software evolution analyses can be expressed with it. As a simple use of Hismo, we define different measurements for histories that describe how software artifacts evolved. We present different examples of historical measurements and history manipulations and show different reverse engineering analyses we have built over the years. Each of these examples exercises different parts of Hismo (see next Section for details).

1.3 Contributions

The novelty of this dissertation resides in providing a generic meta-model for software evolution [Gîrba *et al.*, 2004c; Ducasse *et al.*, 2004]. Our meta-model con-

tinuously evolved as a result of exercising it from the points of view of different reverse engineering analyses that we built over time:

1. Knowing where to start the reverse engineering process can be a daunting task due to size of the system. *Yesterday's Weather* is a measurement showing the retrospective empirical observation that the parts of the system which changed the most in the recent past also suffer important changes in the near future. We use *Yesterday's Weather* for guiding the first steps of reverse engineering by pin-pointing the parts that are likely to change in the near future [Gîrba *et al.*, 2004a].

2. Traditionally, design problems are detected based on structural information. Detection strategies are expressions that combine several measurements to detect design problems. We define history-based detection strategies to use historical measurements to refine the detection of design problems [Raţiu *et al.*, 2004].

3. Understanding the evolution of software systems is a difficult undertaking due to the size of the data. Hierarchies typically group classes with similar semantics. Grasping an overview of the system by understanding hierarchies as a whole reduces the complexity of understanding as opposed to understanding individual classes. We developed the *Hierarchy Evolution Complexity View* visualization to make out different characteristics of the evolution of hierarchies [Gîrba *et al.*, 2005b; Gîrba and Lanza, 2004].

4. Conway's law states that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" [Conway, 1968]. Hence, to get the entire picture one must understand the interaction between the organization and the system. We exploit the author information related to a commit in the versioning system to detect what are the zones of influence of developers and what are their behaviors [Gîrba *et al.*, 2005a].

5. As systems evolve, changes can happen to crosscut the system's structure in a way not apparent at the structural level. Understanding how changes appear in the system can reveal hidden dependencies between different parts of the system. We used concept analysis to detect co-change patterns, like parallel inheritance [Gîrba *et al.*, 2004b].

6. Hismo is implemented in VAN, a tool built on top of the MOOSE reengineering environment [Ducasse *et al.*, 2005]. MOOSE supports the integration of different reengineering tools by making the meta-model explicit and by pro-

viding mechanisms to extend the meta-model as needed by the particular analysis [Nierstrasz *et al.*, 2005]. The tools built on top, take as input entities with their properties and relationships. VAN extends the meta-model with the Hismo and provides several evolution analyses also by using tools built on MOOSE. For example, it uses CODECRAWLER to visualize the histories.

1.4 Structure of the Dissertation

Chapter 2 (p.11) browses the state-of-the-art in the analysis of software evolution, the focus being on making explicit the underlying meta-models of the different approaches. The result of the survey is summarized in a set of requirements for a meta-model for software evolution: (1) different abstraction and detail levels, (2) comparison of property evolutions, (3) combination of property evolutions, (4) historical selection, (5) historical relationships, and (6) historical navigation.

Chapter 3 (p.29) introduces Hismo, our meta-model for understanding software evolution. We show how Hismo centers around the notion of history as a first class entity, and how one can obtain a history representation starting from the meta-model of the structure. The following chapters show several approaches for software evolution analysis, each of them exercising a part of Hismo.

Chapter 4 (p.47) describes the *Yesterday's Weather* measurement as an example of how to combine several historical properties into one analysis. The problem addressed is to identify the parts of the system to start the reverse engineering process from, and the assumption is that the parts that are likely to get changed in the near future are the most important ones. *Yesterday's Weather* shows the retrospective empirical observation that the parts of the system which changed the most in the recent past also suffer important changes in the near future. We apply this approach to three case studies with different characteristics, and show how we can obtain an overview of the evolution of a system and pinpoint the parts that might change in the next versions.

A *detection strategy* is a an expression that combines measurements to detect design problems [Marinescu, 2004]. In **Chapter 5** (p.67) we extend the concept

of detection strategy with historical information to form *history-based detection strategies*. That is, we include historical measurements to refine the design flaws detection. History-based detection strategies show how, based on Hismo, we can express predicates that combine properties observable at the structural level with properties observable in time.

Chapter 6 (p.87) presents an approach to understand how a set of given class hierarchies as a whole have evolved over time. The approach builds on a combination of historical relationships and historical properties. We propose a set of queries to detect several characteristics of hierarchies: how old they are, how much were the classes changed, and how were the inheritance relationships changed.

As systems change, the knowledge of the developers becomes critical for the process of understanding the system. **Chapter 7** (p.105) presents an approach to understand how developers changed the system. We visualize how files were changed, by displaying the history of a file as a line and coloring the line according to the owners of the file over the time. To detect zones of influence of a particular developer, we arrange the files according to a clustering depending on the historical distance given by how developers changed the file: two files are closer together if they are changed in approximately the same periods.

Chapter 8 (p.125) introduces our approach of using Formal Concept Analysis to group entities that change in similar ways. Formal Concept Analysis takes as input elements with properties and returns concepts formed by elements with a set of common properties. As elements we use histories, and for each history we consider it has the property i if it "was changed in the ith version." We use the detailed information to distinguish different types of changes according to the different types of concepts we are interested in.

Much of the development of Hismo comes from our experience with implementing our assumptions in our prototype named VAN, built on top of the MOOSE reengineering environment [Ducasse *et al.*, 2005; Nierstrasz *et al.*, 2005]. **Chapter 9** (p.137) describes our prototype infrastructure. It implements Hismo and our approaches to understand software evolution. The chapter emphasizes how the usage of Hismo allows for the combination of techniques. For example, we show how we use two other tools (CodeCrawler [Lanza and Ducasse, 2005] and ConAn [Arévalo, 2005]) for building evolution analysis.

In **Chapter 10** (p.147) we discuss how Hismo leverages software evolution analysis

by re-analyzing the requirements identified in Chapter 2 (p.11) from the point of view of analyses built on Hismo . The chapter ends with an outlook on the future work opened by our approach.

We use several evolution and meta-modeling related terms throughout the dissertation (*e.g.,* evolution). In **Appendix A** (p.155) we provide the definitions for the most important of these terms.

Chapter 2

Approaches to Understand Software Evolution

Things are what we make of them.

Current approaches to understand software evolution typically rely on ad-hoc models, or their meta-models are specific to the goals of the supported analysis. We aim to offer a meta-model for software evolution analysis. We review the state of the art in software evolution analysis and we discuss the meta-models used. As a result we identified several activities the meta-model should support:

1. *It should provide information at different abstraction and detail levels,*
2. *It should allow comparison of how properties changed,*
3. *It should allow combination of information related to how different properties changed,*
4. *It should allow analyses to be expressed on any group of versions,*
5. *It should provide information regarding the relationships between the evolution of different entities, and*
6. *It should provide means to navigate through evolution.*

2.1 Introduction

In the recent years much research effort have been dedicated to understand software evolution, showing the increasing importance of the domain. The main challenge in software evolution analysis is the size of the data to be studied: the more versions are taken into account, the more data. The more data we have at hand, the more techniques we need to analyze it.

Many valuable approaches have been developed to analyze different traits of software evolution. However, if we are to understand software evolution as a whole, we need means to combine and compare the results of the different analyses.

The approaches developed so far rely on *ad-hoc* models, or their meta-models are specific to the goals of the supported analysis. Because of this reason it is difficult to combine or compare the results of the analyses.

Our goal is to build on the current state-of-the-art and to offer a common infrastructure meta-model for expressing software evolution analyses. To accomplish this task, we review several approaches for analyzing software evolution, the target being to identify the requirements of the different analyses from the point of view of a common evolution meta-model.

The most straightforward way to gather the requirements would be to analyze the different meta-models. Unfortunately, in most of the cases, the meta-models are not detailed (most of the time they are not even mentioned). In these cases we infer the minimal meta-models required for the particular analysis.

From our literature survey we identified two major categories of approaches depending on the granularity level of information representation: *version-centered* approaches and *history-centered* approaches. Version-centered approaches consider version as a representation unit, while the history-centered approaches consider history (*i.e.,* an ordered set of versions) as representation unit.

While in the version-centered approaches, the means is to present the version information and let the user detect the patterns, in the version-centered approaches the aim is to summarize what happened in the history according to a particular point of view. For example, a graphic plotting the values of a property in time is a version-centered approach; on the other hand, a measure of how a property evolved over time is a history-centered approach.

Structure of the Chapter

We discuss the version-centered approaches in Section 2.2 (p.13), and the history-centered approaches in Section 2.3 (p.19). While discussing the different approaches, we emphasize their requirements from the point of view of a common meta-model. Each of the two sections ends with a discussion of the approaches. Section 2.4 (p.25) concludes the chapter with an overall description of the gathered requirements

2.2 Version-Centered Approaches

The *version-centered* analyses use a version as a representation granularity. In general, they target the detection of *when* something happened in history. We identify three classes of approaches and we take a look at each.

2.2.1 Analyzing the Changes Between Two Versions

Comparing two versions is the basis of any evolution analysis. We enumerate several approaches that focus on finding different types of changes.

Diff was the first tool used for comparing the differences between two versions of a file [MacKenzie *et al.*, 2003]. Diff is able to detect addition or deletion of lines of text and it provides the position of these lines in the file. This tool is not useful when the analysis requires finer grained data about what happened in the system, because it does not provide any semantic information of what exactly changed in the system (*e.g.*, in terms of classes or functions).

In another work, Xing and Stroulia used a Diff-like approach to detect different types of fine-grained changes between two versions of a software system. They represented each version of the system in an XMI format [XMI 2.0, 2005] and then applied UML Diff to detect changes like: addition/removal of classes, methods and fields; moving of classes, methods, fields; renaming of classes, methods, fields. Several applications have been based on this approach [Xing and Stroulia, 2004a; Xing and Stroulia, 2004b; Xing and Stroulia, 2004c].

Demeyer *et al.* used the structural measurements to detect refactorings like rename method, or move method [Demeyer *et al.*, 2000]. They represented each

version with a set of metrics, and then identify changes based on analyzing the change in the measurements.

Antoniol and Di Penta used the similarity in vocabulary of terms used in the code to detect refactorings like: rename class, split class, or merge class [Antoniol and Di Penta, 2004]. They represented versions of classes with vectors holding the relevance of the different terms used in the system for the particular class, and they compare the distance between the vectors of different versions to detect the refactorings.

Holt and Pak proposed a detailed visualization of the changes of dependencies between two versions of several modules [Holt and Pak, 1996]. On the same structural representation of the modules, they show the new dependencies, the removed dependencies or the common dependencies.

Burd and Munro defined a set of measurements to quantify the dominance relations which are used to depict the complexity of the calls. They correlated the changes in these measurements with the types of maintainability activities [Burd and Munro, 1999].

Version control systems allow for descriptions of the modifications. These descriptions hide the meaning of the change, but usually, they are just ad-hoc text entries without any structure. Mockus and Votta analyzed these descriptions to classify the changes [Mockus and Votta, 2000]. They could distinguish between corrective, adaptive, inspection, perfective, and other types changes.

Summary:

> *Software evolution analyses need information for detecting changes at different levels of abstraction.*

2.2.2 Analyzing Property Evolutions: Evolution Chart

Since 1970 research is spent on building a theory of evolution by formulating laws based on empirical observations [Lehman *et al.*, 1998; Lehman, 1996; Lehman and Belady, 1985; Lehman *et al.*, 1997; Ramil and Lehman, 2001]. The observations are based on the interpretation of evolution charts which represent some property on the vertical axis (*e.g.,* number of modules) and time on the horizontal axis (see Figure 2.1 (p.15)). Gall *et al.* employed the same kind of approach while analyzing the evolution of a software system to identify discrepancies between the evolution of the entire system and the evolution of its subsystems

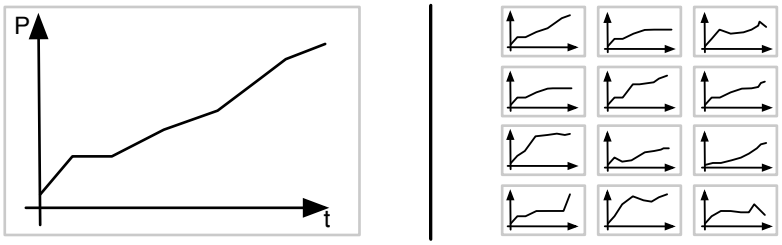


Figure 2.1: The evolution chart shows a property P on the vertical and time on the horizontal.

[Gall *et al.*, 1997]. Recently, the same approach has been used to characterize the evolution of open source projects [Godfrey and Tu, 2000; Capiluppi, 2003; Capiluppi *et al.*, 2004].

This approach is useful when we need to reason about the evolution of a single property, but it makes it difficult to reason in terms of more properties at the same time, and provides only limited ways to compare how the same property evolved in different entities. That is why, typically, the charts are used to reason about the entire system, though the chart can represent any type of entity.

In Figure 2.1 (p.15) we give an example of how to use the evolution charts to compare multiple entities. In the left part of the figure we display a graph with the evolution of a property P of an entity – for example it could represent number of methods in a class (NOM). From the figure we can draw the conclusion that P is growing in time. In the right part of the figure we display the evolution of the same property P in 12 entities. Almost all graphs show a growth of the P property but they do not have the same shape. Using the graphs alone it is difficult to say which are the differences between the evolution of the different entities.

Summary:

> *Software evolution analyses need to compare the information on how properties evolved.*

If we want to correlate the evolution of property P with another property Q, we have an even more difficult problem, and the evolution chart does not ease the task significantly. Chuah and Eick used so called "timewheel glyphs" to display several evolution charts where each chart was rotated around a circle [Chuah and Eick, 1998]. Each evolution chart plotted a different property: number of lines added, the errors recorded between versions, number of people working *etc.*.

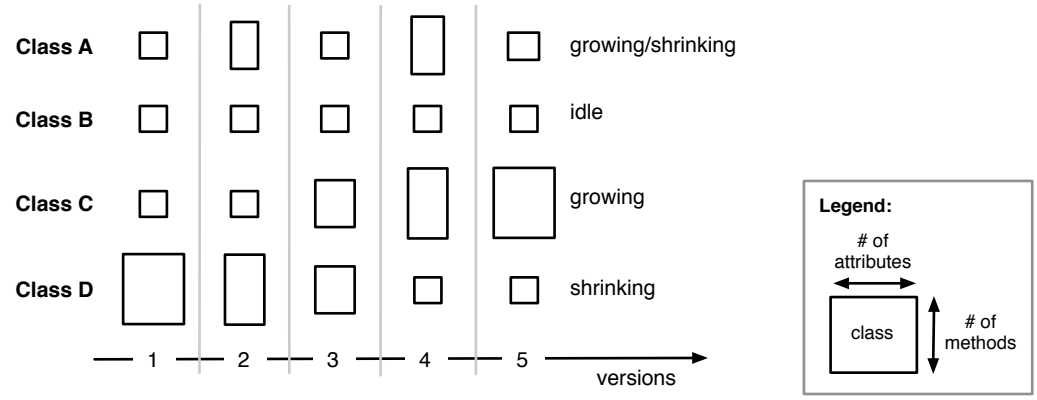


Figure 2.2: The Evolution Matrix shows versions nodes in a matrix. The size of the nodes is given by structural measurements.

They indeed used several evolution charts to allow the analysis to combine the information of how different properties evolved, but they only made use of the overall charts shape, and not the detailed differences.

Summary:

Software evolution analyses need to combine the information on how properties evolved.

2.2.3 Evolution Matrix Visualization

Visualization has been also used to reason about the evolution of multiple properties and to compare the evolution of different entities. Lanza and Ducasse arranged the classes of the history in an Evolution Matrix shown in Figure 2.2 (p.16) [Lanza and Ducasse, 2002]. Each rectangle represents a version of a class and each line holds all the versions of that class (the alignment is realized based on the name of the class). Furthermore, the size of the rectangle is given by different measurements applied on the class version. From the visualization different evolution patterns can be detected such as continuous growth, growing and shrinking phases etc.

Rysselberghe and Demeyer used a scatter plot visualization of the changes to provide an overview of the evolution of systems [Van Rysselberghe and Demeyer,

2004]. They used the visualization to detect patterns of change like: unstable components, coherent entities, design evolution and productivity fluctuations.

Jingwei Wu *et al.* used the spectograph metaphor to visualize how changes occur in software systems [Wu *et al.*, 2004a]. They used colors to denote the age of changes on different parts of the systems.

Jazayeri analyzed the stability of the architecture by using colors to depict for each version of a file how recent are the changes. From the visualization he concluded that old parts tend to stabilize over time [Jazayeri, 2002].

Taylor and Munro used a variation of an Evolution Matrix to visualize file changes with a technique called *revision towers* [Taylor and Munro, 2002]. The purpose of the visualization was to provide a one-to-one comparison between changes of two files over multiple versions.

Voinea *et al.* present a tool called CVSscan that places the analysis at the text line level [Voinea *et al.*, 2005]. They distinguish actions like: deletion of line, insertions of line and modifications of a line. Given a file, they show all versions of each line and map on each version different characteristics, like authors or line status.

With these visualizations, we can reason in terms of several properties at the same time, and we can compare different evolutions. The drawback of the approach resides in the implicitness of the meta-model: there is no explicit entity to which to assign the evolution properties. Because of that it is difficult to combine the evolution information with the version information. For example, we would like to know if the growing classes are large classes, like expressed by the following code written in the Object Constraint Language (OCL) [OCL 2.0, 2003]:

```
context Class

  -- should return true if the class is large and if it was detected as being growing
  derive isGrowingLargeClass: self.isLargeClass() & self.wasGrowing()
```

The above code shows how we would like to be able to put in one single automatic query, both evolution information (self.wasGrowing()), and structural information (self.isLargeClass()). We would only be able to express this if self would know both about the structure and about the evolution.

Summary:

> *Software evolution analyses need to combine the information on how properties evolved with the information from the structural level.*

Another drawback here is that the more versions we have, the more nodes we have, the more difficult it gets to detect patterns when they are spread over a large space.

2.2.4 Discussion of Version-Centered Approaches

The version-centered models allow for the comparison between two versions, and they provide insights into when a particular event happened in the evolution (*e.g.,* a class grew instantly). The visual technique is to represent time on an axis and place different versions along this axis and make visible where the change occurred (*e.g.,* using color, size, position).

Some of the analyses also used version-based techniques to compare the way different entities evolved over time. For example, the evolution chart was used to compare the evolution of different systems to detect patterns of change like continuously growing systems. The Evolution Matrix was also used to detect change patterns like growing classes or idle classes (*i.e.,* classes that do not change). A major technical problem is that the more versions we have the more information we have to interpret.

Furthermore, when patterns are detected, they are attached to structural entities. For example, the authors said that they detected growing and idle classes, yet, taking a closer look at the Evolution Matrix, we see that it is conceptually incorrect because a class is just one rectangle while growing and idle characterize a succession of rectangles. That is, we can say a class is big or small, but growing and idle characterizes the way a class has evolved. From a modeling point of view, we would like to have an explicit entity to which to assign the growing or idle property: the history as an encapsulation of evolution.

2.3 History-Centered Approaches

History-centered approaches have history as an ordered set of versions as representation granularity. In general, they are not interested in when something happened, but they rather seek to detect *what* happened and *where* it happened. In these approaches, the individual versions are no longer represented, they are flattened.

The main idea behind having a history as the unit of representation is to summarize the evolution according to a particular point of view. History-centered approaches often gather measurements of the history to support the understanding of the evolution. However, they are often driven by the information contained in repositories like Concurrent Versioning System (CVS)[1], and lack fine-grained semantic information. For example, some approaches offer file and folder changes but give no semantic information about what exactly changed in a system (*e.g.*, classes or methods).

We present three approaches characterizing the work done in the context of history-centered evolution analyses.

2.3.1 History Measurements

The history measurements aim to quantify what happened in the evolution of an entity. Examples of history measurements are: age of an entity, number of changes in an entity, number of authors that changed the system etc.

Ball and Eick developed multiple visualizations for showing changes that appear in the source code [Ball and Eick, 1996]. For example, they show what is the percentage of bug fixes and feature addition in files, or which lines were changed recently.

Eick *et al.* described Seesoft, a tool for visualizing line oriented software statistics [Eick *et al.*, 1992]. They proposed several types of visualization: number of modification reports touching a line of code, age, stability of a line of code *etc.*.

Mockus *et al.* implemented a tool called SoftChange for characterizing software evolution in general [Mockus *et al.*, 1999]. They used history measurements like: the age of a file, the average lines of code added/deleted in a change, the total number of changes happening at a certain hour etc.

[1] See https://www.cvshome.org/.

Mockus and Weiss used history measurements for developing a method for predicting the risk of software changes [Mockus and Weiss, 2000]. Examples of the measurements were: the number of modules touched, the number of developers involved, or the number of changes.

Eick *et al.* proposed several visualizations to show how developers change using colors and third dimension [Eick *et al.*, 2002]. For example they showed a matrix where each row corresponds to an author, each column corresponds to a module, and each cell in the matrix shows the size of the changes performed by the developer to the module.

Collberg *et al.* used graph-based visualizations to display which parts of class hierarchies were changed [Collberg *et al.*, 2003]. They provide a color scale to distinguish between newer and older changes.

Xiaomin Wu *et al.* visualized the change log information to provide for an overview of the active places in the system as well as of the authors activity [Wu *et al.*, 2004b]. They display measurements like the number of times an author changed a file, or the date of the last commit.

Chuah and Eick presented a way to visualize project information in a so called "infobug" [Chuah and Eick, 1998]. The name of the visualization comes from a figure looking like a bug. They mapped on the different parts of the "infobug" different properties: evolution aspects, programming languages used, and errors found in a software component. They also presented a time wheel visualization to show the evolution of a given characteristic over time.

Typically, in the literature we find measurements which are very close to the type of information available in the versioning systems. As versioning systems provide textual information like lines of code added/removed, the measurements too only measure the size of the change in lines of code. Even though lines of code can be a good indicator for general overviews, it is not a good indicator when more sensitive information is needed. For example, if 10 lines of code are added in a file, this approach does not distinguish whether the code was added to an existent method, or if several completely new methods were added.

Summary:

Software evolution analyses need to combine the information on how properties evolved.

2.3.2 Manipulating Historical Relationships: Historical Co-Change

Gall *et al.* aimed to detect logical coupling between parts of the system [Gall *et al.*, 1998] by identifying the parts of the system which change together. They use this information to define a coupling measurement based on the fact that the more times two modules were changed at the same time, the more strongly they were coupled.

Pinzger and coworkers present a visualization of evolution data using a combination of Kiviat diagrams and a graph visualization [Pinzger *et al.*, 2005]. Each node represents a module in the system and an edge connecting two nodes the historical dependencies between the two modules. For example, to the width of the edge the authors map the co-change history of the two modules. Each node in the graph is displayed using a Kiviat diagram to show how different measurements evolved. They use both code and file measurements. In this model, the authors see the nodes and edges from both the structural perspective and from the evolution perspective.

Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graphs [Hassan and Holt, 2004]. Zimmermann *et al.* defined a measurement of coupling based on co-changes [Zimmermann *et al.*, 2003].

Zimmermann *et al.* aimed to provide a mechanism to warn developers about the correlation of changes between functions. The authors placed their analysis at the level of entities in the meta-model (*e.g.*, methods) [Zimmermann *et al.*, 2004]. They presented the problems residing in mining the CVS repositories, but they did not present the meta-model [Zimmermann and Weißgerber, 2004].

Similar work was carried out by Ying *et al.* [Ying *et al.*, 2004]. The authors applied the approach on two large case studies and analyzed the effectiveness of the recommendations. They concluded that although the "precision and recall are not high, recommendations can reveal valuable dependencies that may not be apparent from other existing analyses."

Xing and Stroulia used the fine-grained changes provided by UML Diff to look for class co-evolution [Xing and Stroulia, 2004a; Xing and Stroulia, 2004b]. They took the type of changes into account when reasoning, and they distinguished between intentional co-evolution and "maintenance smells" (*e.g.*, Shotgun Surgery).

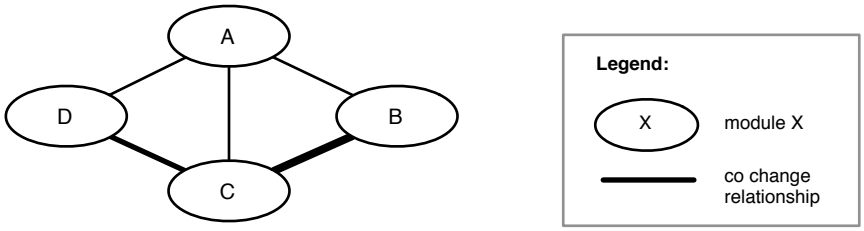


Figure 2.3: Historical co-change example. Each ellipse represents a module and each edge represents a co-change relationship. The thickness of the edge is given by the number of times the two modules changed together.

Eick *et al.* used the number of files changed in the same time as an one indicator of code decay [Eick *et al.*, 2001]. They reported on a large case study that changes are more dispersed at the end of the project, which they interpreted as a sign of code decay.

Burch visualized the co-change patterns in a matrix correlation [Burch *et al.*, 2005]. In their visualization, each line and column is given by a file, and the color of each point is given by the confidence of the co-change relationship between the two files. The confidence is a historical measurement and it is computed as the "number of changes of a pair of items relative to the number of changes of a single item."

In general, the result of the co-change analysis is that two entities (*e.g.*, files) have a relationship if they were changed together. Gall *et al.* provided a visualization, as in Figure 2.3 (p.22), to show how modules changed in the same time [Gall *et al.*, 2003]. The circles represent modules, and the edges represent the co-change relationship: the thicker the edge, the more times the two modules were changed in the same time. In this representation the structural elements from the last version (*i.e.*, the modules) are linked via a historical relationship (*i.e.*, the co-change relationship). In a similar visualization Eick *et al.* used the color to denote the strength of the relationship between the co-changed modules [Eick *et al.*, 2002].

As in the case of the Evolution Matrix (*e.g.*, where classes were said to be growing), in this case too there is a conceptual problem from the modeling point of view: co-change actually links the evolution of the entities and not a particular version of the entities. We would like to have a reification of the evolution (*i.e.*, history), to be able to relate it to the co-change relationship.

Summary:

Software evolution analyses need information about how histories are related from a historical perspective.

2.3.3 Manipulating Historical Entities: Hipikat and Release Meta-Models

Fischer *et al.* modeled bug reports in relation to version control system (CVS) items [Fischer *et al.*, 2003b]. Figure 2.4 (p.24) presents an excerpt of the Release History meta-model. The purpose of this meta-model is to provide a link between the versioning system and the bug reports database. This meta-model recognizes the notion of the history (*i.e.,* CVSItem) which contains multiple versions (*i.e.,* CVSItemLog). The CVSItemLog is related to a Description and to BugReports. Furthermore, it also puts the notion of Feature in relation with the history of an item. The authors used this meta-model to recover features based on the bug reports [Fischer *et al.*, 2003a]. These features get associated with a CVSItem.

The main drawback of this meta-model is that the system is represented with only files and folders, and it does not take into consideration the semantic software structure (*e.g.,* classes or methods). Because it gives no information about what exactly changed in a system, this meta-model does not offer support for analyzing the different types of change. Recently, the authors started to investigate how to enrich the Release History Meta-Model with source code information [Antoniol *et al.*, 2004].

Čubranić and Murphy bridged information from several sources to form what they call a "group memory" [Čubranić and Murphy, 2003]. Čubranić detailed the meta-model to show how they combined CVS repositories, mails, bug reports and documentation [Čubranić, 2004].

Draheim and Pekacki presented the meta-model behind Bloof [Draheim and Pekacki, 2003]. The meta-model is similar to CVS: a File has several Revisions and each Revision has attached a Developer. They used it for defining several measurements like the Developer cumulative productivity measured in changed LOC per day.

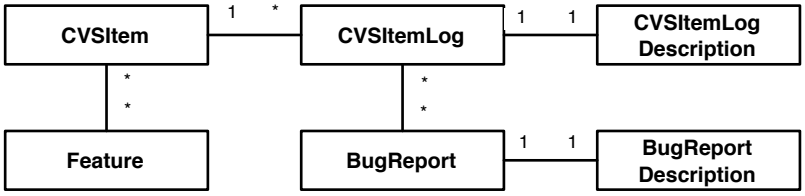


Figure 2.4: The Release History Meta-Model shows how Feature relates to CVSItem.

Summary:

Software evolution analyses need information about how histories are related from a historical perspective.

2.3.4 Discussion of History-Centered Approaches

While in the version-centered analyses, the approach was to present the version information and let the user detect the patterns, in the above examples, the aim is to summarize what happened in the history according to a particular point of view.

For example, an evolution chart displays the versioning data and the user can interpret it in different ways according to the point of view: she can see whether it grows or not, she can see whether it fluctuates or not and so on. As opposed to that, the history measurements encode these points of view and return the values that summarize the evolution. In this case, it is not the reengineer who has to identify the trends or patterns in the history, with the possibility of missing important information.

In general, analyses are influenced by the type of information available. For example, as versioning systems offer information related to the changes of the lines of code, the analyses, too, use addition/deletion of lines code as an indicator of change. While this might be suitable for general overviews, it is not enough for detailed analyses. For example, if we want to detect signs of small fixes, we might look for classes where no method has been added, while only the internal implementation changed.

2.4 Towards a Common Meta-Model for Understanding Software Evolution

A common meta-model for software evolution analysis should allow the expression of all of the above analyses and more. Below we present the list with the different needs of software evolution analyses:

> *Software evolution analyses need detailed information for detecting changes at different levels of abstraction.*
>
> *Software evolution analyses need to compare the information on how properties evolved.*
>
> *Software evolution analyses need to combine the information on how properties evolved with the information from the structural level.*
>
> *Software evolution analyses need to combine the information on how properties evolved.*
>
> *Software evolution analyses need information about how histories are related from a historical perspective.*

Taking the above list as an input we synthesize a list of features an evolution meta-model should support:

Different abstraction and detail levels. The meta-model should provide information at different levels of abstraction such as files, packages, classes, methods for each analyzed version. For example, CVS meta-model offers information about how source code changed (*e.g.*, addition, removals of lines of code), but it does not offer information about additions or removals of methods in classes.

The meta-model should support the expression of detailed information about the structural entity. For example, knowing the authors that changed the classes is an important information for understanding evolution of code ownership.

Comparison of property evolutions. The meta-model should offer means to easily quantify and compare how different entities evolved with respect to a certain property. For example, we must be able to compare the evolution of number of methods in classes, just like we can compare the number of methods in

classes. For that, we need a way to quantify how the number of methods evolve and afterwards we need to associate such a property with an entity.

Combination of different property evolutions. The meta-model should allow for an analysis to be based on the evolution of different properties. Just like we reason about multiple structural properties, we want to be able to reason about how these properties have evolved. For example, when a class has only a few methods, but has a large number of lines of code, we might conclude it should be refactored. At the same time, adding or removing the lines of code in a class while preserving the methods might lead us to conclude that the change was a bug-fix.

Historical relationships. The meta-model should provide information regarding the relationships between the evolution of different entities. For example, we should be able to reason about how two classes changed the number of children in the same time.

Besides the above ones, we introduce two additional generic requirements:

Historical navigation. The meta-model should provide relations between histories to allow for navigation. For example, we should be able to ask our model which methods ever existed in a particular class, or which classes in a particular package have been created in the last period of time.

Historical selection. The analysis should be applicable on any group of versions (*i.e.,* we should be able to select any period in the history).

2.5 Roadmap

In Chapter 3 (p.29), we present Hismo, our meta-model for software evolution. To validate Hismo we used it in several novel analyses that we present in Chapters 4-8.

In each of these chapters we present the analysis in detail, and at the end of the chapter we discuss which part of Hismo does the analysis exercise in a section called *Hismo Validation*. Figure 2.5 (p.27) shows schematically the chapters, the titles of the analyses, and the different features of Hismo they use.

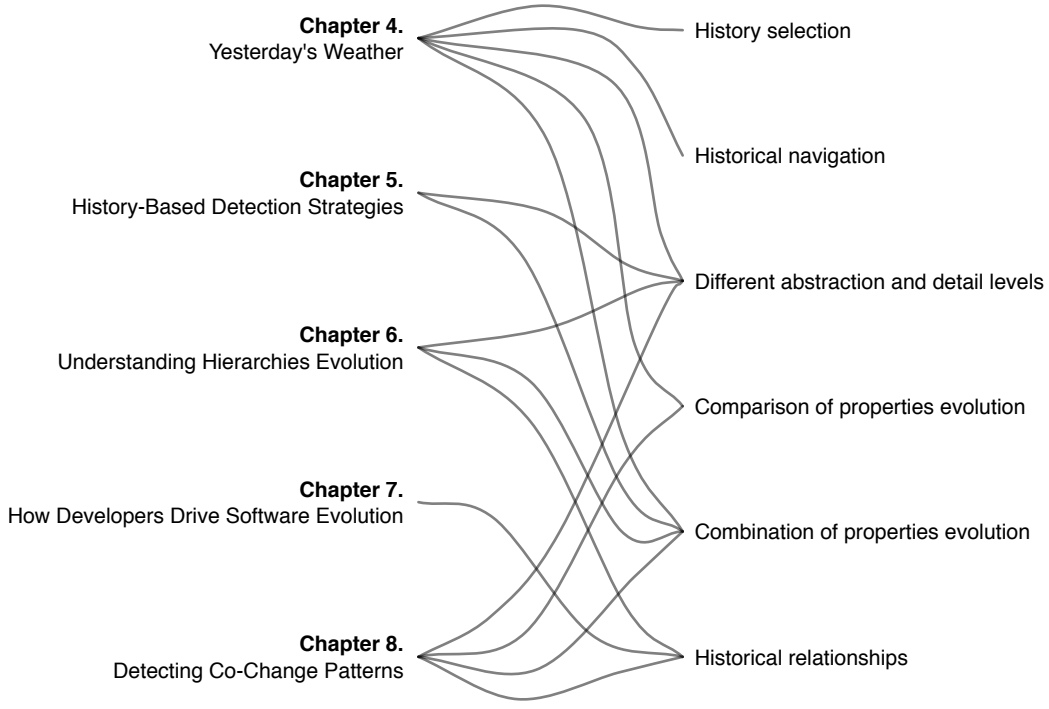


Figure 2.5: The different analyses built using Hismo and the different features of Hismo they use.

Chapter 3

Hismo: Modeling History as a First Class Entity

Every thing has its own flow.

Our solution to model software evolution is to model history as a first class entity. A history is an ordered set of versions and it encapsulates the evolution. History and version are generic concepts and they must be applied to particular entities like packages or classes. We show how starting from any snapshot meta-model we can obtain the historical meta-model through meta-model transformation.

3.1 Introduction

The previous chapter reviews several approaches to understanding software evolution and discusses their needs from the meta-model point of view. We identified several characteristics of a meta-model that supports all these analyses: (1) different abstraction and detail levels, (2) comparison of property evolutions, (3) combination of property evolutions, (4) historical selection, (5) historical relationships, and (6) historical navigation.

In this chapter we introduce Hismo, our solution of modeling history to support software evolution analyses: explicitly model *history* as an ordered set of *versions*.

Structure of the chapter

In the next section we introduce the generic concepts of history, version and snapshot, and discuss their generic relationships. In Section 3.6 (p.39) we introduce the notion of group as a first class entity and we discuss the properties of a group of histories. In Section 3.3 (p.32) we show how we build Hismo based on a snapshot meta-model, and in Section 3.4 (p.34) we provide an example of how Hismo can be mapped to the Evolution Matrix. In Section 3.5 (p.35) we define historical properties and we show how they summarize the evolution. In Section 3.7 (p.41) we discuss the problem of modeling historical relationships. In Section 3.8 (p.43) we generalize Hismo by showing how it is possible to generate the historical meta-model starting from the snapshot meta-model.

3.2 Hismo

The core of Hismo is based on three entities: *History*, *Version* and *Snapshot*. Figure 3.1 (p.31) shows the relationships between these entities in a UML 2.0 diagram [Fowler, 2003]:

Snapshot. This entity is a placeholder that represents the entities whose evolution is studied *i.e.,* file, package, class, methods or any source code artifacts. The particular entities are to be sub-typed from Snapshot as shown in Figure 3.3 (p.33).

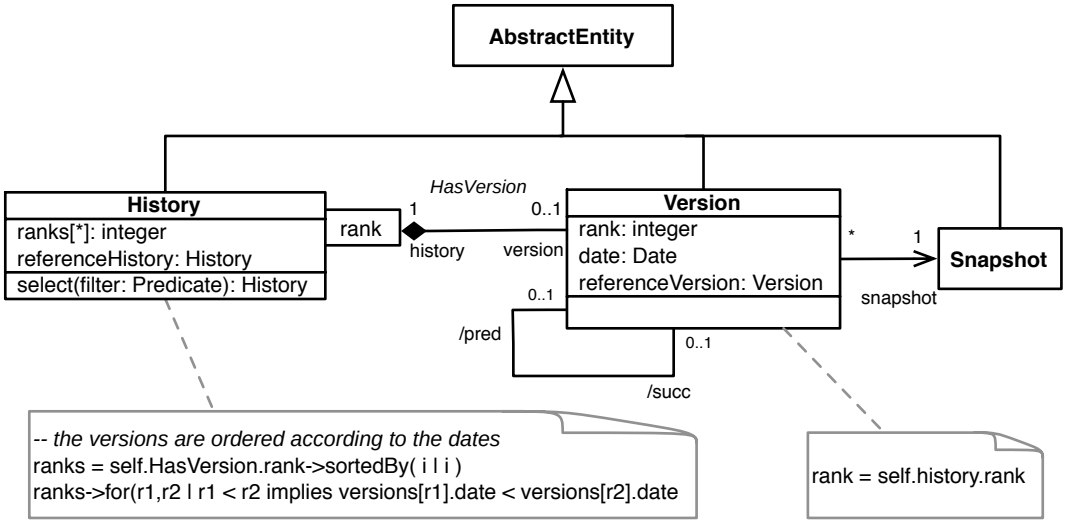


Figure 3.1: Details of the relationship between the History, the Version and the Snapshot. A History has a container of Versions. A Version wraps a Snapshot and adds evolution specific queries.

Version. A Version adds the notion of time to a Snapshot by relating the Snapshot to the History. A Version is identified by a time-stamp and it knows the History it is part of. A Version can exist in only one History. Based on its rank in the History, Version has zero or one predecessor and zero or one successor. Each Version has a reference to the so called referenceVersion which provides a fixed point in the historical space.

History. A History holds a set of Versions. The relationship between History and Version is depicted with a qualified composition which depicts that in a History, each Version is uniquely identified by a rank. From a History we can obtain a sub-History by applying a filter predicate on the set of versions. Each History has a reference to the so called referenceHistory which defines the historical space.

In Hismo, we add time information as a layer on top of the snapshot information. As such, the snapshot data can exist without any reference to history but can still be manipulated in the context of software evolution. Because of this, Hismo can be built on top of *any* snapshot meta-model without interfering with the existing meta-model. There are many meta-models describing structural information, and many analyses are built on these meta-models. With our approach of augmenting

time information on top of structural information, we can reuse the analyses at structural level and incorporate them in the evolution analysis.

History, Version and Snapshot are abstract and generic entities, and as such, the core of Hismo is not tied to any meta-model. These concepts are generic in the sense that they do not hold any specific information for a particular analysis. They provide a framework in which evolution information is represented, but to actually make the meta-model useful one has to *apply* these concepts on specific entities.

For example, we argued that we need detailed information about the different entities in the system such as packages, classes, methods. Figure 3.3 (p.33) shows an example of how Hismo looks like when applied to the FAMIX meta-model [Demeyer *et al.*, 2001].

3.3 Building Hismo Based on a Snapshot Meta-Model

In this section we unveil the details of how to apply the generic concepts of History, Version and Snapshot to specific snapshot meta-models.

We start by taking a detailed look at Hismo applied to Packages and Classes (see Figure 3.2 (p.33)). There is a parallelism between the version entities and the history entities: Each version entity has a corresponding history entity. Also, the relationship at version level (*e.g.,* a Package has more Classes) has a correspondent at the history level (*e.g.,* a PackageHistory has more ClassHistories).

Figure 3.3 (p.33) shows an overview of the history meta-model based on a larger source-code meta-model. Here we use FAMIX, a language independent source code meta-model [Demeyer *et al.*, 2001]. The details of the full meta-model are similar to the one in Figure 3.2 (p.33).

The snapshot entities (*e.g.,* Method) are wrapped by a Version correspondent (*e.g.,* MethodVersion) and the Versions are contained in a History (*e.g.,* MethodHistory). A History does not have direct relation with a Snapshot entity, but through a Version wrapper as shown in Figure 3.1 (p.31). We create Versions as wrappers for SnapshotEntities because in a Version we store the relationship to the History: a Version is aware of the containing History and of its position in the History (*i.e.,* it knows the predecessor and the successor). Thus, we are able to compute properties for a particular Version in the context of the History. For example, having a Version we can navigate to the previous or the next Version.

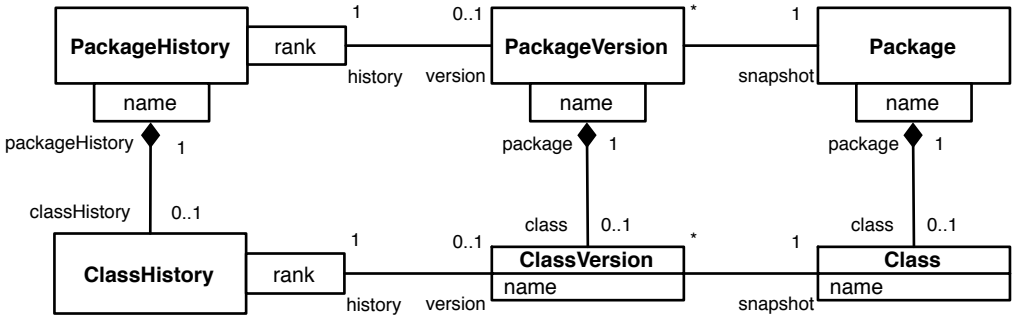


Figure 3.2: Hismo applied to Packages and Classes.

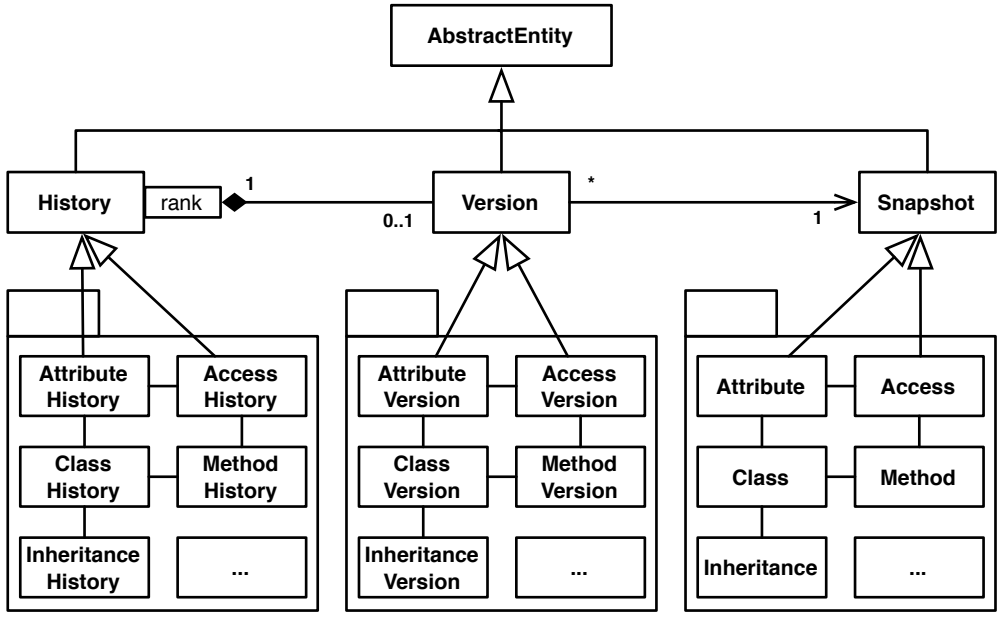


Figure 3.3: An excerpt of Hismo as applied to FAMIX, and its relation with a snapshot meta-model: Every Snapshot (*e.g.,* Class) is wrapped by a corresponding Version (*e.g.,* ClassVersion), and a set of Versions forms a History (*e.g.,* ClassHistory). We did not represent all the inheritance and association relationships to not affect the readability of the picture.

The Entity Identity Problem

A problem raised in the literature is that of what we call *entity identity*: having two entities at two different moments of time, how do we know whether they are two versions of the same history. This problem can also be found in the literature under the name of *origin analysis* [Antoniol and Di Penta, 2004; Zou and Godfrey, 2003].

The most common way to recover the identity is by the name of the entity, that is, if we have two entities with the same name and the same type in two versions, then they are considered to be two versions of the same history. Of course, such approaches fail to recognize refactorings like renaming or moving. Various approaches have been proposed to solve this problem: using information retrieval techniques [Antoniol and Di Penta, 2004], using string matching or entities fingerprints [Zou and Godfrey, 2003].

In our definition, the history is a set of versions, therefore, it also encapsulates the entity identity. We did not specify the algorithm to be used when determining entity, because it is the responsibility of the implementation to determine how the identity is defined. For example, it is possible to first determine the histories based on names and then detect renaming refactorings and merge the histories that are detected as being renamed.

3.4 Mapping Hismo to the Evolution Matrix

In this section we describe how Hismo maps to the Evolution Matrix (see Figure 3.4 (p.35)). In the upper part of the figure we represent Hismo applied to Packages and Classes where a package contains several classes, while in the lower part we show two Evolution Matrices. As described in Section 2.2.3 (p.16), a row represents the evolution of an entity, a class in this case, and a column all the entities of one version – package in this case. As such, In Figure 3.4 (p.35) each cell in the matrix represents a ClassVersion and each column represents a PackageVersion.

In Hismo, a *history* is a sequence of *versions*, thus, each line in an Evolution Matrix represents a ClassHistory (left matrix). Moreover, the whole matrix is actually a line formed by PackageVersions (right matrix), which means that the whole matrix can be seen as a PackageHistory (left matrix).

In the upper part we also represent the relations we have between the entities. On

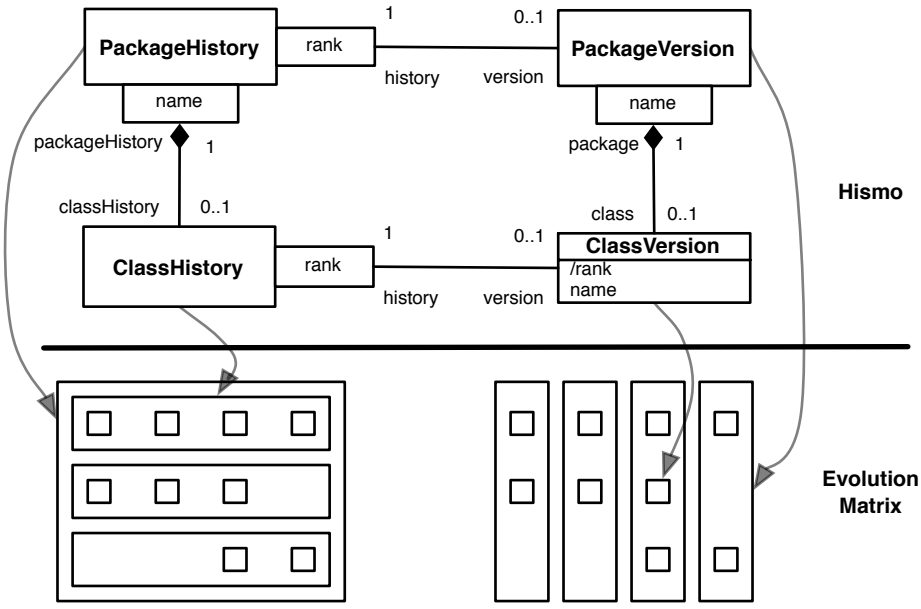


Figure 3.4: Mapping Hismo to the Evolution Matrix. Each cell in the Evolution Matrix represents a version of a class. Each column represents the version of a package. Each line in the Evolution Matrix represents a history. The entire matrix displays the package history.

the right part we show that a PackageVersion *has* multiple ClassVersions, while on the left side we show that in a PackageHistory *there are* multiple ClassHistories.

3.5 History Properties

As discussed in Section 2.3.1 (p.19), history measurements quantify the changes in the history according to a particular interest. The benefit of the historical measurements is that we can understand what happened with an entity *without* a detailed look at each version – *i.e.,* the measurements summarize changes into numbers which are assigned to the corresponding histories.

The shortcoming with most of the existing measurements is that they do not take into account the semantic meaning of the system structure, but they typically rely on primary data like lines of code, files and folders. Such measurements are of

limited use when we need fine grained information.

Figure 3.5 (p.38) gives an example how we can use the detailed information in Hismo to define historical measurements:

*Evolution of a property P (*EP*).* This measurement is defined both for a Version and for a History (H). For a Version, it is defined as the absolute difference of P with the previous version:

$$(i > 1) \quad EP_i(H, P) = |P_i(H) - P_{i-1}(H)| \tag{3.1}$$

For a History it is defined as the sum of the absolute difference of P in subsequent versions. This measurement can be used as an overall indicator of change.

$$(1 \leqslant j < k \leqslant n) \quad EP_{j..k}(H, P) = \sum_{i=j+1}^{k} EP_i(H, P) \tag{3.2}$$

*Latest Evolution of P (*LEP*).* While *EP* treats each change the same, with *LEP* we focus on the latest changes by a weighting function $2^{i-maxRank}$ which decreases the importance of a change as the version i in which it occurs is more distant from the latest considered version $maxRank$.

$$(1 \leqslant j < k \leqslant n) \quad LEP_{j..k}(H, P) = \sum_{i=j+1}^{k} EP_i(H, P)2^{i-k} \tag{3.3}$$

*Earliest Evolution of P (*EEP*).* It is similar to *LEP*, only that it emphasizes the early changes by a weighting function $2^{i-minRank}$ which decreases the importance of a change as the version i in which it occurs is more distant from the first considered version $minRank$.

$$(1 \leqslant j < k \leqslant n) \quad EEP_{j..k}(H, P) = \sum_{i=j+1}^{k} EP_i(H, P)2^{k-i+1} \tag{3.4}$$

Given a History we can obtain a sub-History based on a filtering predicate applied on the versions. Therefore, whichever properties we can compute on Histories, we

can also compute on the sub-Histories.

In Figure 3.6 (p.38) we show an example of applying the defined history measurements to 5 histories of 5 versions each.

— During the displayed history of D (5 versions) P remained 2. That is the reason why all three history measurements were 0.

— Throughout the histories of class A, of class B and of class E the P property was changed the same as shown by the Evolution of P (EP = 7). The Latest and the Earliest Evolution of P (LEP and EEP) values differ for the three class histories which means that (i) the changes are more recent in the history of class B (ii) the changes happened in the early past in the history of class E and (iii) in the history of class A the changes were scattered through the history more evenly.

— The histories of class C and E have almost the same LEP value, because of the similar amount of changes in their recent history. The EP values differ heavily because class E was changed more throughout its history than class C.

The above measurements depend on the P property. For example, P can be the number of methods of a class (NOM), or the number of lines of code of a method (LOC). As a consequence, in the case of EP we talk about ENOM, when P is NOM, or about ELOC when P is LOC. We use the above measurements in Chapter 4 (p.47) and in Chapter 6 (p.87).

In a similar fashion, we define other measurements. Here is a non-exhaustive list:

Age. It counts the number of versions in the history. We use this measurement in Chapter 6 (p.87).

Additions of P / Removals of P. These measurements sum the additions or removals of a property P. Additions are a sign of increase in functionality, while removals are a sign of refactoring. We use the Additions of P measurement in Chapter 8 (p.125).

Number of Changes of P. It counts in how many versions the property P changed with respect to the previous version.

Stability of P. It divides the Number of Changes of P by the number of versions - 1 (*i.e.*, the number of versions in which P could have changed). We use this measurement in Chapter 5 (p.67).

History Maximum / Minimum / Average. These measures the maximum, mini-

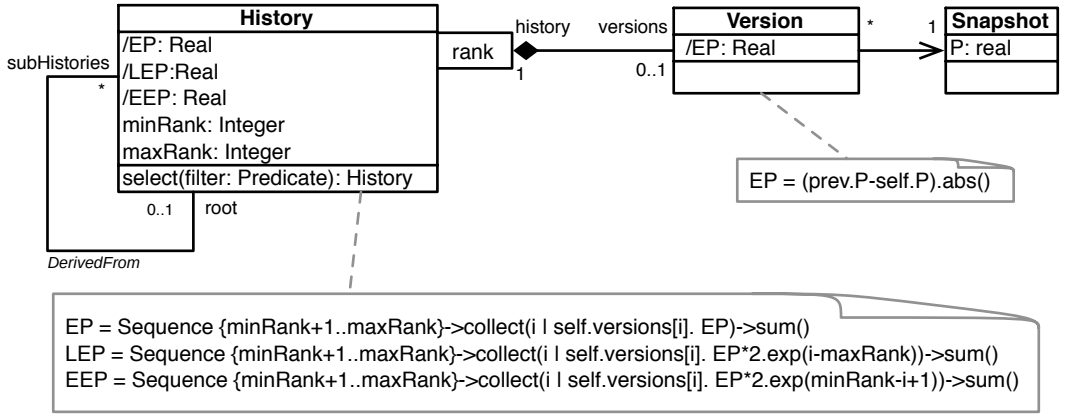


Figure 3.5: Examples of history measurements and how they are computed based on structural measurements.

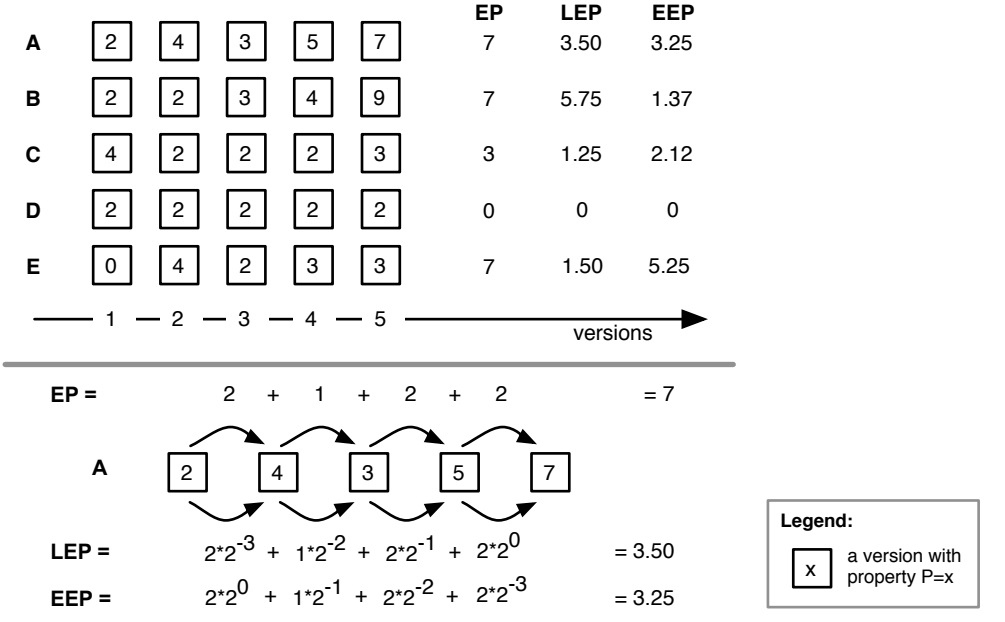


Figure 3.6: Examples of EP, LEP and EEP history measurements. The top part shows the measurements computed for 5 histories. The bottom part shows the computation details.

mum or the average value of P over the versions. We use these measurements in Chapter 6 (p.87).

Persistence of a version Condition. It counts the number of versions in which the Condition is true over the total number of versions. We use this measurement in Chapter 5 (p.67).

Beisdes measurements, other historical properties can be defined. Here are some examples of boolean properties [Lanza and Ducasse, 2002]:

Persistent. A persistent entity is an entity that was present in all versions of the system. We use this property in Chapter 6 (p.87).

Removed. A history is removed if its last version is not part of the system history's last version. We use this property in Chapter 6 (p.87).

Day-fly. Day-fly denotes a history that is Removed and that is 1 version old.

The above measurements and properties do not make use of the actual time information from the versions. For example, Age is measured in number of versions, while it could be measured in the actual time spanned from the beginning until the end of the history. In the context of this dissertation we use the measurements and properties as described above, however we believe there is great potential in considering the actual time dimension.

3.6 Grouping Histories

One important feature when analyzing entities is grouping. It is important because we need to be able to reason about a collection of entities like we reason about individual entities. For example, having at hand a group of classes, we would like to identify the root class of the classes in the group, or to get the average of a property P.

In the same way, we want to be able to manipulate a group of histories. For example, we would like to know which are the top three most changed class histories in the latest period.

Figure 3.7 (p.40) shows the UML diagram of our model in which an AbstractGroup, Group and HistoryGroup are first class entities. In general, we can ask an AbstractGroup for the average of a numerical property P, or we can select the entities with the highest values of a property P.

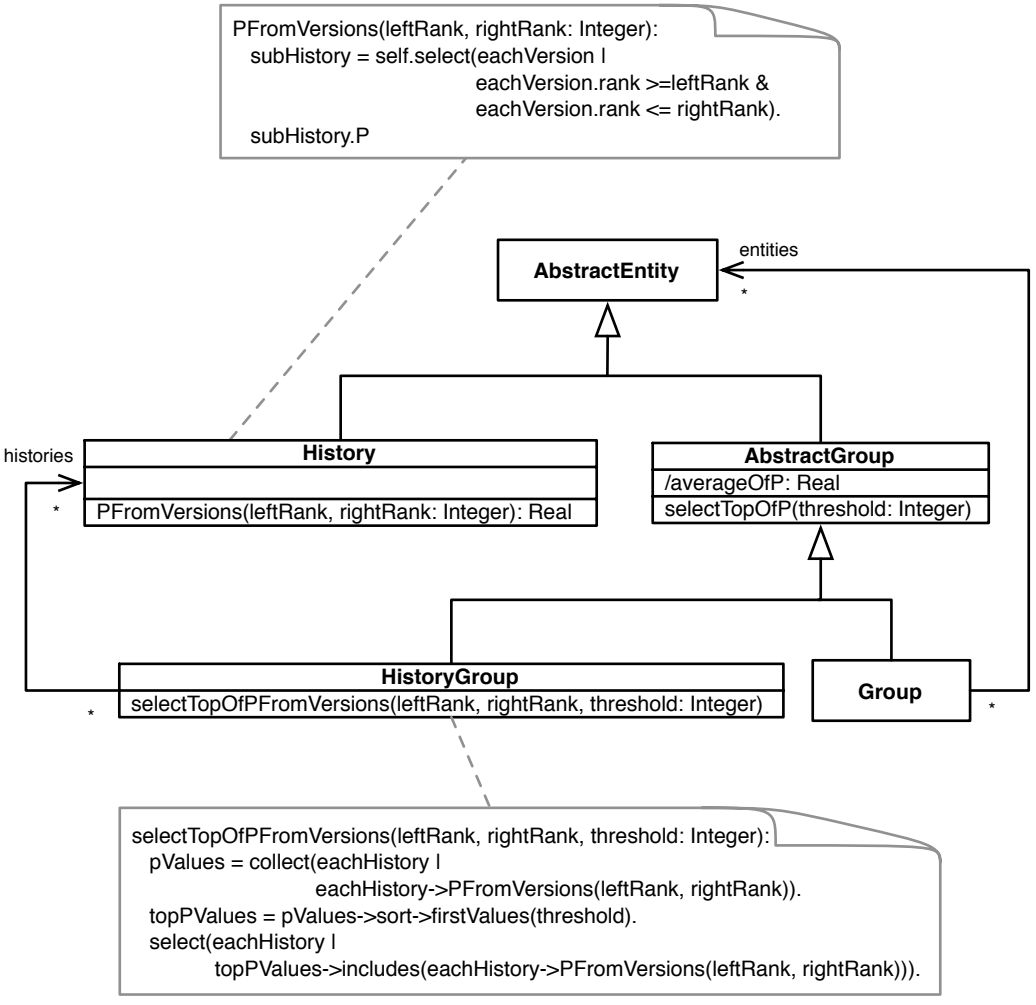


Figure 3.7: HistoryGroup as a first class entity.

A HistoryGroup is a Group that holds histories instead of generic entities and it defines specific queries. In the figure, we give the example of selecting the histories that have the highest historical property P computed only on a selection of histories based on specified versions. This particular query is used in Chapter 4 (p.47).

3.7 Modeling Historical Relationships

In this section we detail how we model relationships between histories.

Modeling Historical Relationships Recovered From Explicit Snapshot Relationships

In the previous sections we showed how to model the history of a snapshot entity like classes or packages. But, in a snapshot meta-model we can model explicitly structural relationships like inheritance or invocations. Having the relationship as a first class entity, allows us to model the corresponding history in exactly the same fashion as we did for structural entities.

Figure 3.8 (p.42) shows an example of how to model inheritance relationships. Starting from "an InheritanceSnapshot links two Classes", we build the version representation "an InheritanceVersion links two ClassVersions" and the history representation "an InheritanceHistory links two ClassHistories." In Chapter 6 (p.87) we give an example of how we can use this meta-model for understanding how hierarchies as a whole evolved.

Modeling Historical Relationships Recovered From Evolution

We can infer historical relationships not only from explicit structural relationships, but from how different entities evolved. In Section 2.3.2 (p.21) we reviewed a number of evolution analyses based on the assumption that two entities are related if they are changed in the same time. We dedicate this section in showing how we model co-change as explicit historical relationships.

Two versions of two entities are related through co-change relationship if they are both changed with respect to their respective previous versions. In the same

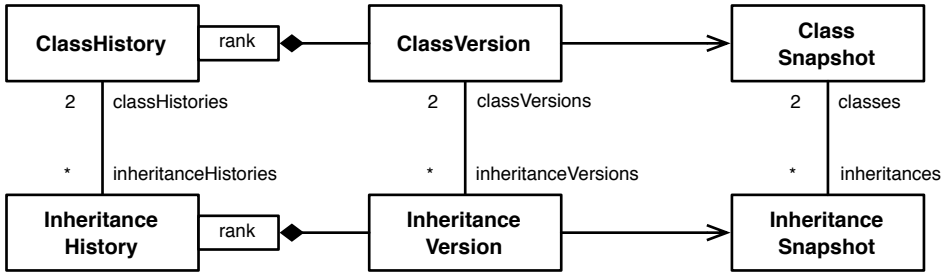


Figure 3.8: Using Hismo for modeling historical relationships.

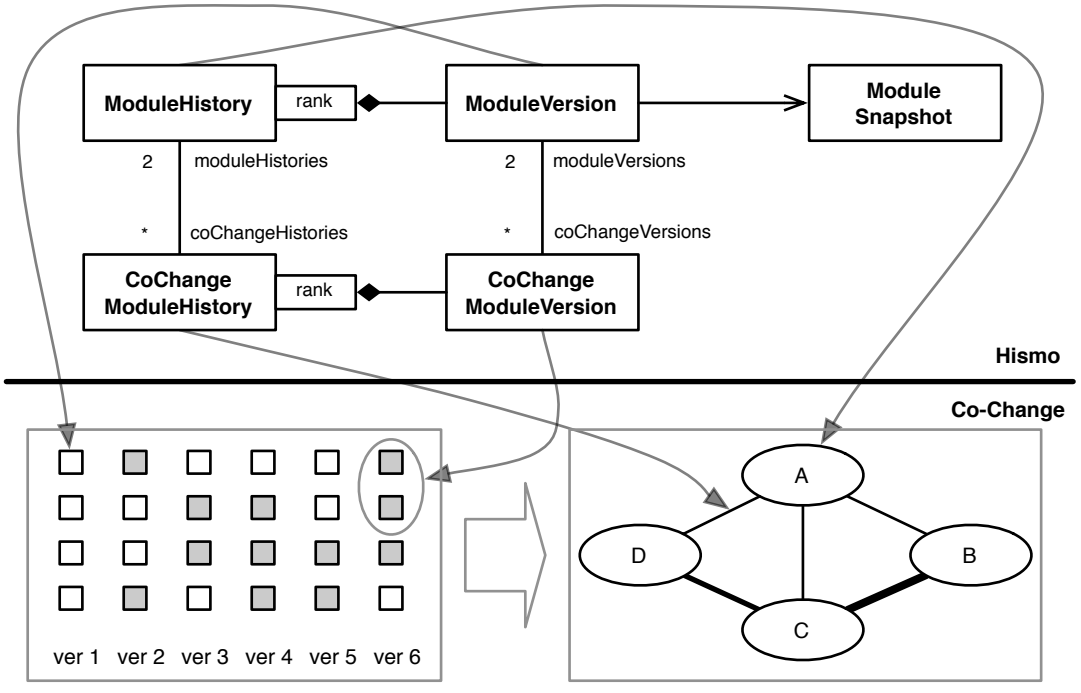


Figure 3.9: Using Hismo for co-change analysis. On the bottom-left side, we show 6 versions of 4 modules: a grayed box represent a module that has been changed, while a white one represents a module that was not changed. On the bottom-right side, we show the result of the evolution of the 4 modules as in Figure 2.3 (p.22).

line, we define a CoChangeHistory to represent the relationship between two histories.

Figure 3.9 (p.42) shows an example of how to model co-changes between modules. On the bottom part we represent the example from Figure 2.3 (p.22): On the bottom-left side we show 6 versions of 4 modules (A, B, C, D) and how they changed from one version to the other (a change is marked with gray). On the bottom-right we show the historical representation. The resulting picture is the same as in Figure 2.3 (p.22) only the meta-model is different. We no longer represent the Modules as ellipses, but ModuleHistories, and the co-change is an explicit historical relationship (*i.e.,* CoChangeModuleHistory).

From the historical point of view, co-change and inheritance are similar, as they both transform into relationships between histories. The only difference between them is that in the case of co-change relationship there is no snapshot entity wrapped by the CoChangeVersion.

3.8 Generalization: Transforming Snapshot Meta-Models into History Meta-Models

In the previous sections we gave examples of how to build the history meta-model based on a snapshot meta-model. In this section, we generalize the approach and show how we can use meta-model transformations for obtaining the history meta-model.

In Figure 3.10 (p.44) we show in details the transformation which generates from a Class entity in the snapshot meta-model the corresponding ClassHistory and ClassVersion entities. Thus, a ClassHistory is a sequence of ClassVersions. Also the model allows us to define history properties based on structural properties.

The bold text in the figure shows how the result only depends on the Snapshot and its properties. For example, having the number of Lines of Code (LOC) as an attribute in a Class, we can derive the minimum or the maximum lines of code in the history. In the figure we show how we derive the Evolution of Lines of Code, as the sum of the absolute differences of the lines of code in subsequent versions. The history properties obtained in this manner, characterize and summarize the evolution.

Figure 3.11 (p.44) shows how we can obtain the relationships between the meta-

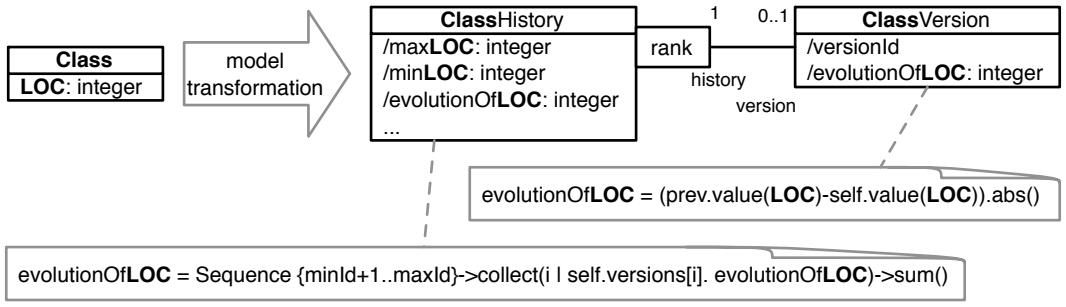


Figure 3.10: Transforming the Snapshot to obtain corresponding History and Version and deriving historical properties.

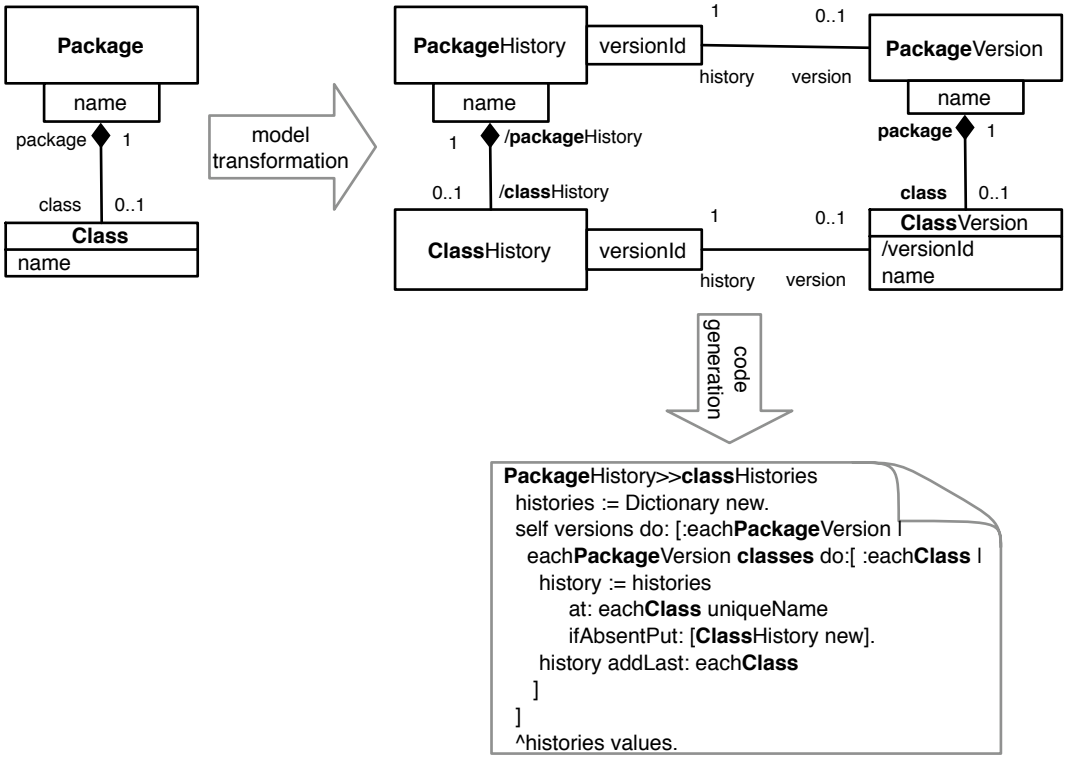


Figure 3.11: Obtaining the relationships between histories by transforming the snapshot meta-model.

model entities starting from the structural relationships. On the left side of the figure we have a Package containing multiple Classes. After the transformation we have the PackageHistory as containing multiple ClassHistories. On the down-right side of the figure we see the generated code in Smalltalk for obtaining the ClassHistories starting from a PackageHistory. Similarly to Figure 3.10 (p.44), the bold text in the figure shows how the algorithm only depends on the Snapshots and their relationships.

3.9 Summary

In this chapter we introduced Hismo, our meta-model which explicitly models history as a collection of versions. In Hismo, the historical perspective is added on top of the snapshot perspective. As a result, we can reuse the analyses built at structural level in the historical context.

We gave examples of different historical properties, and we gave evidence of how such measurements can be used to characterize histories. We introduced the notion of grouping histories and we showed how we can define queries that take into account the historical properties. We completed the meta-model picture by showing how we build historical relationships based both on snapshot relationships (*e.g.,* inheritance history) and on evolution (*e.g.,* co-change). Furthermore, we generalized our approach, by showing how the historical meta-model can be obtained by transforming the snapshot meta-model.

The next chapters discuss several evolution analyses, each one exercising a different part of the meta-model (see Section 2.5 (p.27)). Each chapter has a similar structure: first we discuss in detail the approach to show its relevance for reverse engineering, and in the end we show how Hismo supports the expression of the approach.

Chapter 4

Yesterday's Weather

Selecting Histories and Combining Historical Properties

Yesterday is a fact.
Tomorrow is a possibility.
Today is a challenge.

Knowing where to start reverse engineering a large software system, when no information other than the system's source code itself is available, is a daunting task. Having the history of the code could be of help if this would not imply analyzing a huge amount of data. In this chapter we present an approach for identifying candidate parts of the system for reverse engineering and reengineering efforts. Our solution is based on summarizing the changes in the evolution of object-oriented software systems by defining history measurements. Our approach, named Yesterday's Weather*, is a measurement based on the retrospective empirical observation that the parts of the system which changed the most in the recent past also suffer important changes in the near future. We apply this approach on three case studies with different characteristics and show how we can obtain an overview of the evolution of a system and pinpoint the parts that might change in the next versions.*

Yesterday's Weather *is an example of how to build complex historical measurements by combining historical properties computed on selections of histories.*

4.1 Introduction

When starting a reverse engineering effort, knowing where to start is a key question. When only the code of the application is available, the history of a software system could be helpful as it holds valuable information about the life of the system, its growth, decay, refactoring operations, and bug-fixing phases. However, analyzing a software system's history is difficult due to the large amount of complex data that needs to be interpreted. Therefore, history analysis requires one to create higher level views of the data.

The basic assumption of this dissertation is that the parts of the system that need to change are those that need to be understood first. We can find out about the tendencies of changes by looking at the past ones. However, not every change in the history of the system is relevant for the future changes. For example, the parts of a system which changed in its early versions are not necessarily *important*[1] for the near future: Mens and Demeyer suggested that the evolution-prone parts of a system are those which have changed a lot recently [Mens and Demeyer, 2001].

We aim to measure how relevant it is to start reverse engineering the parts of the system which changed the most in the recent past. Based on historical information, we identify the parts of the system that changed the most in the recent past and check the assumption that they are likely to be among the most changed ones in the near future. If this assumption held many times in the system history, then the recently changed parts are good candidates for reverse engineering. Our experiments show that important changes do not necessarily imply that they only occur in the largest parts (*e.g.,* in terms of lines of code). Therefore identifying the big parts in the last version of a software system is not necessarily a good indicator for future changes.

We concentrate on the evolution of object-oriented systems, where by the parts of a system we understand structural entities like packages, classes or methods. We identify the parts that are likely to change by defining evolutionary measurements that summarize the history of those parts. We show the relevance of these measurements in our approach which we name *Yesterday's Weather*. Our approach is similar to the historical observation of the weather: a way of guessing what the weather will be like today is to think it will stay the same as yesterday. This heuristic can have very high success rates, however, its success is not the same in all places: In the Sahara desert the chance that the weather stays the same from

[1] By *important* we denote the fact that these parts will be affected by changes.

one day to the next is higher than in Belgium, where the weather can change in a few hours. Therefore, to use such a heuristic for "successful weather forecasts", we need to know how relevant the heuristic is for the place we want to use it. We obtain the relevancy by analyzing the historical information about the climate of the place we are interested in.

Yesterday's Weather is a measurement applied on a system history and it characterizes the "climate" of a software system. More specifically, *Yesterday's Weather* provides an indication that allows one to evaluate the relevance of starting reverse engineering from the classes that changed the most recently.

Structure of the Chapter

We start by presenting an overview of *Yesterday's Weather* in Section 4.2 (p.49). In Section 4.3 (p.50) we go into the details of computing and interpreting *Yesterday's Weather*. In Section 4.4 (p.54) we present the results obtained on three case studies and then discuss the variation points of our approach. In Section 4.5 (p.60) we discuss different variation points of our approach. Section 4.6 (p.63) presents the related work. In Section 4.7 (p.64) we summarize the approach, and we conclude the chapter with a discussion on how the usage of Hismo makes the expression of *Yesterday's Weather* simple (Section 4.8 (p.65)).

4.2 Yesterday's Weather in a Nutshell

We define *Yesterday's Weather* (*YW*) to be the retrospective empirical observation of the phenomenon that at least one of the heavily changed parts in the recent history is also among the most changed parts in the near future.

Our approach consists in identifying, for each version of a subject system, the parts that were changed the most in the recent history and in checking if these are also among the most changed parts in the successive versions. We count the number of versions in which this assumption holds and divide it by the total number of analyzed versions to obtain the value of *Yesterday's Weather*.

Example. Suppose that we want to analyze how classes change and for a system *YW* yields a value of 50%. This means that the history of the system has shown that in 50% of the cases at least one of the classes that was changed a

lot in the recent past would also be among the most changed classes in the near future.

YW characterizes the history of a system and is useful from a reverse engineering point of view to identify parts that are likely to change in the next version. On one hand we use such information to make out progressive development phases in the evolution of the system (*e.g.,* what is/was the current focus of development?). In phases where the developers concentrate on a certain part of the system, the evolution would be fairly easy to predict. During repairing phases due to bug reports or little fixes the developers change the software system apparently at random places which leads to a decrease of predictability (*i.e.,* the weather becomes unstable). On the other hand it also gives a general impression of the system (*i.e.,* how stable is the climate of the whole system?). By interpreting the *YW* we identify that the changes are either focused on some parts over a certain period of time, or they move unpredictably from one place to another.

Example. If the *YW* value of a software system S_1 is 10%, this implies that the changes in the system were rather discontinuous – maybe due to new development or bug fix phases. If the *YW* yields an 80% value for a system S_2, this implies the changes in the system were continuous. In such a system, we say it is relevant to start the reverse engineering from the classes which were heavily changed lately, while this is not the case in system S_1.

4.3 Yesterday's Weather in Detail

Before defining the *YW* function, we introduce the notion of *top n* of entities out of an original set *S* of entities with the highest *P* property value:

$$(0 < n < 1) \quad Top_P(S, n) = S' \left| \begin{array}{l} S' \subseteq S, \\ |S'| = n \\ \forall x \in S', \forall y \in S - S' \\ P(x) > P(y) \end{array} \right. \tag{4.1}$$

To check the *Yesterday's Weather* assumption with respect to classes, for a system version i, we compare the set of class histories with the highest $LENOM_{1..i}$ values (the *candidates* set) with the set of the class histories with the highest $EENOM_{i..n}$ values (the *really-changed* set). The *Yesterday's Weather* assumption holds if the intersection of these sets is not empty, that is at least one class history belongs

to both sets. This means that for the classes in version i at least one of the recently most changed classes is among the most changed classes in the near future relative to version i. If the assumption holds for version i we have a hit (as shown in Equation 4.2 (p.51) and Figure 4.1 (p.52)).

In general, we formally define the *Yesterday's Weather* hit function at version i applied to a set of histories S with respect to a property P and given two threshold values t_1 and t_2 as follows:

$$(i > 1; t_1, t_2 \geq 1)$$
$$YW_i(S, P, t_1, t_2) = \begin{cases} 1, & Top_{LEP_{1..i}}(S, t1) \cap Top_{EEP_{i..n}}(S, t2) \neq \emptyset \\ 0, & Top_{LEP_{1..i}}(S, t1) \cap Top_{EEP_{i..n}}(S, t2) = \emptyset \end{cases} \tag{4.2}$$

The overall *Yesterday's Weather* is computed by counting the hits for all versions and dividing them by the total number of possible hits. Thus, we obtain the result as a percentage with values between 0% and 100%.

We formally define the *Yesterday's Weather* applied to n versions of a set of histories S with respect to a property P and given two threshold values t_1 and t_2 as in Equation 4.3 (p.51).

$$(n > 2; t_1, t_2 \geq 1)$$
$$YW_{1..n}(S, P, t_1, t_2) = \frac{\sum_{i=2}^{n-1} YW_i(S, P, t_1, t_2)}{n - 2} \tag{4.3}$$

Example. In Figure 4.1 (p.52) we present an example of how we check *Yesterday's Weather* with respect to a certain version. We display 6 versions of a system with 7 classes (A-G). We want to check *Yesterday's Weather* when considering the 4th version to be the present one. Therefore, the versions between 1 to 3 are the past versions, and the 5th and 6th are the future ones.

We also consider the dimensions of the *candidates* and the *really-changed* set to be 3, that is, we want to check the assumption that at least one of the top three classes which were changed in the recent history will also be among the top three most changed classes in the near future. We draw circles with a continuous line around the A, C and F classes to mark them as being the top three classes which changed the most in the recent history with respect to the 4th version. A, C and F are candidates for a *Yesterday's Weather* hit: While B changed recently, it is not

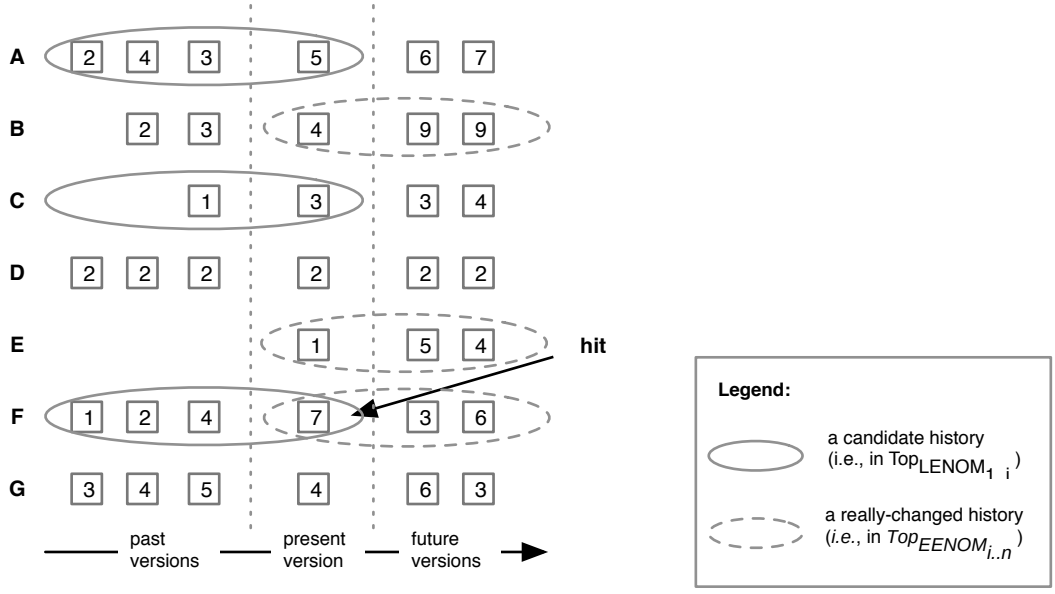


Figure 4.1: The detection of a *Yesterday's Weather* hit with respect to classes.

a candidate because A, C and F changed more than B and in this case we only concentrate on the top three most changed classes. We marked with a dotted circle the classes which change the most in the next versions after the 4th one. We get a hit if the intersection between the continuous line circles and the dotted circles is not empty. In the presented example we get a hit because of class F.

In Figure 4.2 (p.53) we show how we compute the overall *Yesterday's Weather* for a system history with 10 versions. The grayed figures show that we had a hit in that particular version, while the white ones show we did not. In the example we have 6 hits out of possible 10, making the value of *Yesterday's Weather* to be 60%.

Yesterday's Weather Interpretation

Suppose we have to analyze a system history with 40 versions, where each version consists on average of 400 classes, and suppose we compute *YW(*S_1*,NOM,20,20)* and get a result of 10%: The "climate" of the system is unpredictable with respect to the important changes. In such a "climate" it is not relevant to consider the

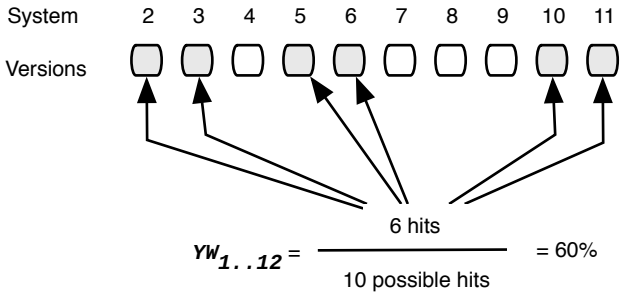


Figure 4.2: The computation of the overall *Yesterday's Weather*.

latest changed classes to be important for the next versions.

If, on the other hand, we compute the *YW(*S_2*,NOM,5,5)* and get a result of 80%, it means that during the analyzed period in 80% of the versions at least one of the 5 classes that changed the most in the recent past is among the 5 classes that change the most in the near future. Therefore, we have a great chance to find, among the first 5 classes which were heavily changed recently, at least one class which would be important (from a reverse engineering point of view) for the next version.

The value of *YW* depends on the dimensions of the sets we want to compare. For example, on each line in the table in Figure 4.3 (p.56) we display different results we obtained, on the same history, depending on the sizes of the sets. For example, for 40 versions of Jun, when we considered the *LENOM*$_{1..i}$ set size to be 5 and the *EENOM*$_{i..n}$ set size to be 5, the *YW* was 50% (*i.e., YW(*Jun*, NOM,5,5)* = 50%). When the *LENOM*$_{1..i}$ set size was 10 and the *EENOM*$_{i..n}$ set size was 10, the *YW* was 79% (*YW(*Jun*, NOM,10,10)* = 79%).

In *YW(S, candidates, really-changed)* the dimensions of the *candidates* and *really-changed* sets represent thresholds that can be changed to reduce or enlarge the scope of the analysis. Thus, using higher thresholds increases the chance of a hit but also increases the scope, while by using lower tresholds we reduce the scope, but we also reduce the probability to have a hit. Both thresholds have specific interpretations:

1. The *candidates* set threshold represents the number of the parts which changed the most in the recent past. The lower this threshold is the more accurate the assumption is. For example, imagine that for one system we choose a *LE*

threshold of 1 and an *EE* threshold of 5 and we get a *YW* value of 60% (*i.e.*, *YW*(S_1,*P*,***1***,*5)* =60%). For another similar system we get *YW*(S_2,*P*,***3***,*5)* =60%. It means that in the first system you have a 60% chance that the part identified as changing the most in the recent past will be among the 5 parts that change the most in the near future, while in the second system, we have to investigate three candidate parts to have a 60% chance of finding one important part for the near future.

2. The size of the *really-changed* set is the second threshold and it shows how important – *i.e.*, how affected by changes – the candidates are. The lower this threshold is, the more important the candidates are. Suppose we have *YW*(S_1,*P*,*5*,***5)*** = 60% in one system and *YW*(S_2,*P*,*5*,***1)*** = 60% in another system. It means that in the first system we have a 60% chance that one candidate will be *among* the first 5 important parts in the next versions, while in the second system we have a 60% chance that one of the candidates will be *the most* important part in the next version.

4.4 Validation

Our approach measures the relevance of starting reverse engineering from the latest changed parts of the system. As a validation, we compute our approach on three available case studies with different characteristics (Jun, CodeCrawler and JBoss), and discuss the results from the threshold values and the history sample. We also compare the results obtained by *YW* against the size of different parts of the system and conclude that big size is not a reliable indicator for the future changes.

Jun[2] is a 3D-graphics framework currently consisting of more than 700 classes written in Smalltalk. The project started in 1996 and is still under development. As experimental data we took every 5th version starting from version 5 (the first public version) to version 200. The time distance between version 5 and version 200 is about two years, and the considered versions were released about 15-20 days apart. In the first analyzed version there were 160 classes, in the last analyzed version there were 740 classes. In total there were 814 different classes which were present in the system over this part of its history, and there were 2397 methods added or removed.

[2]See http://www.srainc.com/Jun/.

CodeCrawler[3] is a language independent reverse engineering tool which combines metrics and software visualization. In the first analyzed version there were 92 classes and 591 methods, while in the last analyzed version there were 187 classes and 1392 methods. In the considered history, there were 298 different classes present in the system over the considered history and 1034 methods added or removed in subsequent versions.

JBoss[4] is an open source J2EE application server written in Java. The versions we selected for the experiments were at two weeks distance from one another starting from the beginning of 2001 until 2003. The first version has 632 classes and 102 packages, the last one has 4015 classes and 509 packages.

We chose these case studies because of their differences. Jun and JBoss have been developed by a team of developers while CodeCrawler is a single developer project. Furthermore, Jun and CodeCrawler are written in Smalltalk, while JBoss is written in Java.

4.4.1 Yesterday's Weather in Jun, CodeCrawler and JBoss

Figure 4.3 (p.56) presents the results of the *YW* for the case studies for different number of versions while keeping the thresholds constant. High values (*e.g.,* 79% for Jun or more than 90% for CodeCrawler) denote a stable climate of the case studies: the changes either went slowly from one part to another of the system, or the changes were concentrated into some classes.

When we choose more distance between releases, we take into consideration the accumulation of changes between the releases: the candidate parts are not necessarily heavily changed in just one version, but they could be changed over more versions.

Jun. When we doubled the thresholds when analyzing 40 versions of Jun, we obtained 29% more in the *YW* value (*i.e., YW(*Jun_{40}*, NOM, 5, 5)* = 50% becomes *YW(*Jun_{40}*, NOM, 10, 10)* = 79%). Moreover, when we doubled the thresholds when analyzing 10 versions, we more than doubled the *YW* value (*i.e., YW(*Jun_{10}*, NOM, 5, 5)* = 37% becomes *YW(*Jun_{10}*, NOM, 10, 10)* = 87%). These facts show that in Jun there were classes which were changed over a long period of time, but these changes are not identified when we analyze versions which are closer to each other.

[3] See http://www.iam.unibe.ch/ scg/Research/CodeCrawler/.
[4] See http://sourceforge.net/projects/jboss/.

History sample	YW(3,3)	YW(5,5)	YW(10,10)
40 versions of Jun (Jun_{40})	40%	50%	79%
20 versions of Jun (Jun_{20})	39%	55%	77%
10 versions of Jun (Jun_{10})	37%	37%	87%
40 versions of CodeCrawler (CC_{40})	68%	92%	100%
20 versions of CodeCrawler (CC_{20})	61%	94%	100%
10 versions of CodeCrawler (CC_{10})	62%	100%	100%
40 versions of JBoss ($JBoss_{40}$)	11%	26%	53%
20 versions of JBoss ($JBoss_{20}$)	28%	38%	67%
10 versions of JBoss ($JBoss_{10}$)	50%	63%	63%

Figure 4.3: *YW* computed on *classes* with respect to methods on different sets of versions of Jun, CodeCrawler and JBoss and different threshold values.

To show the relevance of *YW* we display the class histories that provoked a hit when computing *YW*(Jun_{40},*NOM,10,10)* for 40 versions of Jun (see Figure 4.4 (p.57)). We focused on the size of the classes in terms of number of methods and determined that *YW* predicts changes in classes which are not necessarily big classes (*e.g.,* JunOpenGLPerspective). We grayed the classes which provoked a hit when computing *YW* and which were not in the top 10 of the biggest classes in the last version. 17 out of 22 classes are not in the first 10 classes in terms of number of method: in Jun a big class is not necessarily an important class in terms of future changes.

CodeCrawler. CodeCrawler is a project developed mainly by one developer, and as such can be considered a system with a focused and guided development with little external factors. This assumption is backed up by the data which reveals very high *YW* values for low thresholds, resulting in a "stable climate" of the system. Note that CodeCrawler is much smaller than Jun, the thresholds must thus be seen as relatively lax.

Figure 4.5 (p.58) displays, using the same notation as in Figure 4.4 (p.57), the class histories that provoked a hit in *YW(5,5)*. As in the case of Jun, the hits were not necessarily provoked by big classes and not all big classes provoked a hit. This

YW(Jun_{40}, NOM, 10,10) Hit Classes	NOM	
JunOpenGLDisplayModel	150	In top 10 biggest classes in the last version
JunWin32Interface	104	
JunBody	85	
JunOpenGL3dObject	75	
JunOpenGL3dObject_class	71	
JunOpenGL3dNurbsSurface	55	**NOT** in top 10 biggest classes in the last version
JunLoop	55	
Jun3dLine	51	
JunOpenGLProjection	48	
JunUNION	47	
JunOpenGL3dCompoundObject	41	
JunPolygon	34	
JunBody_class	31	
JunVertex	30	
JunOpenGL3dVertexesObject	23	
JunOpenGL3dCompoundObject_class	21	
JunOpenGL3dVertex	19	
JunUNION_class	19	
JunOpenGL3dPolygon	15	
JunOpenGLPerspective	12	
JunOpenGLTestController	9	
JunOpenGLTestView	1	

Figure 4.4: The class histories that provoked a hit when computing *YW(*Jun_{40}*,NOM,10,10)* and their number of methods in their last version. In this case study, the big classes are not necessarily relevant for the future changes.

YW(CC_{40}, NOM, 5,5) Hit Classes	NOM	
CCDrawing	123	In top 10 biggest classes in the last version
CCAbstractGraph	99	
CCGraph	69	
CCNode	47	
CodeCrawler	42	
CCConstants_class	39	
CCEdge	36	**NOT i**n top 10 biggest classes in the last version
CCControlPanel	29	
CCGroupNodePlugIn	25	
CCModelSelector	24	
CCRepositorySubcanvas	17	
CodeCrawler_class	15	
CCService	0	

Figure 4.5: The class histories that provoked a hit when computing *YW(CC_{40},NOM,5,5)* and their number of methods in their last version. In this case study, the big classes are not necessarily relevant for the future changes.

shows that in CodeCrawler there is not always a relationship between changes and size. Therefore, identifying the big classes from the last version, is not necessarily a good indicator for detecting classes which are important, in terms of change, for the next versions.

JBoss. JBoss is a significantly larger system than the Jun. When we applied *YW* on the classes of JBoss, we did not obtain a high value: In JBoss it is less relevant to start reverse engineering from the latest changed classes.

The highest values were obtained when analyzing versions 2 months apart from each other (*i.e.,* 10 versions in total). This is an indicator that there are classes that change over a longer period of time, and that these changes do not show when analyzing more finer grained versions.

We applied *YW* on the packages of JBoss with respect to methods and we displayed the results in Figure 4.6 (p.59). In Figure 4.7 (p.59) we show the packages that provoked a hit. In this case too, we see that the hits are not only provoked by the largest packages.

History sample	YW(3,3)	YW(5,5)	YW(10,10)
40 versions of JBoss ($JBoss_{40}$)	21%	45%	82%
20 versions of JBoss ($JBoss_{20}$)	17%	50%	100%
10 versions of JBoss ($JBoss_{10}$)	13%	63%	100%

Figure 4.6: *YW* computed on *packages* with respect to the total number of methods on different sets of versions of JBoss and different threshold values.

YW($JBoss_{40}$,NOM,10,10) Hit Packages	NOM	
org::jboss::util	715	In top 10 biggest packages in the last version
org::jboss::ejb	513	
org::jboss::management::j2ee	436	
org::jboss::ejb::plugins::cmp::jdbc	396	
org::jboss::security	172	**NOT** in top 10 biggest packages in the last version
org::jboss::system	140	
org::jboss::naming	100	

Figure 4.7: The package histories provoking a hit when computing *YW($JBoss_{40}$,NOM,10,10)* and their number of methods in their last version. In this case study, the big packages are not necessarily relevant for the future changes.

In all three considered case studies we detected that the size of different parts of the systems is not necessarily a good indicator for predicting future changes.

4.4.2 The Evolution of Yesterday's Weather in Jun

In Figure 4.8 (p.60) we represent a chart which shows Jun's evolution of *Yesterday's Weather* over time. The points in the chart show the value of *Yesterday's Weather* until that version: in version 15 *Yesterday's Weather* is 100%, drops in version 25, grows again until version 100 and then finally has an oscillating descending shape.

Based on this view we can detect phases in the evolution where the changes were focused and followed by other changes in the same part of the system (the ascending trends in the graph) and phases where the changes were rather unfocused (the descending trends in the graph). In the first half of the analyzed versions, in 90%

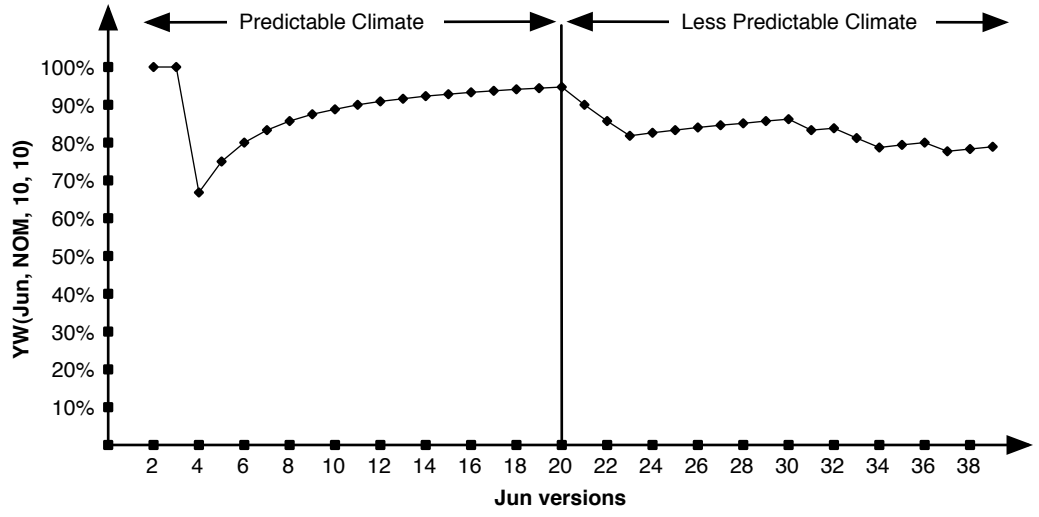


Figure 4.8: The evolution of the values of $YW(Jun_{40},NOM,10,10)$ when applied to classes. The diagram reveals phases in which the predictability increases and during which changes are more focused (*e.g.*, the first part of the history) and phases in which the predictability decreases and changes are more unfocused (*e.g.*, the second part of the history).

of the cases at least one class which was in the top 10 of the most changed classes in the last period was also in the top 10 of the most changed classes in the next version. In the last 20 versions that we analyzed, the probability drops. Therefore, in the first half of the analyzed period the development was more continuous and focused than in the second half.

4.5 Variation Points

In this section we explain the impact of the decisions we took when defining the *YW* measurement.

On the impact of the weighting function

The *LENOM* measure weighs each change using the function 2^{i-k} (see Section 3.5 (p.35)). This function actually acts like a window over the complete history of the

changes by considering as relevant only the last four versions. This window is important as it lowers the impact of early development. For example, if a big class was developed in the early versions but now suffers only bug-fixes, it will not be selected as a candidate for future important changes. Increasing the window size favors the candidacy of the large classes in the system, even if they are not changing anymore, and reduces the relevancy of the prediction. Note that although the value of *LENOM* takes into account only the last four versions, the *YW* measurement is computed over the complete history.

On the impact of the *at least* condition

With the current *YW* assumption we consider to have a hit if *at least one* part which was heavily changed recently also gets changed a lot in the next versions. If we have *YW(S,P,10,10)* = 60%, we do not know if the assumption held for 10 out of the 10 candidate part histories or just for one of them. *YW* gives relevant results in two cases:

1. *High value of* YW *when considering low thresholds.* Low thresholds (*e.g.,* 5) mean low scope (both of candidates or of the importance of the *really-changed* entities), and if for such low thresholds we obtain a high *YW* value, we can characterize the changes as being continuous, and therefore it is relevant to look at the most recently changed classes to detect one which will probably undergo an important change during the near future, *e.g.,* the next versions.

2. *Low value of* YW *when considering high thresholds.* When obtaining low *YW* values for high thresholds, we can characterize the changes as being discontinuous, and therefore looking at the most recently changed parts is not necessarily relevant for the future changes in the system.

In Figure 4.1 (p.52) the hit is provoked by one out of three candidates. Yet, in another version the hit could be provoked by more candidates. A possible enhancement of *Yesterday's Weather* would be to compute an average of the number of parts that matched the YW_i assumption. The result of this average would complement the *YW* value, by showing its overall accuracy. For example, if the *YW* value is high, then the higher the value of this average is, the more important parts for reengineering are likely to be identified with the *YW* heuristic.

On the impact of the release period

Another variation point when computing *Yesterday's Weather* is the release period. If we consider the release period of one week, we focus the analysis on immediate changes. If, on the other hand, we consider the release period of half a year, we emphasize the size of the changes that accumulate in the histories.

Example. Suppose that when we consider the release period of a big system of one week we obtain *YW(S,P,5,5)* = 60% and when we consider the release period of half a year we obtain *YW(S,P,5,5)* = 20%. It means that from one week to another the development is quite focused, and the bigger parts of the system tend to stabilize over a long period of time, thus leading to apparently unexpected changes, *e.g.*, bug-fixing, patching, small functionality increase all over the system.

The variation of *YW* allows one to fine-tune the information. It is the combination of short and focused releases and the decrease of *YW* that allows one to conclude that the system stabilizes - that is, the parts that were changed in the past are no longer changed in the future. Note that, by considering longer release periods, the additions and removals of methods from the same class between consecutive releases will not show in the history measurements.

On the impact of the number of versions

The number of versions represents another variation point when computing *YW*. Increasing or decreasing the number of versions affects the overall *YW*, but has little effect on the value of individual YW_i. The longer the considered history, the less important is a hit/non-hit in the overall *YW*. By increasing the number of versions while keeping the same period between versions, we let the early changes affect the overall *YW*. Therefore, when keeping the period between versions constant, by increasing the number of versions we obtain a long-term trend, while by decreasing the number of versions we concentrate on the short-term trend.

On the impact of the granularity level

YW can be applied at any level of abstraction. We showed the results we obtained on classes and packages with respect to adding and removing methods. Such analysis requires the knowledge about the structure of the system. When such

information is not available, *YW* can be applied, for example, on files and folders with respect to adding and removing lines of code.

4.6 Related Work

Measurements have traditionally been used to deal with the problem of analyzing the history of software systems. Ramil and Lehman explored the implication of the evolution metrics on software maintenance [Ramil and Lehman, 2001]. They used the number of modules to describe the size of a version and define evolutionary measurements which take into account differences between consecutive versions. Recently, the same approach has been employed to characterize the evolution of open source projects [Godfrey and Tu, 2000; Capiluppi, 2003; Capiluppi *et al.*, 2004].

Jazayeri analyzed the stability of the architecture by using colors to depict the changes. Based on the visualization he analyzed how the old code tends to stabilize over time [Jazayeri, 2002].

Rysselberghe and Demeyer used a simple visualization to provide an overview of the evolution of systems: they displayed a plot chart, where each dot represents a commit of the corresponding file [Van Rysselberghe and Demeyer, 2004]. Based on the visualization they detected patterns of evolution.

Jingwei Wu *et al.* used the spectograph metaphor to visualize how changes occur in software systems [Wu *et al.*, 2004a]. They used colors to denote the age of changes on different parts of the systems.

These approaches make use of raw data provided by the versioning system: folders, files and lines of text. As opposed to that, our approach takes into consideration the structure of the system and makes use of the semantics of changes.

Burd and Munro defined a set of measurements to quantify the dominance relations which are used to depict the complexity of the calls. They correlated the changes in these measurements with the types of maintainability activities [Burd and Munro, 1999].

Gold and Mohan defined a framework to understand the conceptual changes in an evolving system [Gold and Mohan, 2003]. Based on measuring the detected concepts they could differentiate between different maintenance activities.

Grosser, Sahraoui and Valtchev applied Case-Based Reasoning on the history of object-oriented system as a solution to a complementary problem to ours: to predict the preservation of the class interfaces [Grosser *et al.*, 2002]. They also considered the interfaces of a class to be the relevant indicator of the stability of a class. Sahraoui *et al.* employed machine learning combined with a fuzzy approach to understand the stability of the class interfaces [Sahraoui *et al.*, 2000].

Our approach differs from the above mentioned ones because we consider the history to be a first class entity and define history measurements which are applied to the whole history of the system and which summarize the evolution of that system. Thus we do not have to analyze manually in detail evolution charts. The drawback of our approach consists in the inherent noise which resides in compressing large amounts of data into numbers.

Fischer *et al.* analyzed the evolution of systems in relation with bug reports to track the hidden dependencies between features [Fischer *et al.*, 2003a]. Demeyer *et al.* proposed practical assumptions to identify where to start a reverse engineering effort: working on the most buggy part first or focusing on clients most important requirements [Demeyer *et al.*, 2002]. These approaches, are based on information that is outside the code, while our analysis is based on code alone.

In a related approach, Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graph [Hassan and Holt, 2004].

4.7 Summarizing Yesterday's Weather

One of the most important issues when starting reverse engineering is knowing where to start. When only the code of the application is available, the history of a software system could be of help. However, analyzing the history is difficult because of the interpretation of large quantities of complex data. We presented our approach of summarizing the history by defining history measurements.

We use the term *Yesterday's Weather* to depict the retrospective empirical observation that at least one of the parts of the system that were heavily changed in the last period will also be among the most changed parts in the near future. We computed it on three case studies and showed how it can summarize the changes in the history of a system. We use the approach to pinpoint parts of the system in

the latest version which would make good candidates for a first step in reverse engineering. We looked closely at how the big changes are related with the size of the classes and packages, and validated our approach by showing that big changes can occur in classes and packages which are not big in terms of size (*i.e.*, number of methods). Thus, our approach is useful to reveal candidates for reengineering which are otherwise undetectable if we only analyze the size of the different parts of the system's last version.

When reverse engineering, we should take the auxiliary development information into account. For example, we could correlate the shape of the evolution of *Yesterday's Weather* with the changes in the team or with the changes in the development process. An example of an analysis of the history of the developers is shown in Chapter 7 (p.105). In the future, we would like to correlate *Yesterday's Weather* with such information from outside the source code.

4.8 Hismo Validation

The approach consists of identifying, for each version of a subject system, the parts that were changed the most in the recent history and in checking if these are also among the most changed parts in the successive versions. The *YW* value is given by the number of versions in which this assumption holds divided by the total number of analyzed versions. Below we give an OCL code for computing *YW* for a SystemHistory with respect to classes:

```
context SystemHistory

  -- returns true if the YW assumption holds for a given version versionRank
  derive versionYW(versionRank):
    yesterdayTopHistories = self.classHistories->selectTopLENOMFromVersions(minRank, versionRank-1).
    todayTopHistories = self.classHistories->selectTopEENOMFromVersions(versionRank, maxRank).
    yesterdayTopHistories->intersectWith(todayTopHistories)->isNotEmpty().

  -- answers the number of versions in which the assumption holds
  -- divided by the total number of analyzed versions
  derive overallYW:
    ywVersionResults = Sequence(minRank+2..maxRank-1)->collect(i | self.versionYW(i).
    ywVersionResults->sum() / (maxRank-minRank-2)
```

The code reveals several features of Hismo :

— We navigate the meta-model by asking a SystemHistory for all the ClassHistories (self.classHistories).

— classHistories->selectTopLENOMFromVersions(minRank, versionRank-1) returns the class histories that are in the top of LENOM (Latest Evolution of Number of Methods) in the period between the first version (minRank) and the version before the wanted version (versionRank-1). That is, it returns the classes that were changed the most in the recent history. This predicate implies applying a historical measurement (*i.e.*, LENOM) on a selection of a history, and then ordering the histories according to the results of the measurement.

— Similarly, classHistories->selectTopEENOMFromVersions(versionRank, maxRank) returns the class histories that are the most changed in the early history between the wanted version (versionRank) and the last version (maxRank).

— The result of versionYW is given by the intersection of the past changed class histories and the future changed class histories yesterdayTopHistories->intersectWith(todayTopHistories)->isNotEmpty(). This simple intersection is possible because the yesterdayTopHistories and todayTopHistories are subsets of all classHistories.

Chapter 5

History-Based Detection Strategies

Combining Historical Properties with Structural Properties

From an abstract enough point of view,
any two things are similar.

As systems evolve and their structure decays, maintainers need accurate and automatic identification of the design problems. Current approaches for automatic detection of design problems take into account only a single version of a system, and consequently, they miss essential information as design problems appear and evolve over time. Our approach is to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection. Our means is to define measurements which summarize how persistent the problem was and how much maintenance effort was spent on the suspected structure. We apply our approach to a large scale case study and show how it improves the accuracy of the detection of God Classes *and* Data Classes*, and additionally how it adds valuable semantic information about the evolution of flawed design structures.*

This approach shows how to combine historical properties with snapshot properties.

5.1 Introduction

Maintenance effort is reported to be more than half of the overall development effort and most of the maintenance effort is spent on adapting and introducing new requirements, rather than in repairing errors [Bennett and Rajlich, 2000; Sommerville, 2000]. One important source of maintainability problems is the accumulation of poor or improper design decisions. This is the reason why, during the past years, the issue of identifying and correcting design problems became an important concern for the object-oriented community [Fowler *et al.*, 1999; Riel, 1996; Demeyer *et al.*, 2002].

Various analysis approaches have been developed to automatically detect where the object-oriented design problems are located, but these approaches only make use of the information found in the last version of the system (*i.e.,* the version which is maintained) [Ciupke, 1999; Marinescu, 2001]. For example, they look for improper distribution of functionality among classes of a system without asking whether or not it raised maintenance problems in the past.

We argue that the evolution information of the problematic classes over their lifetime can give useful information to system maintainers. We propose a new approach which enriches the detection of design problems by combining the analysis based on a single version with the information related to the evolution of the suspected flawed classes over time.

We show how we apply our approach when detecting two of the most well known design flaws: *Data Class* and *God Class*. Marinescu detected these flaws by applying measurement-based rules to a single version of a system [Marinescu, 2002; Marinescu, 2004]. He named these rules *detection strategies*. The result of a detection strategy is a list of *suspects*: design structures (*e.g.,* classes) which are suspected of being flawed. We enlarge the concept of detection strategies by taking into account the history of the suspects (*i.e.,* all versions of the suspects). We define history measurements which summarize the evolution of the suspects and combine the results with the classical detection strategies.

Structure of the Chapter

After we present the metaphor of our approach, we briefly describe the concept of *detection strategy* and discuss the detection of *Data Classes* and *God Classes* (Section 5.3 (p.69)). In Section 5.5 (p.74) we define the history measurements needed

to extend the detection strategies and discuss the way we use historical information in detection strategies. We then apply the new detection strategies on a large open source case study and discuss in detail the results (Section 5.6 (p.77)).We give an overview of the related work in Section 5.8 (p.83), we summarize the approach in Section 5.9 (p.84). In Section 5.10 (p.85) we discuss the approach pointing out the benefits of using Hismo.

5.2 The Evolution of Design Flaw Suspects

Design flaws (*e.g., God Class* or *Data Class*) are like human diseases - each of them evolves in a special way. Some diseases are hereditary, others are acquired during the life-time. The hereditary diseases are there since we were born. If physicians are given a history of our health status over time they can give their diagnostic in a more precise way. Moreover there are diseases (*e.g.,* benign tumors) with which our organism is accustomed and thus, they represent no danger for our health and we don't even consider them to be diseases any more.

In a similar fashion we use the system's evolution to increase the accuracy of the design flaw detection. We analyze the history of the suspects to see whether the flaw caused problems in the past. If in the past the flaw proved not to be harmful then it is less dangerous. For example, in many cases, the generated code needs no maintenance so the system which incorporates it can live a long and serene life no matter how the generated code appears in the sources (*e.g.,* large classes or unreadable code).

The design flaws evolve with the system they belong to. As systems get older their diseases are more and more prominent and need to be more and more taken into account.

5.3 Detection Strategies

A detection strategy is a generic mechanism for analyzing a source code model using metrics. It is defined by its author as *the quantifiable expression of a rule, by which design fragments that are conformant to that rule can be detected in the source code* [Marinescu, 2002].

Detection strategies allow one to work with measurements at a more abstract level,

which is conceptually much closer to our real intentions in using metrics (*e.g.*, for detecting design problems). The result of applying a detection strategy is a list of design structures suspected of being flaed. Using this mechanism it becomes possible to quantify several design flaws described in the literature [Riel, 1996; Fowler *et al.*, 1999].

We present below the detection strategies for *Data Class* and *God Class*. Their presentation will also clarify the structure of a detection strategy.

5.3.1 *God Class* Detection Strategy

God Class "refers to a class that tends to centralize the intelligence of the system. An instance of a *God Class* performs most of the work, delegating only minor details to a set of trivial classes and using the data from other classes" [Marinescu, 2002].

To detect a *God Class* we look for classes which use a lot of data from the classes around them while either being highly complex or having a large state and low cohesion between methods. The *God Class* detection strategy is a quantified measurement-based rule expressing the above description (see Equation 5.1 (p.70)). We introduce below the measurements used:

— Access to Foreign Data (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods [Marinescu, 2002].

— Weighted Method Count (WMC) is the sum of the statical complexity of all methods in a class [Chidamber and Kemerer, 1994]. We considered the McCabe's cyclomatic complexity as a complexity measure [McCabe, 1976].

— Tight Class Cohesion (TCC) is the relative number of directly connected methods [Bieman and Kang, 1995].

— Number of Attributes (NOA) [Lorenz and Kidd, 1994].

$$GodClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \\ \forall C \in S' \\ (ATFD(C) > 40) \wedge ((WMC(C) > 75) \vee \\ ((TCC < 0.2) \wedge (NOA > 20))) \end{array} \right. \quad (5.1)$$

5.3.2 *Data Class* Detection Strategy

Data Classes "are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes" [Fowler *et al.*, 1999].

To detect a *Data Class* we look for classes which have a low complexity and high exposure to their internal state (*i.e.*, a lot of either accessor methods or public attributes). The *Data Class* detection strategy in Equation 5.2 (p.71) uses the following measurements:

— Weight of a Class (WOC) is the number of non-accessor methods in the interface of the class divided by the total number of interface members [Marinescu, 2001].

— Number of Methods (NOM) [Lorenz and Kidd, 1994].

— Weighted Method Count (WMC) [Chidamber and Kemerer, 1994].

— Number of Public Attributes (NOPA) is defined as the number of non-inherited attributes that belong to the interface of a class [Marinescu, 2001].

— Number of Accessor Methods (NOAM) is defined as the number of the non-inherited accessor-methods declared in the interface of a class [Marinescu, 2001].

$$DataClass(S) = S' \left| \begin{array}{l} S' \subseteq S, \\ \forall C \in S' \\ ((\frac{WMC(C)}{NOM(C)} < 1.1) \wedge (WOC(C) < 0.5)) \wedge \\ ((NOPA(C) > 4) \vee (NOAM(C) > 4)) \end{array} \right. \tag{5.2}$$

5.3.3 Detection Strategy Discussion

As shown in the Equation 5.1 (p.70) and Equation 5.2 (p.71) the detection strategies are based on a skeleton of measurements and thresholds for each measurement (*e.g.*, $ATFD > 40$). While the measurements skeleton can be obtained by translating directly the available informal rules (*e.g.*, heuristics or bad smells), the particular sets of thresholds are mainly chosen based on the experience of the analyst.

As this experience can differ from person to person the thresholds represent the weak point of the detection strategies.

The process of using detection strategies to detecting design flaws is not fully automated. Indeed, detection strategies are automated, but the result of a detection strategy is a list of suspects which requires further human intervention to verify the flaw.

5.4 History Measurements

We refine the detection of design flaws by taking into consideration how *stable* the suspects were in the past and how long they have been suspected of being flawed. We name *persistently flawed*[1] the entities which were suspects a large part of their life-time (*i.e.,* more than 95% of their life time). Thus we further introduce two measurements applied on the history of a suspect: $Stab$ and $Pers$.

5.4.1 Measuring the Stability of Classes

We consider that a class was stable with respect to a measurement M between version $i-1$ and version i if there was no change in the measurement. As an overall indicator of stability, we define the $Stab$ measurement applied on a class history H as a fraction of the number of versions in which there was a change in the M measurement over the total number of versions - 1 (see Equation 5.3 (p.72)).

$$(i > 1) \qquad Stab_i(H, M) = \begin{cases} 1, & M_i(H) - M_{i-1}(H) = 0 \\ 0, & M_i(H) - M_{i-1}(H) \neq 0 \end{cases}$$

$$(n > 2) \qquad Stab_{1..n}(H, M) = \frac{\sum_{i=2}^{n} Stab_i(H, M)}{n-1} \tag{5.3}$$

The $Stab$ measurement returns a value between 0 and 1, where 0 means that the history was changed in all versions and 1 means that it was never changed.

[1]The adjective persistent is a bit overloaded. In this dissertation we use its first meaning: *existing for a long or longer than usual time or continuously.* Merriam-Webster Dictionary

The functionality of classes is defined in their methods. For the purpose of the current analysis, we consider that a class was changed if at least one method was added or removed. Thus, we will use $Stab$ with respect to the Number of Methods of a class (NOM).

5.4.2 Measuring the Persistency of a Design Flaw

We define the $Pers$ measurement of a flaw F for a class history H with n versions, *i.e.,* 1 being the oldest version and n being the most recent version (see Equation 5.4).

$$(i \geq 1) \qquad Suspect_i(H, F) = \begin{cases} 1, & H_i \; is \; suspect \; of \; flaw \; F \\ 0, & H_i \; is \; not \; suspect \; of \; flaw \; F \end{cases}$$

$$(n > 2) \qquad Pers_{1..n}(H, F) = \frac{\sum_{i=1}^{n} Suspect_i(H, F)}{n} \tag{5.4}$$

The $Pers$ measurement returns a value between 0 and 1, and it tells us in what measure the birth of the flaw is related with the design stage or with the evolution of the system. For example, if for a history $Pers$ is 1, it means that the flaw was present from the very beginning of the history.

Example. The top part of Figure 5.1 (p.74) presents 5 class histories and the results of the $Stab$ and $Pers$ measurements, and the bottom part shows in details how we obtain the values for the two measurements in the case of A. We can interpret the persistent flaws in one of the following ways:

1. The developers are conscious of these flaws and they could not avoid making them. This could happen because of particularities of the modeled system - the essential complexities [Brooks, 1987] - or the need to meet other quality characteristics (*e.g.,* efficiency).
2. The developers are not conscious of the flaws. The cause for this can be either the lack of expertise in object-oriented design or the trade-offs the programmers had to do due to external constraints (*e.g.,* time pressure).

The flaws which are not persistent are named *volatile*, and are the result of the system's evolution. These situations are usually malignant because they reveal

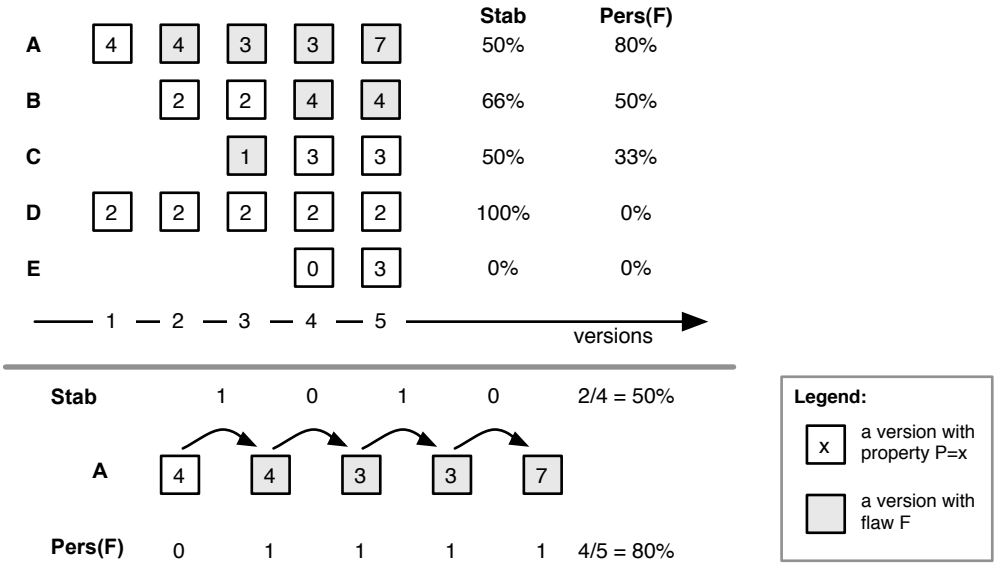


Figure 5.1: Examples of the computation of the $Stab$ and $Pers$ measurements.

the erosion of the initial design. They have two major causes:

1. The apparition of new (usually functional) requirements which forced the developers to modify the initial design to meet them.
2. The accumulation of accidental complexity in certain areas of the system due to the changing requirements [Brooks, 1987].

From the point of view of maintenance, we are interested mainly in the detecting the erosion of the original design.

5.5 Detection Strategies Enriched with Historical Information

We use the history measurements to enrich the *God Class* and *Data Class* detection strategies. The detection strategies used here are based on the work of Raţiu [Raţiu, 2003]. In Equation 5.5 (p.75) and in Equation 5.6 (p.75) we define the *StableGodClass* and *PersistentGodClass* detection strategies. The rest of the detection strategies used further are defined in a similar fashion. The only difference is that

while stability is measured for a class in isolation, the instability for a class is measured relatively to the other classes within the system.

$$StableGodClass(S_{1..n}) = S' \left| \begin{array}{l} S' \subseteq GodClass(S_n), \\ \forall C \in S' \\ Stab(C) > 95\% \end{array} \right. \tag{5.5}$$

$$PersGodClass(S_{1..n}) = S' \left| \begin{array}{l} S' \subseteq GodClass(S_n), \\ \forall C \in S' \\ Pers(C, GodClass) > 95\% \end{array} \right. \tag{5.6}$$

***God Classes* and Stability.** *God Classes* are big and complex classes which encapsulate a great amount of system's knowledge. They are known to be a source of maintainability problems [Riel, 1996]. However, not all *God Classes* raise problems for maintainers. The stable *God Classes* are a benign part of the *God Class* suspects because the system's evolution was not disturbed by their presence. For example, they could implement a complex yet very well delimited part of the system containing a strongly cohesive group of features (*e.g.*, an interface with a library).

On the other hand, the changes of a system are driven by changes in its features. Whenever a class implements more features it is more likely to be changed. *God Classes* with a low stability were modified many times during their life-time. Therefore, we can identify *God Classes* which have produced maintenance problems during their life from the set of all *God Classes* identified within the system. The unstable *God Classes* are the malignant sub-set of *God Class* suspects.

***God Classes* and Persistency.** The persistent *God Class* are those classes which have been suspects for almost their entire life. Particularizing the reasons given above for persistent suspects in general, a class is usually born *God Class* because one of the following reasons:

1. It encapsulates some of the essential complexities of the modeled system. For example, it can address performance problems related to delegation or it can belong to a generated part of the system.

2. It is the result of a bad design because of the procedural way of regarding data and functionality, which emphasis a functional decomposition instead of a data centric one.

It is obvious that *God Classes* which are problematic belong only to the latter category because in the first category the design problem can not be eliminated.

God Class suspects which are volatile obtained the *God Class* status during their lifetime. We argue that persistent *God Classes* are less dangerous than those which are not persistent. The former were designed to be large and important classes from the beginning and thus are not so dangerous. The latter more likely occur due to the accumulation of accidental complexity resulted from the repeated changes of requirements and they degrade the structure of the system.

***Data Classes* and Stability.** *Data Classes* are lightweight classes which contain only little functionality. While *God Classes* carry on the work, *Data Classes* are only dumb data providers whose data is used from within other (possible many) classes. Modifications within *Data Classes* require a lot of work from programmers, as the principle of *locality of change* is violated. Thus, regarding the efforts implied by their change, programmers are less likely to change *Data Classes*. Based on this, we infer that the more relevant functionality a *Data Class* contains the higher are its chances to become the subject of a change. From this point of view, "classic" *Data Classes*, which are nothing else but dumb data carriers with a very light functionality, should be rather stable.

***Data Classes* and Persistency.** Persistent *Data Classes* represent those classes which were born with this disease. They break from the beginning of their life the fundamental rule of object-oriented programming which states that data and its associated functionality should stay together. Particularizing the reasons which could lead to persistent *Data Classes* we obtain that:

1. The class is used only as a grouping mechanism to put together some data. For example such a class can be used where is necessary to transmit some unrelated data through the system.

2. There was a design problem as *Data Classes* do not use any of the mechanisms specific to object-oriented programming (*i.e.,* encapsulation, dynamic binding or polymorphism). The data of these classes belong together but the corresponding functionality is somewhere else.

Data Classes which are volatile are classes which got to be *Data Class* during their life. A class can become a *Data Class* either by requirements change in the direction of functionality removal or by refactorings. Functionality removal while keeping the data is unlikely to happen. Furthermore by properly applying the refactorings as defined in [Fowler *et al.*, 1999] we can not get classes with related data but no functionality (*i.e.,* the malignant set of *Data Classes*). The only

Data Classes we are likely to get are those which belong to the harmless category, because the class is used only as a modularization mechanism for moving easily unrelated data.

5.6 Validation

We applied our approach to three case studies: two in-house projects and Jun[2], a large open source 3D-graphics framework written in Smalltalk. As experimental data we chose every 5th version starting from version 5 (the first public version) to version 200. In the first analyzed version there were 160 classes while in the last analyzed version there were 694 classes. There were 814 different classes which were present in the system over this part of its history. Within these classes 2397 methods were added or removed through the analyzed history.

We first applied the detection strategies and then the suspects were both manually inspected at the source-code level and looked-up in Jun's user manual. Based on the manual inspection we determined which suspects were false positives, however we did not have access to any internal expertise.

The history information allowed us to distinguish among the suspects provided by single-version detection strategies, the *harmful* and *harmless* ones. This distinction among suspects drives the structure of the entire section.

The analysis shows how, by adding information related to the evolution of classes, the accuracy of the detection results was improved on this case-study, both for *God Classes* and *Data Classes*. Additionally, it shows that the history of the system adds valuable semantic information about the evolution of flawed design structures.

Harmless *God Classes*. The *God Classes* which are *persistent* and *stable* during their life are the most harmless ones. They lived in symbiosis with the system along its entire life and raised no maintainability problems in the past.

When taking a closer look at the Figure 5.2 (p.78) which present the *God Class* suspects we observe that more than 20% of them are persistent and stable (5 out of 24). These classes in spite of their large size, did not harm the system's maintainability in the past, because the developers did not need to change them, so it is unlikely that they will harm it in the future. Almost all of these classes

[2]See http://www.srainc.com/Jun/.

	GodClass Suspect	Persistent	Volatile	Stable	Unstable
	JunHistogramModel	x		x	
	JunLispCons	x		x	
	JunLispInterpreter	x		x	
	JunSourceCodeDifference	x		x	
	JunSourceCodeDifferenceBrowser	x		x	
	JunChartAbstract	x			x
	JunOpenGLGraph	x			x
	JunOpenGLDisplayModel	x			x
	JunOpenGLShowModel	x			x
	JunUNION	x			x
	JunImageProcessor_class	x			
	JunOpenGLRotationModel	x			
	JunUniFileModel	x			
	JunVoronoi2dProcessor	x			
	JunMessageSpy		x	x	
	JunOpenGLDisplayLight		x	x	
	Jun3dLine		x		x
	JunBody_class		x		x
	JunEdge		x		x
	JunLoop		x		x
	JunOpenGL3dCompoundObject		x		x
	JunOpenGL3dObject_class		x		x
	JunPolygon		x		x
	JunBody		x		
Total	**24**	**14**	**10**	**7**	**12**

Figure 5.2: *God Classes* detected in version 200 of Jun case-study and their history properties.

belong to particular domains which are weakly related with the main purpose of the application. We can observe this even by looking at their names. In Jun these classes belong to a special category named "Goodies" which means that they are optional add-ons.

— *JunLispInterpreter* is a class that implements a Lisp interpreter, one of the supplementary utilities of Jun. This is an example of a GodClass that models a matured utility part of the system.

— *JunSourceCodeDifferenceBrowser* is used to support the evolution of Jun. It belongs to the effort of the developers to support the evolution of the library itself.

Continuing to look at the Figure 5.2 (p.78) we notice that some of the *God Classes* were stable during their lifetime even if they were not necessarily persistent. The manual inspection revealed that some of these classes were born as skeletons of *God Classes* and waited to be filled with functionality at a later time. Another part of them was not detected to be persistent because of the hard thresholds of the detection strategies. We can consider these classes to be a less dangerous category of *God Classes*.

— *JunOpenGLDisplayLight* is a GUI class (*i.e.,* a descendant of UI.ApplicationModel), which suddenly grew in version 195.

— *JunMessageSpy* is a helper class which is used for displaying profiling information for messages. It also belongs to the 'Goodies' category. This class was detected as being *God Class* from the beginning because of the hard thresholds used in the detection strategy.

Harmful *God Classes*. The *God Classes* which were both *volatile* and *unstable* are the most dangerous ones. Looking at the Figure 5.2 (p.78) we can easily see that almost 30% of the *God Classes* are harmful (7 out of 24). They grew as a result of complexity accumulation over the system's evolution and presented a lot of maintainability problems in the past. The inspection of volatile unstable *God Classes* reveals that they all belong to the core of the modeled domain, which is in this case graphics.

— *JunOpenGL3dCompoundObject* implements the composition of more 3D objects. Its growth is the result of a continuous accumulation of complexity from version 75 to version 200.

— *JunBody* models a single 3D solid element. Between version 100 and 150 its

	DataClass Suspect	Persistent	Volatile	Stable	Unstable
	JunParameter	x		x	
	JunPenTransformation	x		x	
	JunVoronoi2dTriangle	x		x	
	JunVrmlTexture2Node	x		x	
	Jun3dTransformation	x			
	JunVrmlIndexedFaceSetNode20	x			
	JunVrmlTransformNode	x			
	JunHistoryNode		x		
	JunVrmlMaterialNode		x		
	JunVrmlNavigationInfoNode		x		
	JunVrmlViewPointNode		x		
Total	**11**	**7**	**4**	**4**	**0**

Figure 5.3: *Data Classes* detected in version 200 of the Jun case study and their history properties.

complexity grew by a factor of 3.

— *JunEdge* element represents a section where 2 faces meet. It had a continuous growth of WMC complexity between versions 10 and 155.

Harmless *Data Classes*. We consider volatile *Data Classes* to be less dangerous. The manual inspection revealed that the accessor methods of these classes are not used from exterior classes. They are used only as local wrappers for their instance variables or as instance initializers from their meta-classes.

Harmful *Data Classes*. The most dangerous *Data Classes* are those which were designed that way from the beginning. The manual inspection revealed that 3 out of 7 of these *Data Classes* (*i.e.*, almost 50% of persistent *Data Classes*) are used from within other classes.

— *JunVoronoi2dTriangle* has its accessors used from within JunVoronoi2dProcessor which is a persistent *God Class*.

— *Jun3dTransformation* is used from JunOpenGL3dObject_class which is a volatile *God Class*.

— *JunParameter* is used from within JunParameterModel.

Suspects	Detected	False Positives
Classic GodClass Suspects	24	---
Harmless GodClass Suspects	5	0
Harmful GodClass Suspects	12	0
Not Classified GodClass Suspects	7	---
Classic DataClass Suspects	11	---
Harmless DataClass Suspects	7	4
Harmful DataClass Suspects	4	0
Not Classified DataClass Suspects	0	---

Figure 5.4: Summary of the results of the history-based detection strategies as applied on Jun.

The other 4 persistent *Data Class* suspects proved to be false positives as the manual inspection of the suspects proved that their accessors are used only from within their class or for initialization from their meta-class.

Summary. Figure 5.4 (p.81) summarises the results of the case-study analysis, showing explicitly the accuracy improvement compared with the single-version detection strategies.

From the total of 24 classes which were suspect of being *God Classes* using the classic detection strategies, there were 5 of them detected to be harmless and 7 of them detected to be harmful. After the manual inspection, no false positives were found. There were 12 suspects being *God Classes* which were not classified as being either harmful or harmless. This category of suspects require manual inspection as the time information could not improve the classic detection.

The classic *Data Class* detection strategies detected 11 suspects. 7 of these were detected as being harmful and the other 4 detected to be harmless. After the manual inspection, we found 4 false positives out of 7 of the *Data Classes* detected as being harmful.

5.7 Variation Points

During the experiments we had to choose among many different possibilities to deal with the time information. We consider necessary a final discussion centered on the possible variations of this approach. Thus, the purpose of this section is to put the chosen approach in a larger context driving in the same time the directions for future work.

On the variations of the stability measurement

We consider a class unstable if *at least* one method was added or removed regardless of the kind of method. Therefore, we ignored the changes produced at a lower level (*e.g.,* bug-fixes) and do not distinguish the quality of changes. A possible solution would be to sum the number of changes in successive versions and complement the $Stab$ measurement with size information.

Also, the $Stab$ measurement considers all changes equal, regardless of the period of time in which they appeared. Another possible extension is to consider just the latest changes by weighting each change with the distance from the last version analyzed.

On the impact of the number of analyzed versions

The more versions we consider in our analysis, the more we favor the capture of the small changes in the $Stab$ measurement.

On the impact of the starting point of the analyzed history

We found out that persistent and stable *God Classes* are harmless because they usually implement an optional and standalone feature (*e.g.,* a code browser in a 3D framework). These classes are part of the initial design, as persistency means the existence of the flaw almost since the very beginning of the class life. If we consider the analyzed period to be from the middle of their actual life, we cannot detect whether they were really persistent. Therefore, for persistency we need to consider the history from the beginning of the system's history.

On the impact of the threshold values

The thresholds represent the weakest point of the detection strategies because they are established mainly using the human experience. In this chapter the time information is used to supplement the lack of precision for particular sets of thresholds [Raţiu, 2003].

Mihancea developed a "tuning machine" which tries to infer automatically the proper threshold values [Mihancea and Marinescu, 2005]. This approach is based on building a repository of design fragments that have been identified by engineers as being affected by a particular design problem, and based on this reference samples, the "tuning machine" selects those threshold values that maximize the number of correctly detected samples. The drawback of the approach is that the examples repository must be large enough to allow a proper tuning, and collecting these samples is not easy.

Another approach to deal with setting the thresholds is based on statistical analysis of the common values found in a database of projects [Lanza *et al.*, 2006].

5.8 Related Work

During the past years, different approaches have been developed to address the problem of detecting design flaws. Ciupke employed queries usually implemented in Prolog to detect "critical design fragments" [Ciupke, 1999]. Tourwe *et al.* also explored the use of logic programming to detect design flaws. van Emden *et al.* detected bad smells by looking at code patterns [van Emden and Moonen, 2002]. As mentioned earlier, Marinescu defined detection strategies which are measurement-based rules aimed to detect design problems [Marinescu, 2001; Marinescu, 2002; Marinescu, 2004]. Visualization techniques have also been used to understand design [Demeyer *et al.*, 1999].

These approaches have been applied on one single version alone, therefore missing useful information related to the history of the system. Our approach is to complement the detection of design fragments with history information of the suspected flawed structure.

Demeyer *et al.* analyzed the history of three systems to detect "refactorings via change metrics" [Demeyer *et al.*, 2000]. Krajewski defined a methodology for analyzing the history of software systems [Krajewski, 2003].

These approaches used measurements as an indicator of changes from one version to another. We consider history to be a first class entity and define history measurements which summarize the entire evolution of that entity.

Visualization techniques in combination with measurements were also used to understand history information [Lanza and Ducasse, 2002; Jazayeri *et al.*, 1999; Jazayeri, 2002]. As opposed to visualization, our approach is automatic and it reduces the scope of analysis, yet we believe that the two approaches are complementary.

In [Fischer *et al.*, 2003a] the authors used information outside the code history and looked for feature tracking, thus gaining semantic information for software understanding. Our approach differs as we only make use of the code history. Another approach was developed by Gall *et al.* to detect hidden dependencies between modules, but they considered the module as unit of analysis while we base our analysis on detailed structural information from the code [Gall *et al.*, 1998].

Jun has been the target of evolution research, however the focus was to use the history information to describe the development process and the lessons learnt from developing Jun [Aoki *et al.*, 2001].

5.9 Summarizing History-Based Detection Strategies

We refined the original concept of *detection strategy*, by using historical information of the suspected flawed structures. Using this approach we showed how the detection of *God Classes* and *Data Classes* can become more accurate. Our approach leads to a twofold benefit:

1. *Elimination of false positives* by filtering out, with the help of history information, the harmless suspects from those provided by a single-version detection strategy.
2. *Identification of most dangerous suspects* by using additional information on the evolution of initial suspects over their analyzed history.

To consolidate and refine the results obtained on the Jun case study, the approach needs to be applied further on large-scale systems. We also intend to extend our investigations towards the usage of historical information for detecting other design flaws.

The $Stab$ measurement does not take into consideration the sizes of the change. We would like to investigate how this indicator of stability could be improved by considering the number of methods changed between successive versions. Other interesting investigation issues are: the impact on stability of other change measurements (*e.g.,* lines of code) and the detection of periodic appearing and disappearing of flaws.

5.10 Hismo Validation

Originally, the detection strategies only took into account structural measurements. We used Hismo to extend the detection strategies with the time dimension. For example, we use the historical information to qualify *God Classes* as being harmless if they were stable for a large part of their history. Below we present how we define the expression with Hismo:

```
context ClassHistory

  -- returns true if the ClassHistory is a GodClass in the last version and
  -- it did not change the number of methods in more then 95% of the versions
  derive isHarmlessGodClass:
    (self.versions->last().isGodClass()) &
    (self.stabilityOfNOM > 0.95)
```

The code shows the predicate isHarmlessGodClass defined in the context of a ClassHistory. The predicate is an example of how we can use in a single expression both historical information (*i.e.,* stability of number of methods) and structural information (*i.e., God Class* characteristic in the last version) to build the reasoning.

One variation of the approach is in using other stability indicators than the changes in the number of methods. For example, we can use changes in the number of statements of the class. The simplicity of the above OCL code, easily allows for the replacement of the self.stabilityOfNOM with self.stabilityOfNOS.

Chapter 6

Characterizing the Evolution of Hierarchies

Combining Historical Relationships and Historical Properties

Problem solving efficiency grows
with the abstractness level of problem understanding.

Analyzing historical information can show how a software system evolved into its current state, which parts of the system are stable and which have changed more. However, historical analysis implies processing a vast amount of information making the interpretation of the results difficult. Rather than analyzing individual classes, we aim to analyze class hierarchies as they group classes according to similar semantics. We use historical measurements to define rules by which to detect different characteristics of the evolution of class hierarchies. Furthermore, we discuss the results we obtained by visualizing them using a novel polymetric view. We apply our approach on two large open source case studies and classify their class hierarchies based on their history.

With this analysis we show how to use relationship histories and how to combine history properties.

6.1 Introduction

History holds useful information that can be used when reverse engineering a system. However, analyzing historical information is difficult due to the vast amount of information that needs to be processed, transformed, and understood. Therefore, we need higher level views of the data which allow the reverse engineer to ignore the irrelevant details.

We concentrate on describing and characterizing the evolution of class hierarchies. Class hierarchies provide a grouping of classes based on similar semantics thus, characterizing a hierarchy as a whole reduces the complexity of understanding big systems. In particular, we seek answers to the following questions:

1. *How old are the classes of a hierarchy?* On one hand, the old classes may be part of the original design and thus hold useful information about the system's design. On the other hand, a freshly introduced hierarchy might indicate places where future work will be required.

2. *Were there changes in the inheritance relationships?* Changes in the inheritance relationships might indicate class renaming refactorings, introduction of abstract classes, or removal of classes.

3. *Are classes from one hierarchy modified more than those from another one?* Understanding which parts of the system changed more is important because they might be also changed in the future [Gîrba *et al.*, 2004a].

4. *Are the changes evenly distributed among the classes of a hierarchy?* If we find that the root of the hierarchy is often changed, it might indicate that effort was spent to factor out functionality from the subclasses and to push it to the superclass, but it might also be a sign of violations of the open-closed principle [Meyer, 1988].

We are set to detect four characteristics of class hierarchy evolution: (1) the age of the hierarchy, (2) the inheritance relationship stability, (3) the class size stability, and (4) the change balance. We quantify these characteristics in a set of measurements-based rules, and define a language for describing different evolution patterns of class hierarchies.

To analyze the obtained results we developed a visualization called *Hierarchy Evolution Complexity View*, an evolutionary polymetric view [Lanza, 2003]. We use *software visualization* because visual displays allow the human brain to study multiple aspects of complex problems – like reverse engineering – in parallel [Stasko

et al., 1998]. We validated our approach on two open source case studies written in Java and Smalltalk.

The contributions of this chapter are: (1) the characterization of class hierarchy evolution based on explicit rules which detect different change characteristics, and (2) the definition of a new polymetric view which takes into account the history of entities.

Structure of the chapter

In Section 6.2 (p.89) we define rules to detect different evolution patterns of the evolution of class hierarchies and in Section 6.3 (p.91) we introduce the visualization based on the historical measurements measurements. In Section 6.4 (p.93) we apply our approach on two large case studies. We discuss variation points of our approach (Section 6.5 (p.98)) and we browse the related work (Section 6.6 (p.101)). We summarize the chapter in Section 6.7 (p.102) and in Section 6.8 (p.103) we end with a discussion of the approach from the point of view of using Hismo.

6.2 Characterizing Class Hierarchy Histories

6.2.1 Modeling Class Hierarchy Histories

We consider a class hierarchy history as being a group of class histories and a group of inheritance histories. To measure class hierarchy histories, we apply group operators like average (*Avg*), maximum (*Max*) and total (*Tot*). Thus we have the average age of the class histories in a class hierarchy Ch as given by: $Avg(ClassAge, Ch)$.

6.2.2 Detecting Class Hierarchies Evolution Patterns

We formulate a vocabulary based on four characteristics: (1) hierarchy age, (2) inheritance relationship stability, (3) class size stability, and (4) change balance. These four characteristics are orthogonal with each other. For example, the same hierarchy can be old but at the same time can have unstable classes.

We use the history measurements to define rules to qualify a class hierarchy based on the four characteristics:

1. How old is a hierarchy? We distinguish the following types of hierarchy histories based on the average age of their class histories:

   — *Newborn*. A newborn hierarchy is a freshly introduced hierarchy (*i.e.,* on the average the age of the class histories is no more than a tenth of the age of the system).

   $$Avg(ClassAge, Ch) < 0.1 * SystemAge$$

   — *Young*. A young hierarchy is older than a newborn hierarchy, but its age is less than half of the system age.

   $$Avg(ClassAge, Ch) \geqslant 0.1 * SystemAge \wedge$$
   $$Avg(ClassAge, Ch) < 0.5 * SystemAge$$

   — *Old*. Old hierarchies have been in the system for a long time, but not for the entire life of the system.

   $$Avg(ClassAge, Ch) \geqslant 0.5 * SystemAge \wedge$$
   $$Avg(ClassAge, Ch) < 0.9 * SystemAge$$

   — *Persistent*. We say a hierarchy is persistent if the classes were present in almost all versions of the system (*i.e.,* in more than 90% of the system versions).

   $$Avg(ClassAge, Ch) \geqslant 0.9 * SystemAge$$

2. Were there changes in the inheritance relationship? We divide the hierarchies into two categories:

   — *Solid*. We define a hierarchy as being solid when the inheritance relationships between classes are stable and old.

   $$Tot(RemovedInh, Ch) < 0.3 * NOInh(Ch)$$

   — *Fragile*. A hierarchy is fragile when there are many inheritance relationships which disappear.

   $$Tot(RemovedInh, Ch) \geqslant 0.3 * NOInh(Ch)$$

3. Are classes from one hierarchy modified more than classes from another hierarchy? From the stability of size point of view we detect two kinds of hierarchies:

   — *Stable*. In a stable hierarchy the classes have few methods and statements added or removed compared with the rest of the system.

   $$Avg(ENOM, Ch) < Avg(ENOM, S) \wedge$$
   $$Avg(ENOS, Ch) < Avg(ENOS, S)$$

   — *Unstable*. In an unstable hierarchy many methods are being added and removed during its evolution.

   $$Avg(ENOM, Ch) \geqslant Avg(ENOM, S) \vee$$
   $$Avg(ENOS, Ch) \geqslant Avg(ENOS, S)$$

4. Are the changes evenly distributed among the classes of a hierarchy? We distinguish two labels from the change balance point of view:

   — *Balanced*. In a balanced hierarchy, the changes are evenly performed among its classes.

   $$Avg(ENOM, Ch) \geqslant 0.8 * Max(ENOM, Ch)$$

   — *Unbalanced*. An unbalanced hierarchy is one in which the changes are not equally distributed in the classes.

   $$Avg(ENOM, Ch) < 0.8 * Max(ENOM, Ch)$$

6.3 Class Hierarchy History Complexity View

To analyze the results we obtain when applying the rules defined in the previous section, we use visualization as it allows one to identify cases where different characteristics apply at the same time for a given hierarchy. The visualization we propose is called *Hierarchy Evolution Complexity View* and is a polymetric view

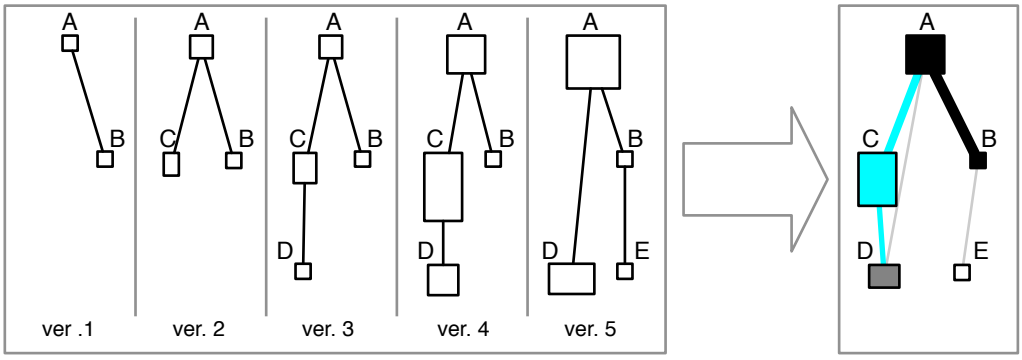


Figure 6.1: An example of the *Hierarchy Evolution Complexity View*. *Hierarchy Evolution Complexity View* (right hand side) summarizes the hierarchy history (left hand side).

[Lanza, 2003]. *Hierarchy Evolution Complexity View* uses a simple tree layout to seemingly display classes and inheritance relationships. What is new is that it actually visualizes the *histories* of classes and of inheritance relationships.

Nodes and edges which have been removed while the system was evolving (*i.e.*, they are not present anymore) have a cyan color. The color of the class history nodes, and the width of the inheritance edges represent their age: the darker the nodes and the wider the edges, the more *grounded* in time they are, *i.e.*, the longer they have been present in the system. Thus, lightly colored nodes and thin edges represent *younger* classes and inheritance relationships.

The width of the class history node is given by the Evolution of Number of Methods (*i.e., ENOM*) while the height is given by the fifth part of the Evolution of Number of Statements (*i.e., ENOS*/5). We chose to divide *ENOS* by 5 because in the case studies we analyzed there was an average of 5 statements per method. Thus, the wider a node is, the more methods were added or removed in subsequent versions in that class history; the greater the height of a node is, the more statements were added or removed in subsequent versions in that class history.

We chose to use *ENOM* and *ENOS* because we wanted to see the correlation between changes in the behavior of a class and its internal implementation changes.

Example. In Figure 6.1 (p.92) we show an example of such a view in which we display an imaginary hierarchy history. From the figure we infer the following information:

— Classes A and B are in the system from the very beginning, and they appear

colored in black. The inheritance relationship between these classes is black and thick marking that the relationship is old. Class E is small and white, because it was recently introduced in the system.

— Class C was removed from the system and is colored in cyan. Class D has been introduced after several versions as a subclass of C, but in the last version it has become a subclass of A.

— Class B is small because there were no changes detected in it. Classes A and D have the same width, but class D appears to have less statements added or removed because the node is more wide than tall. Class C is much taller compared to its width, denoting a greater implementation effort.

Based on the visualization we can detect two more characteristics:

— *Heterogeneous*. We characterize a class hierarchy history as being heterogeneous if the class histories have a wide range of ages. Such a hierarchy will appear colored with a wide range of grays.

— *Unstable Root*. In a hierarchy with an unstable root, the root node is large compared with the rest of the nodes.

6.4 Validation

For our experiments we chose two open source systems: JBoss[1] and Jun[2].

JBoss. For the first case study, we chose 14 versions of JBoss. JBoss is an open source J2EE application server written in Java. The versions we selected for the experiments were at two months distance from one another starting from the beginning of 2001 until the end of 2003. The first version has 628 classes, the last one has 4975 classes.

Jun. As a second case study we selected 40 versions of Jun. Jun is a 3D-graphics framework written in Smalltalk currently consisting of more than 700 classes. As experimental data we took every 5th version starting from version 5 (the first public version) to version 200. The time distance between version 5 and version 200 is about two years, and the considered versions were released about 15-20 days apart. In terms of number of classes, in version 5 of Jun there are 170 classes while in version 200 there are 740 classes.

[1] See http://www.jboss.org.
[2] See http://www.srainc.com/Jun/.

Figure 6.3 (p.96) and Figure 6.4 (p.97) show some of the results we obtained on six large hierarchies of both case studies. The results are accompanied by the visualizations.

6.4.1 Class Hierarchies of JBoss

Figure 6.2 (p.95) shows the largest hierarchy in JBoss (ServiceMBeanSupport), and Figure 6.3 (p.96) shows five other hierarchies from JBoss: JBossTestCase, Stats, J2EEManagedObject, MetaData and SimpleNode (we name the hierarchies according to the names of their root classes). On the bottom part of the figure we show the characterization of each hierarchy according to the proposed characteristics. For space reasons Figure 6.2 (p.95) is scaled with a 0.5 ratio (*i.e.,* zoomed out) as compared with the Figure 6.3 (p.96).

ServiceMBeanSupport is a large hierarchy with nodes and edges of different colors and shapes: As a whole, we classify it as heterogeneous from the age point of view, fragile from the inheritance relationship point of view, and unstable and unbalanced from the changes point of view.

The J2EEManagedObject is a sub-hierarchy of ServiceMBeanSupport and is heterogeneous and unbalanced from the point of view of performed changes and is fragile in terms of inheritance. In the visualization the hierarchy is displayed with nodes colored in various grays and with many cyan edges and nodes.

JBossTestCase is composed of classes of different age. On average, the nodes are rather small, meaning that the classes were stable. The hierarchy is heterogeneous from the class age point of view which shows that the tests were continuously developed along the project. Also, because most of the nodes are small it means that most of the classes are stable both from the methods and from the implementation point of view. In the unstable classes there were more implementation changes than methods added or removed. Also, once the test classes were created not many test methods were added or removed afterwards.

The Stats hierarchy has been recently introduced and did not yet experience major changes.

MetaData is represented with nodes of different sizes colored either in dark colors or in cyan: It is an old hierarchy which is unstable and unbalanced from the performed changes point of view. The edges are either thick and dark or cyan, and as the number of removed edges are not high, we characterize the inheritance relationships as being solid.

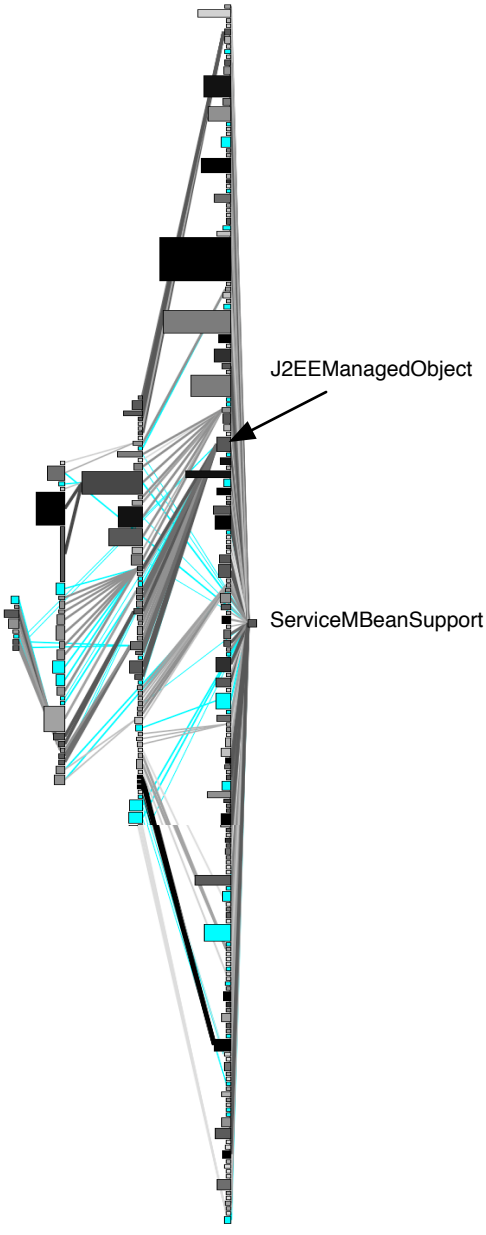
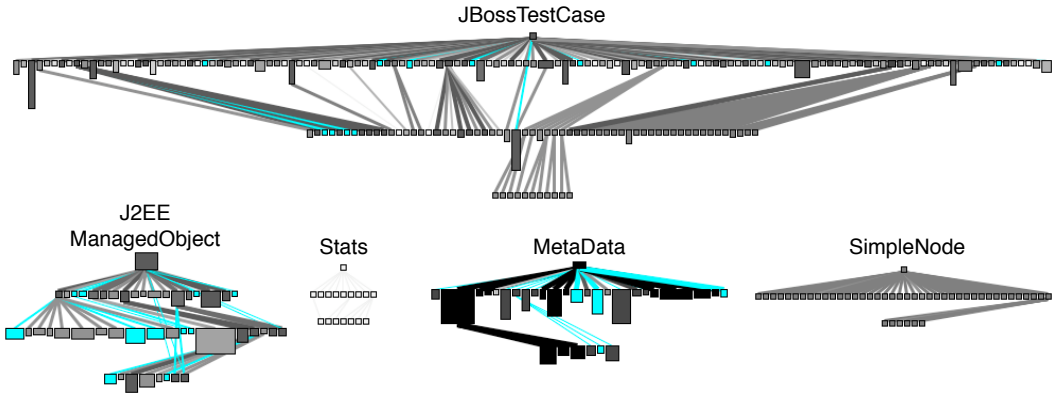


Figure 6.2: A *Hierarchy Evolution Complexity View* of the evolution of the largest hierarchy from 14 versions of JBoss.



Hierarchy	AGE	Inheritance	Stability	Change Balance
ServiceMBeanSupport	Old, Heterogeneous	Fragile	Unstable	Unbalanced
JBossTestCase	Old, Heterogeneous	Solid	Stable	Balanced
J2EEManagedObject	Old, Heterogeneous	Fragile	Unstable	Unbalanced
Stats	Newborn	Solid	Stable	Balanced
MetaData	Old	Solid	Unstable	Unbalanced
Simple Node	Old	Solid	Stable	Balanced

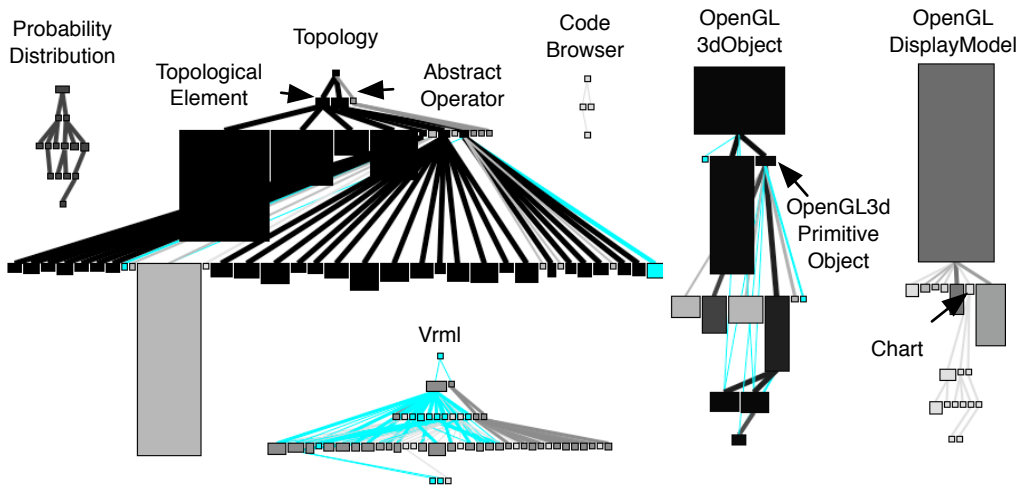
Figure 6.3: A *Hierarchy Evolution Complexity View* of the evolution of five hierarchies from 14 versions of JBoss.

The SimpleNode hierarchy is fairly old and experienced very few changes during its lifetime, making it thus a very stable hierarchy.

6.4.2 Class Hierarchies of Jun

Figure 6.4 (p.97) shows six of the hierarchies of Jun: Topology, CodeBrowser, OpenGL3dObject, Vrml, OpenGLDisplayModel, and ProbabilityDistribution. We accompany the visualization with the characterization of each hierarchy according to the proposed characteristics.

The Topology hierarchy is the largest and oldest hierarchy in the system. In the figure, we marked the two sub-hierarchies: AbstractOperator and TopologicalElement. The TopologicalElement sub-hierarchy is composed of classes which were changed



Hierarchy	AGE	Inheritance	Stability	Change Balance
Topology	Old	Solid	Unstable	Unbalanced
CodeBrowser	Newborn	Solid	Stable	Balanced
OpenGL3dObject	Old	Fragile	Unstable	Unbalanced, Unstable Root
Vrml	Persistent	Fragile	Stable	Balanced
OpenGLDisplayModel	Old	Solid	Stable	Balanced, Unstable Root
ProbabilityDistribution	Old	Solid	Stable	Balanced

Figure 6.4: A *Hierarchy Evolution Complexity View* of the evolution of six hierarchies from the 40 versions of Jun.

a lot during their life time. Three of the leaf classes were detected as being *God Classes* (see Chapter 5 (p.67)). A large part of the AbstractOperator hierarchy has been in the system from the first version, but there is a young sub-hierarchy which looks different.

The CodeBrowser hierarchy is thin and lightly colored, meaning that it has been recently added to the system.

The OpenGL3dObject hierarchy experienced three times an insertion of a class in the middle of the hierarchy: There are removed inheritance relationships colored in cyan.

The Vrml hierarchy proved to have undergone extensive renaming refactorings: We

see many removed nodes and removed inheritance relationships. Even the root class has been removed at a certain point in time: The original hierarchy has thus been split into two distinct hierarchies.

The OpenGLDisplayModel hierarchy has an old and unstable root. This is denoted by a large rectangle colored in dark gray. The Chart sub-hierarchy is thin and lightly colored denoting it is newborn.

ProbabilityDistribution is an old hierarchy and very stable from the inheritance relationships point of view. Also, the classes in the hierarchy were changed very little during their history.

6.5 Variation Points

On the impact of the release period

A variation point when computing *E* measurement is the release period. If, for example, we consider the release period of one week, we focus the analysis to immediate changes. If we consider the release period of half a year, we emphasize the size of the changes that accumulate in the class histories.

On the impact of the number of analyzed versions

The number of versions is another variation point when computing the measurements. By increasing the number of analyzed versions we obtain a long-term indicator of effort, while by decreasing the number of versions we concentrate on the short-term indicator of effort.

On the impact of the threshold values

Changing the threshold used for characterizing the evolution is also a variation point. For example, instead of using 10% of the total number of system versions for qualifying a class hierarchy as young, we can use 3 versions as the threshold (that is the classes should be introduced no later than 2 versions ago).

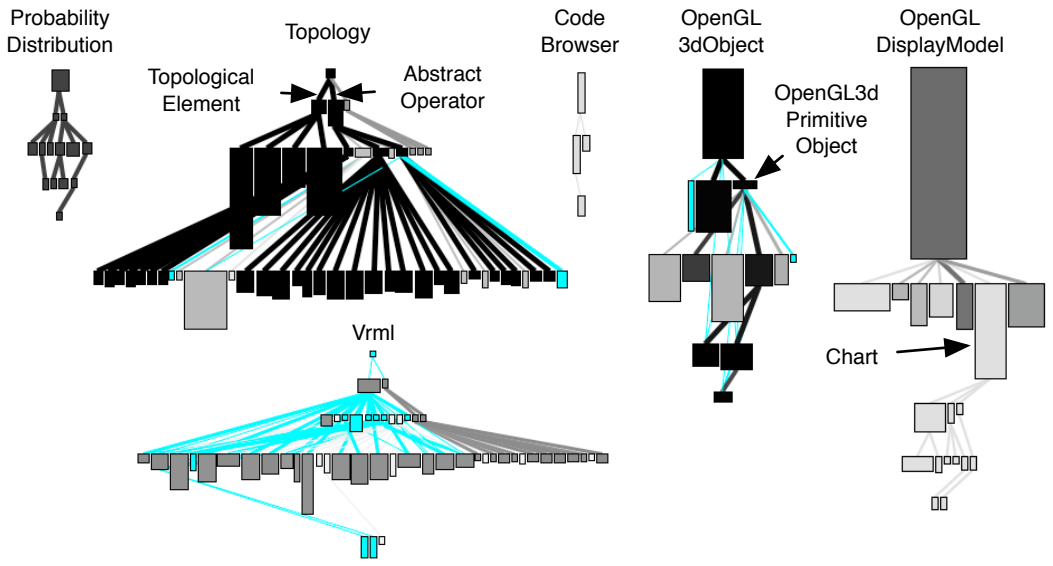


Figure 6.5: A modified *Hierarchy Evolution Complexity View* of the evolution of six hierarchies from the Jun case study. The node width is given by the instability of number of methods and the height is given by the last number of methods.

On the variations of the visualization

In our visualization we sought answers to four questions regarding the age of the hierarchy, the inheritance relationship stability, the class size stability and the change balance. The purpose of the visualization is to provide an overview of the evolution of hierarchies, but is of limited use when a deep understanding is required.

However, we are convinced that the information that one can extract from the analysis we propose and the use of an evolutionary polymetric view such as the *Hierarchy Evolution Complexity View* is useful: It reveals information about the system which would be otherwise difficult to extract (*e.g.,* knowing that a hierarchy is stable/unstable in time is valuable for deciding maintenance effort and doing quality assessment). In addition, we have to stress that polymetric views, as we implement them, are intrinsically interactive and that just looking at the visualization is only of limited value. Indeed, the viewer must interact with the visualization to extract finer-grained and more useful information, *e.g.,* accessing

the source code.

Example. In Figure 6.4 (p.97) one would like to know what class has been removed from the Topology hierarchy and also why, since this quite large hierarchy has been very stable in terms of inheritance relationships. The viewer can do so by pointing and inspecting the cyan class history node.

Figure 6.5 (p.99) shows a modified *Hierarchy Evolution Complexity View* applied on the Jun hierarchies. In this view we used for the width and the height of the nodes other measurements, namely the last number of methods (NOM) as the height and the instability of the number of methods (INOM) as the width. The instability of the number of methods is computed as the number of versions in which the number of methods changed over the total number of versions in which it could have changed.

While in the original view the nodes dimensions show the correlation between implementation effort, and the behavior addition and removal effort, this view shows the correlation of the actual size in terms of number of methods and the number of times methods were added or removed in the classes. Thus, in this view we can detect whether the instability is correlated with actual size.

In Figure 6.5 (p.99), the CodeBrowser hierarchy, is newborn and appears small in the original view, but the modified view shows that the classes are among the tallest displayed, meaning that the classes are rather big in size, but there were no changes in them.

In the Topology hierarchy the TopologicalElement hierarchy appeared to be unstable in the original view, but in the modified view it appears as the classes were not changed many times, because the nodes are not very wide. The rest of the Topology hierarchy is stable, except for one class which, even if it is younger than the rest, was changed many times.

In the OpenGLDisplayModel hierarchy there is a correlation of the instability of number of methods and the size of the root class because the node is tall and wide. On the other hand, the Chart hierarchy is newborn and appears small in the original view, while in this view the root is tall meaning that it is a large class.

The OpenGL3dObject hierarchy also has a large and unstable root. The OpenGL3d-PrimitiveObject class appears as being small in size, but it changed its number of methods many times, because it is not tall, but it is wide. Also, its direct subclasses have about the same width, but their height differs. This means that

those classes changed their number of methods about the same amount of times although their actual size differs.

The classes in the ProbabilityDistribution hierarchy appear small and stable. In the Vrml hierarchy the size of the classes vary, but the classes are stable.

6.6 Related Work

Burd and Munro analyzed the influence of changes on the maintainability of software systems. They defined a set of measurements to quantify the dominance relations which are used to depict the complexity of the calls [Burd and Munro, 1999].

Lanza's Evolution Matrix visualized the system's history in a matrix in which each row is the history of a class [Lanza and Ducasse, 2002]. A cell in the Evolution Matrix represents a class and the dimensions of the cell are given by evolutionary measurements computed on subsequent versions. Jazayeri analyzed the stability of the architecture by using colors to depict the changes [Jazayeri, 2002]. Jingwei Wu *et al.* used the spectograph metaphor to visualize how changes occur in software systems [Wu *et al.*, 2004a]. Rysselberghe and Demeyer use a simple visualization based on information in version control systems to provide an overview of the evolution of systems [Van Rysselberghe and Demeyer, 2004].

Our approach differs from the above mentioned ones because we consider history to be a first class entity and define history measurements which are applied on the whole history of an entity and which summarize the evolution of that entity. The authors already used the notion of history to analyze how changes appear in the software systems [Gîrba *et al.*, 2004a]. The drawback of our approach consists in the inherent noise which resides in compressing large amounts of data into numbers.

Taylor and Munro visualized CVS data with a technique called *revision towers* [Taylor and Munro, 2002]. Ball and Eick developed visualizations for showing changes that appear in the source code [Ball and Eick, 1996]. These approaches reside at a different granularity level, *i.e.,* files, and thus does not display source code artifacts as in our approach.

Gulla proposed different visualizations of C code versions, but to our knowledge there was no implementation [Gulla, 1992]. For example, he displayed several versions of the structure and let the user identify the changes, by superimposing the

figures. Holt and Pak proposed a detailed visualization of the old and new dependencies between modules [Holt and Pak, 1996]. Collberg *et al.* used graph-based visualizations to display the changes authors make to class hierarchies [Collberg *et al.*, 2003]. However, they did not give any representation of the dimension of the effort and of the removals of entities.

Another metrics-based approach to detect refactorings of classes was developed by Demeyer *et al.* [Demeyer *et al.*, 2000]. While they focused on detecting refactorings, we focus on offering means to understand where and how the development effort was spent in a hierarchy.

6.7 Summary of the Approach

This approach set to answer four questions: (1) How old are the classes of a hierarchy?, (2) Were there changes in the inheritance relationship?, (3) Are classes from one hierarchy modified more than those from another one?, and (4) Are the changes evenly distributed among the classes of a hierarchy?

The history of a system holds the information necessary for answering the above questions, but the analysis is difficult due to the large amount of data. We approached this problem by defining the history as a first class entity and then we defined history measurements which summarize the evolution of an entity.

Based on the questions we formulated a vocabulary of terms and we used the measurements to formulate rules to characterize the evolution of class hierachies. Furthermore, we displayed the results using a new polymetric view of the evolution of class hierarchies called *Hierarchy Evolution Complexity View*.

We applied our approach on two large open source projects and showed how we could describe the evolution of class hierarchies.

In the future, we want to investigate possibilities of using other measurements and of adding more semantic information to the view we propose. For example, we want to add information like refactorings that have been performed.

6.8 Hismo Validation

The presented approach exercises several characteristics of Hismo: it combines historical properties with historical relationships.

Originally, polymetric views were only used for analyzing structural information (*e.g.,* the hierarchy of classes). Figure 6.1 (p.92) shows an example of how the evolution over 5 versions of a hierarchy (left side) is summarized in one polymetric view (right side). On the left hand side we represent the hierarchy structure in each version – *i.e.,* classes as nodes and inheritance relationships as edges. On the right hand side we display ClassHistories as nodes and InheritanceHistories as edges. The color of each node represents the age of the ClassHistory: the darker the node the older the ClassHistory. The size of the node denotes how much it was changed: the larger the node, the more the ClassHistory was changed. Both the thickness and the color of the edge represent the age of the InheritanceHistory. Furthermore, the cyan color denotes removed histories.

Below we show how the queries presented can be expressed with Hismo. For example, the Newborn query:

$$Avg(ClassAge, Ch) < 0.1 * SystemAge$$

. . . can be expressed like:

context HistoryGroup

-- returns true if the average age of the class histories is lower than 10% of the system age
derive newborn:
self.averageOfAge < 0.1 * self.referenceHistory->age

Chapter 7

How Developers Drive Software Evolution

Manipulating Historical Relationships

In the end, no two things are truly independent.

As systems evolve their structure change in ways not expected upfront, and the knowledge of the developers becomes more and more critical for the process of understanding the system. That is, when we want to understand a certain issue of the system we ask the knowledgeable developers. Yet, in large systems, not every developer is knowledgeable in all the details of the system. Thus, we would want to know which developer is knowledgeable in the issue at hand. In this chapter we make use of the mapping between the changes and the author identifiers provided by versioning repositories to recover their zones and periods of influence in the system. We first define a measurement for the notion of code ownership. We use this measurement to define the Ownership Map *visualization to understand when and how different developers interacted in which way and in which part of the system. We report the results we obtained on several large systems.*

The Ownership Map *visualization is an example of how to use historical relationships.*

7.1 Introduction

Software systems need to change in ways that challenge the original design. Even if the original documentation exists, it might not reflect the code anymore. In such situations, it is crucial to get access to developer knowledge to understand the system. As systems grow larger, not all developers know about the entire system, thus, to make the best use of developer knowledge, we need to know which developer is knowledgeable in which part of the system.

From another perspective, Conway's law states that "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" [Conway, 1968]. That is, the shape of the organization reflects on the shape of the system. As such, to understand the system, one also has to understand the interaction between the developers and the system [Demeyer *et al.*, 2002].

In this chapter we aim to understand how the developers drove the evolution of the system. In particular we provide answers to the following questions:

— How many authors developed the system?

— Which author developed which part of the system and in which period?

— What were the behaviors of the developers?

In our approach, we assume that the original developer of a line of code is the most knowledgeable in that line of code. We use this assumption to determine the owner of a piece of code (*e.g.,* a file) as being the developer that owns the largest part of that piece of code. We make use of the ownership to provide a visualization that helps to understand how developers interacted with the system. The visualization represents files as lines, and colors these lines according to the ownership over time.

Contrary to similar approaches (*e.g.,* [Van Rysselberghe and Demeyer, 2004]), we give a semantic order to the file axis (*i.e.,* we do not rely on the names of the files) by clustering the files based on their history of changes: files committed in the same period are related [Gall *et al.*, 1998].

We implemented our approach in Chronia, a tool built on top of the Moose reengineering environment [Ducasse *et al.*, 2005]. As CVS is a de facto versioning system, our implementation relies on the CVS model. Our aim was to provide a solution that gives fast results, therefore, our approach relies only on information

from the CVS log without checking out the whole repository. As a consequence, we can analyze large systems in a very short period of time, making the approach usable in the very early stages of reverse engineering.

To show the usefulness of our solution we applied it on several large case studies. We report here some of the findings and discuss different facets of the approach.

Structure of the Chapter

In Section 7.2 (p.107) we define how we measure the code ownership. In Section 7.3 (p.109), we use this measurement to introduce our *Ownership Map* visualization of how developers changed the system. Section 7.4 (p.113) shows the results we obtained on several large case studies, and Section 7.5 (p.120) discusses the approach including details of the implementation. Section 7.6 (p.121) presents the related work. We conclude and present the future work in Section 7.7 (p.123), and we discuss the approach from the point of view of Hismo in Section 7.8 (p.124).

7.2 Data Extraction From the CVS Log

This section introduces a measurement to characterize the code ownership. The assumption is that the original developer of a line of code is the most knowledgeable in that line of code. Based on this assumption, we determine the owner of a piece of code as being the developer that owns the most lines of that piece of code.

The straightforward approach is to checkout all file versions ever committed to the versioning repository and to compute the code ownership from diff information between each subsequent revisions f_{n-1} and f_n. From an implementation point of view this implies the transfer of large amounts of data over the network, and long computations.

We aim to provide for an approach that can provide the results fast on projects which are large and have a long history. As CVS is a very common versioning system, we tuned our approach to work with the information CVS can provide. In particular we compute the ownership of the code based only on the CVS log information.

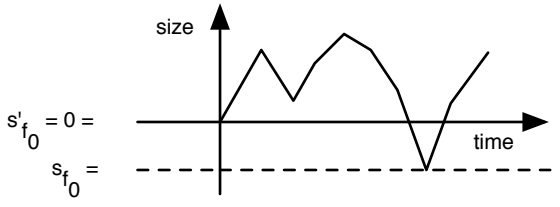
Below we present a snippet from a CVS log. The log lists for each version f_n of a file, the time t_{f_n} of its commit, the name of its author α_{f_n}, some state information and finally the number of added and removed lines as deltas a_{f_n} and r_{f_n}. We use these numbers to recover both the file size s_{f_n}, and the code ownership $own^{\alpha}_{f_n}$.

```
----------------------------
revision 1.38
date: 2005/04/20 13:11:24;  author: girba;  state: Exp;  lines: +36 -11
added implementation section
----------------------------
revision 1.37
date: 2005/04/20 11:45:22;  author: akuhn;  state: Exp;  lines: +4 -5
fixed errors in ownership formula
----------------------------
revision 1.36
date: 2005/04/20 07:49:58;  author: mseeberg;  state: Exp;  lines: +16 -16
Fixed math to get pdflatex through without errors.
----------------------------
```

Figure 7.1: Snapshot from a CVS log.

7.2.1 Measuring File Size

Let s_{f_n} be the size of revision f_n, measured in number of lines. The number of lines is not given in the CVS log, but can be computed from the deltas a_{f_n} and r_{f_n} of added and removed lines. Even though the CVS log does not give the initial size s_{f_0}, we can give an estimate based on the fact that one cannot remove more lines from a file than were ever contained. We define s_{f_n} as in Figure 7.2 (p.108): we first calculate the sizes starting with an initial size of 0, and then in a second pass adjust the values with the lowest value encountered in the first pass.



$$s'_{f_0} := 0$$
$$s'_{f_n} := s'_{f_{n-1}} + a_{f_{n-1}} - r_{f_n}$$
$$s_{f_0} := |min\{s'_x\}|$$
$$s_{f_n} := s_{f_{n-1}} + a_{f_n} - r_{f_n}$$

Figure 7.2: The computation of the initial size.

This is a pessimistic estimate, since lines that never changed are not covered by the deltas in the CVS log. This is an acceptable assumption since our main focus

is telling the story of the developers, not measuring lines that were never touched by a developer. Furthermore in a long-living system the content of files is entirely replaced or rewritten at least once if not several times. Thus the estimate matches the correct size of most files.

7.2.2 Measuring Code Ownership

A developer owns a line of code if he was the last one that committed a change to that line. In the same way, we define file ownership as the percentage of lines he owns in a file. And the overall owner of a file is the developer that owns the largest part of it.

Let $own^{\alpha}_{f_n}$ be the percentage of lines in revision f_n owned by author α. Given the file size s_{f_n}, and both the author α_{f_n} that committed the change and a_{f_n} the number of lines he added, we defined ownership as:

$$own^{\alpha}_{f_0} := \begin{cases} 1, & \alpha = \alpha_{f_0} \\ 0, & else \end{cases}$$

$$own^{\alpha}_{f_n} := own^{\alpha}_{f_{n-1}} \frac{s_{f_n} - a_{f_n}}{s_{f_n}} + \begin{cases} \frac{a_{f_n}}{s_{f_n}}, & \alpha = \alpha_{f_n} \\ 0, & else \end{cases}$$

In the definition we assume that the removed lines r_{f_n} are evenly distributed over the ownership of the predecessor developers f_{n-1}.

7.3 The Ownership Map View

We introduce a the *Ownership Map* visualization as in Figure 7.3 (p.110). The visualization is similar to the Evolution Matrix [Lanza and Ducasse, 2002]: each line represents a history of a file, and each circle on a line represents a change to that file.

The color of the circle denotes the author that made the change. The size of the circle reflects the proportion of the file that got changed *i.e.,* the larger the change, the larger the circle. And the color of the line denotes the author who owns most of the file.

Bertin assessed that one of the good practices in information visualization is to offer to the viewer visualizations that can be grasped at one glance [Bertin, 1974].

The colors used in our visualizations follow visual guidelines suggested by Bertin, Tufte and Ware - *e.g.*, we take into account that the human brain is capable of processing fewer than a dozen distinct colors [Tufte, 1990; Ware, 2000].

In a large system, we can have hundreds of developers. Because the human eye is not capable of distinguishing that many colors, we only display the authors who committed most of all changes using distinct colors; the remaining authors are represented in gray. Furthermore, we also represent with gray files that came into the CVS repository with the initial import, because these files are usually sources from another project with unknown authors and are thus not necessarily created by the author that performed the import. In short, a gray line represents either an unknown owner, or an unimportant one.

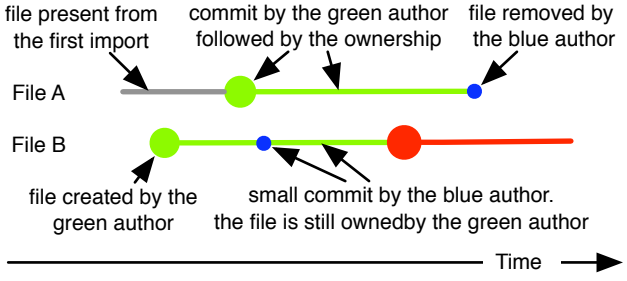


Figure 7.3: Example of ownership visualization of two files.

In the example from Figure 7.3 (p.110), each line represents the lifetime of a file; each circle represents a change. File A appears gray in the first part as it originates from the initial import. Later the green author significantly changed the file, and he became the owner of the file. In the end, the blue author deleted the file. File B was created by the green author. Afterwards, the blue author changed the file, but still the green author owned the largest part, so the line remains green. At some point, the red author committed a large change and took over the ownership. The file was not deleted.

7.3.1 Ordering the Axes

Ordering the Time Axis. Subsequent file revisions committed by the same author are grouped together to form a transaction of changes *i.e.*, a commit. We use a single linkage clustering with a threshold of 180 seconds to obtain these groups. This solution is similar to the sliding time window approach of Zimmerman *et al.*

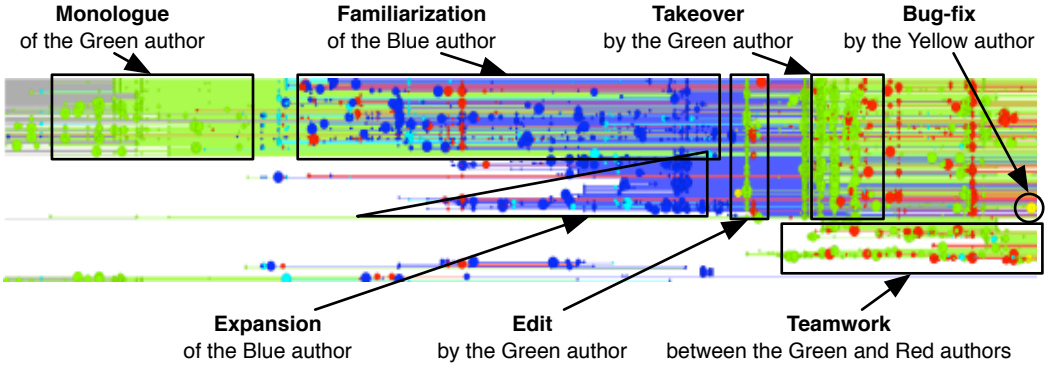


Figure 7.4: Example of the Ownership Map view. The view reveals different patterns: Monologue, Familiarization, Edit, Takeover, Teamwork, Bug-fix.

when they analyzed co-changes in the system [Zimmermann *et al.*, 2004]. The difference is that in our approach the revisions in a commit do not have to have the same log comment, thus any quick subsequent revisions by the same author are grouped into one commit.

Ordering the Files Axis. A system may contain thousands of files; furthermore, an author might change multiple files that are not near each other if we would represent the files in an alphabetical order. Likewise, it is important to keep an overview of the big parts of the system. Thus, we need an order that groups files with co-occurring changes near each other, while still preserving the overall structure of the system. To meet this requirement we split the system into high-level modules (*e.g.*, the top level folders), and order inside each module the files by the similarity of their history. To order the files in a meaningful way, we define a distance metric between the commit signature of files and order the files based on a hierarchical clustering.

Let H_f be the commit signature of a file, a set with all timestamps t_{f_n} of each of its revisions f_n. Based on this the distance between two commit signatures H_a and H_b can be defined as the modified Hausdorff distance [1] $\delta(H_a, H_b)$:

$$D(H_n, H_m) := \sum_{n \in H_n} min^2\{|m - n| : m \in H_m\}$$

[1]The Hausdorff metric is named after the german mathematician Felix Hausdorff (1868-1942) and is used to measure the distance between two sets with elements from a metric space.

$$\delta(H_a, H_b) := max\{D(H_a, H_b), D(H_b, H_a)\}$$

With this metric we can order the files according to the result of a hierarchical clustering algorithm [Jain *et al.*, 1999]. From this algorithm a dendrogram can be built: this is a hierarchical tree with clusters as its nodes and the files as its leaves. Traversing this tree and collecting its leaves yields an ordering that places files with similar histories near each other and files with dissimilar histories far apart from each other.

The files axes of the *Ownership Map* views are ordered with *average linkage* clustering and *larger-clusters-first* tree traversal. Nevertheless, our tool Chronia allows customization of the ordering.

7.3.2 Behavioral Patterns

The Overview Map reveals semantic information about the work of the developer. Figure 7.4 (p.111) shows a part of the *Ownership Map* of the Outsight case study (for more details see Section 7.4.1 (p.114)). In this view we can identify several different behavioral patterns of the developers:

Monologue. Monologue denotes a period during which all changes and most files belong to the same author. It shows on a *Ownership Map* as a unicolored rectangle with change circles in the same color.

Dialogue. As opposed to Monologue, Dialogue denotes a period with changes done by multiple authors and mixed code ownership. On a *Ownership Map* it is denoted by rectangles filled with circles and lines in different colors.

Teamwork. Teamwork is a special case of Dialogue, where two or more developers commit a quick succession of changes to multiple files. On a *Ownership Map* it shows as circles of alternating colors looking like a bunch of bubbles. In our example, we see in the bottom right part of the figure a collaboration between Red and Green.

Silence. Silence denotes an uneventful period with nearly no changes at all. It is visible on an *Ownership Map* as a rectangle with constant line colors and none or just few change circles.

Takeover. Takeover denotes a behavior where a developer takes over a large amount of code in a short amount of time - *i.e.,* the developer seizes ownership of a subsystem in a few commits. It is visible on a *Ownership Map* as a

vertical stripe of single color circles together with an ensuing change of the lines to that color. A Takeover is commonly followed by subsequent changes done by the same author. If a Takeover marks a transition from activity to Silence we classify it as an *Epilogue*.

Familiarization. As opposed to Takeover, Familiarization characterizes an accommodation over a longer period of time. The developer applies selective and small changes to foreign code, resulting in a slow but steady acquisition of the subsystem. In our example, Blue started to work on code originally owned by Green, until he finally took over ownership.

Expansion. Not only changes to existing files are important, but also the expansion of the system by adding new files. In our example, after Blue familiarized himself with the code, he began to extend the system with new files.

Cleaning. Cleaning is the opposite of expansion as it denotes an author that removes a part of the system. We do not see this behavior in the example.

Bugfix. By bug fix we denote a small, localized change that does not affect the ownership of the file. On a *Ownership Map* it shows as a sole circle in a color differing from its surrounding.

Edit. Not every change necessarily fulfills a functional role. For example, cleaning the comments, changing the names of identifiers to conform to a naming convention, or reshaping the code are sanity actions that are necessary but do not add functionality. We call such an action *Edit*, as it is similar to the work of a book editor. An Edit is visible on a *Ownership Map* as a vertical stripe of unicolored circles, but in difference to a Takeover neither the ownership is affected nor is it ensued by further changes by the same author. If an Edit marks a transition from activity to Silence we classify it as an *Epilogue*.

7.4 Validation

We applied our approach on several large case studies: Outsight, Ant, Tomcat, JEdit and JBoss. Due to the space limitations we report the details from the Outsight case study, and we give an overall impression on the other four well-known open source projects.

Outsight. Outsight is a commercial web application written in Java and JSP. The CVS repository goes back three years and spans across two development iterations

separated by half a year of maintenance. The system is written by four developers and has about 500 Java files and about 500 JSP files.

Open-source Case Studies. We choose Ant, Tomcat, JEdit, and JBoss to illustrate different fingerprints systems can have on an *Ownership Map*. Ant has about 4500 files, Tomcat about 1250 files, JEdit about 500 files, and JBoss about 2000 files. The CVS repository of each project goes back several years.

7.4.1 Outsight

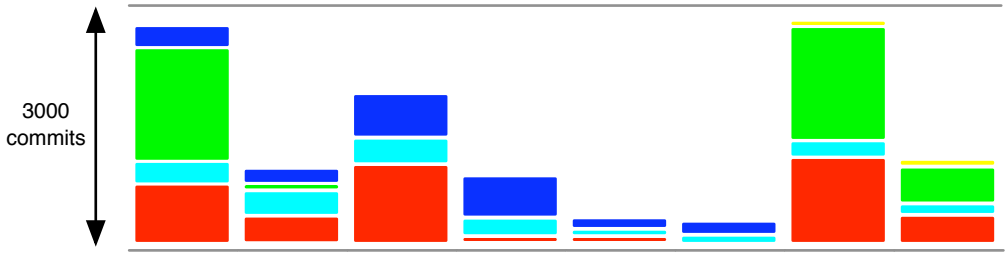
The first step to acquire an overview of a system is to build a histogram of the team to get an impression about the fluctuations of the team members over time. Figure 7.5 (p.115) shows that a team of four developers is working on the system. There is also a fifth author contributing with changes in the last two periods only.

Figure 7.6 (p.117) shows the *Ownership Map* of our case study. The upper half are Java files, the bottom half are JSP pages. The files of both modules are ordered according to the similarity of their commit signature. For the sake of readability we use S1 as a shorthand for the Java files part of the system, and S2 as a shorthand for the JSP files part. Time is cut into eight periods P1 to P8, each covering three months. The paragraphs below discuss each period in detail, and show how to read the *Ownership Map* in order to answer our initial questions.

The shorthands in paranthesis denote the labels R1 to R15 as given on Figure 7.6 (p.117).

Period 1. In this period four developers are working on the system. Their collaboration maps the separation of S1 and S2: while Green is working by himself on S2 (R5), the others are collaborating on S1. This is a good example of Monologue versus Dialogue. A closer look on S1 reveals two hotspots of Teamwork between Red and Cyan (R1,R3), as well as large mutations of the file structure. In the top part multiple Cleanings happen (R2), often accompanied by Expansions in the lower part.

Period 2. Green leaves the team and Blue takes over responsibility of S2. He starts doing this during a slow Familiarization period (R6) which lasts until the end of P3. In the meantime Red and Cyan continue their Teamwork on S1 (R4) and Red starts adding some files, which foreshadow the future Expansion in P3.



	P1	P2	P3	P4	P5	P6	P7	P8
Yellow	0	0	0	0	0	0	8	16
Blue	216	125	501	471	66	100	2	0
Green	1427	7	0	0	0	0	1431	401
Cyan	223	251	265	158	11	34	141	65
Red	710	278	969	3	5	0	1062	297

Figure 7.5: Number of commits per team member in periods of three months.

Period 3. This period is dominated by a big growth of the system, the number of files doubles as large Expansions happen in both S1 and S2. The histogram in Figure 7.5 (p.115) identifies Red as the main contributor. The Expansion of S1 evolves in sudden steps (R9), and as their file base grows the Teamwork between Red and Cyan becomes less tight. The Expansion of S2 happens in small steps (R8), as Blue continues to familiarize himself with S2 and slowly but steady takes over the ownership of most files in this subsystem (R6). Also an Edit of Red in S2 can be identified (R7).

Period 4. Activity moves down from S1 to S2, leaving S1 in a Silence only broken by selective changes. Table 7.5 (p.115) shows that Red left the team, which consists now of Cyan and Green only. Cyan acts as an allrounder providing changes to both S1 and S2, and Blue is further working on S2. The work of Blue culminates in an Epilogue marking the end of this period (R8). He has now completely taken over ownership of S2, while the ownership of subsystem S1 is shared between Red and Cyan.

Period 5 and 6. Starting with this period the system goes into maintenance. Only small changes occur, mainly by author Blue.

Period 7. After two periods of maintenance the team resumes work on the system. In Table 7.5 (p.115) we see how the composition of the team changed: Blue leaves and Green comes back. Green restarts with an Edit in S2 (R11), later

followed by a quick sequence of Takeovers (R13) and thus claiming back the ownership over his former code. Simultaneously, he starts expanding S2 in Teamwork with Red (R12).

First we find in S1 selective changes by Red and Cyan scattered over the subsystem, followed by a period of Silence, and culminating with a Takeover by Red in the end *i.e.,* an Epilogue (R14). The Takeover in S1 stretches down into S2, but there being a mere Edit. Furthermore we can identify two selective Bug-fixes (R10) by author Yellow who is a new team member.

Period 8. In this period, the main contributors are Red and Green: Red works in both S1 and S2, while Green remains true to S2. As Red mainly finished in the previous period his work on S1 with an Epilogue, his activity now moves down to S2. There we find an Edit (R15) as well as the continuation of the Teamwork between Red and Green (R12) in the Expansion started in P7. Yet again, as in the previous period, we find small Bug-fixes applied by Yellow.

To summarize these finding we give a description of each author's behavior, and in what part of the system he is knowledgeable.

Red author. Red is working mostly on S1, and acquires in the end some knowledge of S2. He commits some edits and may thus be a team member being responsible for ensuring code quality standards. As he owns a good part of S1 during the whole history and even closed that subsystem at the end of P7 with an Epilogue, he is the most knowledgeable developer in S1.

Cyan author. Cyan is the only developer that was in the team during all periods, thus he is the developer most familiar with the history of the system. He worked mostly on S1 and he owned large parts of this subsystem till the end of P7. His knowledge of S2 depends on the kind of changes Red introduced in his Epilogue. A quick look into the CVS log messages reveals that Red's Epilogue was in fact a larger than usual Edit and not a real Takeover: Cyan is as knowledgeable in S1 as Red.

Green author. Green only worked in S2, and he has only little impact on S1. He founded S2 with a Monologue, lost his ownership to Blue during P2 to P6, but in P7 he claimed back again the overall ownership of this subsystem. He is definitely the developer most knowledgeable with S2, being the main expert of this subsystem.

Blue author. Blue left the team after P4, thus he is not familiar with any changes applied since then. Furthermore, although he became an expert on S2 through

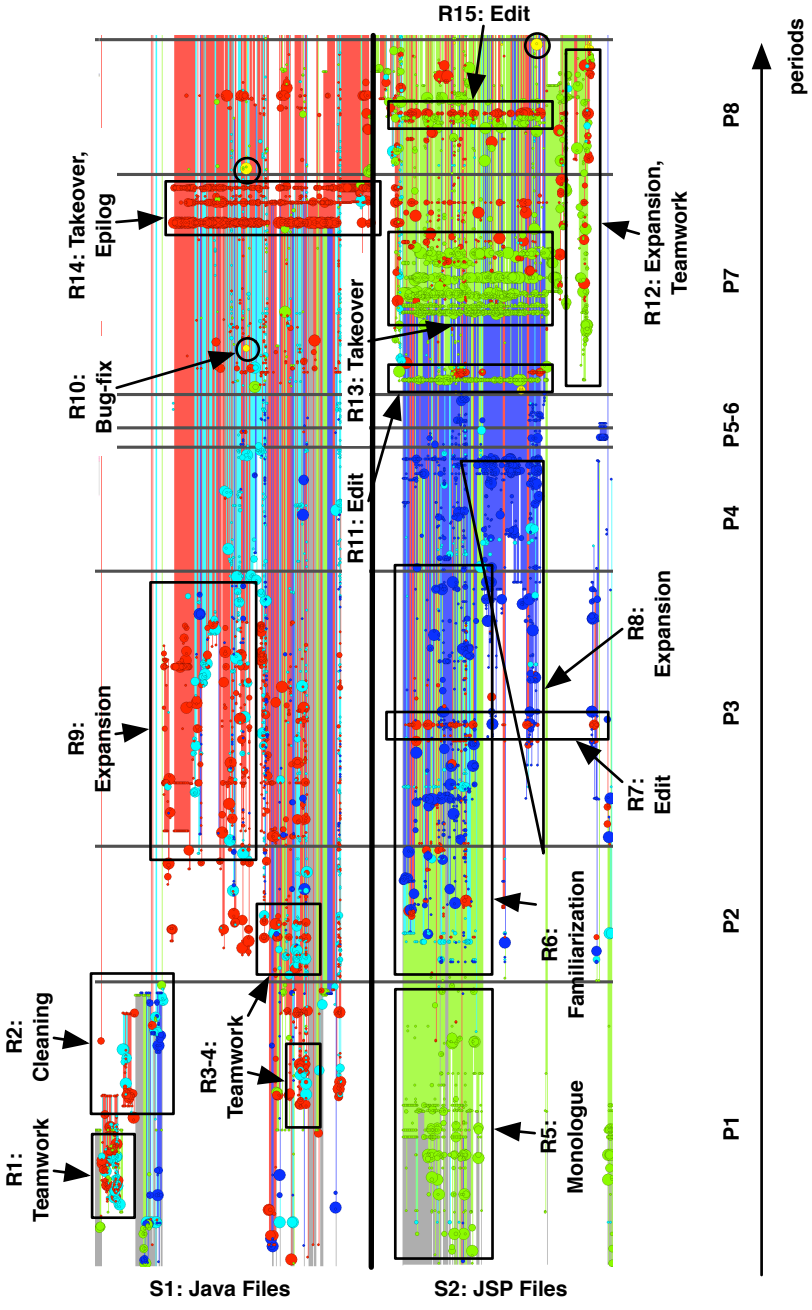


Figure 7.6: The Ownership Map of the Outsight case study.

Familiarization, his knowledge might be of little value since Green claimed that subsystem back with multiple Takeovers and many following changes.

Yellow author. Yellow is a pure Bug-fix provider.

7.4.2 Ant, Tomcat, JEdit and JBoss

Figure 7.6 (p.117) shows the *Ownership Map* of four open-source projects: Ant, Tomcat, JEdit, and JBoss. The views are plotted with the same parameters as the map in the previous case study, the only difference is that vertical lines slice the time axis into periods of twelve instead of three months. Ant has about 4500 files with 60000 revisions, Tomcat about 1250 files and 13000 revisions, JEdit about 500 files and 11000 revisions, and JBoss about 2000 files with 23000 revisions.

Each view shows a different but common pattern. The paragraphs below discuss each pattern briefly.

Ant. The view is dominated by a huge Expansion. After a development period, the very same files fall victim to a huge Cleaning. This pattern is found in many open-source projects: Developers start a new side-project and after growing up, it moves to an own repository, or the side-project ceases and is removed from the repository. In this case, the spin-off is the ceased Myrmidon project, a development effort as potential implementation of Ant2, a successor to Ant.

Tomcat. The colors in this view are, apart from some large blocks of Silence, well mixed. The *Ownership Map* shows much Dialogue and hotspots with Teamwork. Thus this project has developers that collaborate well.

JEdit. This view is dominated by one sole developer who is the driving force behind the project. This pattern is also often found in open-source projects: a single author contributes with about 80% of the code.

JBoss. The colors in this view indicate that the team underwent large fluctuations. We see twice a sudden change in the color of both commits and code ownership: once mid 2001 and once mid 2003. Both changes are accompained by Cleanings and Expansions. Thus the composition of the team changed twice significantly, and the new teams restructured the system.

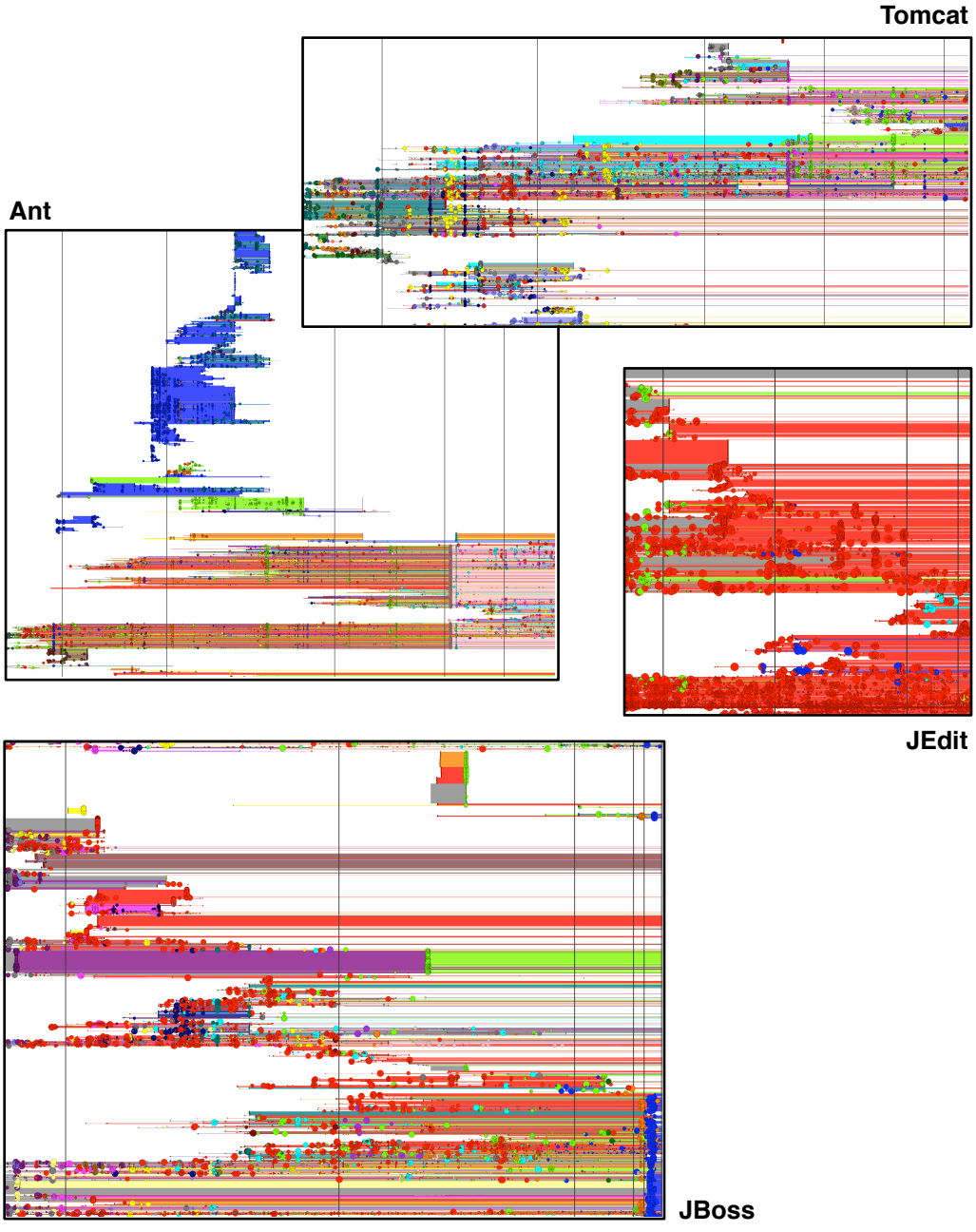


Figure 7.7: The Ownership Map of Ant, Tomcat, JEdit, and JBoss.

7.5 Variation Points

On the exploratory nature of the implementation

We implemented our approach in CHRONIA, a tool built on top of the MOOSE reengineering environment [Ducasse *et al.*, 2005]. Figure 9.5 (p.144) emphasizes the interactive nature of our tool.

On the left hand side of Figure 9.5 (p.144) we show CHRONIA visualizing the overall history of the project, which provides a first overview. Since there is too much data we cannot give the reasoning only from this view, thus, CHRONIA allows for interactive zooming. For example, in the window on the lower right, we see CHRONIA zoomed into the bottom right part of the original view. Furthermore, when moving the mouse over the *Ownership Map*, we complement the view by also highlighting in the lists on the right the current position on both time and file axis. These lists show all file names and the timestamps of all commits. As CHRONIA is built on top of MOOSE, it makes use of the MOOSE contextual menus to open detailed views on particular files, modules or authors. For example, in the top right window we see a view with metrics and measurements of a file revision.

On the scalability of the visualization

Although CHRONIA provides zooming interaction, one may lose the focus on the interesting project periods. A solution would be to further abstract the time and group commits to versions that cover longer time periods. The same applies to the file axis grouping related files into modules.

On the decision to rely on CVS log only

Our approach relies only on the information from the CVS log without checking out the whole repository. There are two main reasons for that decision.

First, we aim to provide a solution that gives fast results; *e.g.,* building the *Ownership Map* of JBoss takes 7,8 minutes on a regular 3 GHz Pentium 4 machine, including the time spent fetching the CVS log information from the *Apache.org* server.

Second, it is much easier to get access to closed source case studies from industry, when only metainformation is required and not the source code itself. We consider this an advantage of our approach.

On the shortcomings of CVS as a versioning system

As CVS lacks support for true file renaming or moving, this information is not recoverable without time consuming calculations. To move a file, one must remove it and add it later under another name. Our approach identifies the author doing the renaming as the new owner of the file, where in truth she only did rename it. For this reason, renaming directories impacts the computation of code ownership in a way not desired.

On the perspective of interpreting the *Ownership Map*

In our visualization we sought answers to questions regarding the developers and their behaviors. We analyzed the files from an author perspective point of view, and not from a file perspective point of view. Thus the *Ownership Map* tells the story of the developers and not of the files *e.g.,* concerning small commits: subsequent commits by different author to one file do not show up as a hotspot, while a commit by one author across multiple files does. The later being the pattern we termed *Edit*.

7.6 Related Work

Analyzing the way developers interact with the system has attracted only few research. A visualization similar to *Ownership Map* is used to visualize how authors change a wiki page [Viégas *et al.*, 2004].

Xiaomin Wu *et al.* visualized the change log information to provide an overview of the active places in the system as well as of the authors activity [Wu *et al.*, 2004b]. They display measurements like the number of times an author changed a file, or the date of the last commit.

Measurements and visualization have long been used to analyze how software systems evolve.

Ball and Eick developed multiple visualizations for showing changes that appear in the source code [Ball and Eick, 1996]. For example, they showed what is the percentage of bug fixes and feature addition in files, or which lines were changed recently.

Chuah and Eick proposed three visualizations for comparing and correlating different evolution information like the number of lines added, the errors recorded between versions, number of people working etc. [Chuah and Eick, 1998].

Rysselberghe and Demeyer used a scatter plot visualization of the changes to provide an overview of the evolution of systems and to detect patterns of change [Van Rysselberghe and Demeyer, 2004].

Jingwei Wu *et al.* used the spectrograph metaphor to visualize how changes occur in software systems [Wu *et al.*, 2004a]. They used colors to denote the age of changes on different parts of the systems.

Jazayeri analyzed the stability of the architecture by using colors to depict the changes [Jazayeri, 2002]. From the visualization he concluded that old parts tend to stabilize over time.

Lanza and Ducasse visualized the evolution of classes in the Evolution Matrix [Lanza and Ducasse, 2002]. Each class version is represented using a rectangle. The size of the rectangle is given by different measurements applied on the class version. From the visualization different evolution patterns can be detected such as continuous growth, growing and shrinking phases etc.

The difference between our approach and the above visualizations is that we do display the files as they appear in the system structure, but we order them based on the how they were committed to help the reverse engineer make out development phases. Furthermore, our analysis focuses on authors and not on the structure of the system.

Another relevant reverse engineering domain is the analysis of the co-change history.

Gall *et al.* aimed to detect logical coupling between parts of the system by identifying the parts of the system which change together [Gall *et al.*, 1998]. They used this information to define a coupling measurement based on the fact that the more times two modules were changed at the same time, the more they were coupled.

Zimmerman *et al.* aimed to provide mechanism to warn developers about the cor-

relation of changes between functions. The authors placed their analysis at the level of entities in the meta-model (*e.g.*, methods) [Zimmermann *et al.*, 2004]. The same authors defined a measurement of coupling based on co-changes [Zimmermann *et al.*, 2003].

Hassan *et al.* analyzed the types of data that are good predictors of change propagation, and came to the conclusion that historical co-change is a better mechanism than structural dependencies like call-graph [Hassan and Holt, 2004].

The difference between our approach and the co-change analysis is that we measure the distance between files based on the Hausdorf metric of the commit time stamps, and not only based on the exact co-change.

7.7 Summarizing the Approach

In this chapter we set out to understand how the developers drove the evolution of the system. In particular we ask the following questions:

— How many authors developed the system?

— Which author developed which part of the system?

— What were the behaviors of the developers?

To answer them, we define the *Ownership Map* visualization based on the notion of code ownership. In addition we semantically group files that have a similar *commit signature* leading to a visualization that is not based on alphabetical ordering of the files but on co-change relationships between the file histories. The *Ownership Map* helps in answering which author is knowledgeable in which part of the system and also reveals behavioral patterns. To show the usefulness we implemented the approach and applied it on several case studies. We reported some of the findings and we discussed the benefits and the limitations as we perceived them during the experiments.

In the future, we would like to investigate the application of the approach at other levels of abstraction besides files, and to take into consideration types of changes beyond just the change of a line of code.

7.8 Hismo Validation

The Hausdorf metric measures the distance between two sets from the same metric space. We used it to measure the distance between the time-stamps of the commits of different file histories. The smaller this distance is, the more similarly the two files were changed. We then used a clustering to group files that were changed in a similar way.

Below we give the distance between two file histories as expressed in Hismo:

```
context FileHistory

  -- returns the distance between the current file history and the parameter
  derive hausdorfDistanceTo(anotherHistory: FileHistory):
      self.versions->sum(eachVersion |
          anotherHistory->versions->min(anotherVersion |
              (anotherVersion.date - eachVersion.date ) * (anotherVersion.date - eachVersion.date)
          )
      )
```

We used this distance to cluster the histories, much in the same way as Gall *et al.* used the exact co-change [Gall *et al.*, 2003]. The modeling decisions are discussed in Section 3.7 (p.41).

Chapter 8

Detecting Co-Change Patterns

Detecting Historical Relationships Through Version Comparison

Correlations are where we see them.

Software systems need to change over time to cope with new requirements, and due to design decisions, the changes happen to crosscut the system's structure. Understanding how changes appear in the system can reveal hidden dependencies between different parts of the system. We propose the usage of concept analysis to identify the parts of the system that change in the same way and in the same time. We apply our approach at different levels of abstraction (i.e., method, class, package) and we detect fine grained changes (i.e., statements were added in a class, but no method was added there). Concept analysis is a technique that identifies entities that have the same properties, but it requires manual inspection due to the large candidates it detects. We propose a heuristic that dramatically eliminates the quality of the detection. We apply our approach on one large case study and show how we can identify hidden dependencies and detect bad smells.

We show how we detect co-change relationships by analyzing similar changes using detailed version comparison.

8.1 Introduction

Software systems need to change over time to cope with the new requirements [Lehman and Belady, 1985]. As the requirements happen to crosscut the system's structure, changes will have to be made in multiple places. Understanding how changes appear in the system is important for detecting hidden dependencies between its parts.

In the recent period work has been carried out to detect and interpret groups of software entities that change together [Gall *et al.*, 1998; Itkonen *et al.*, 2004; Zimmermann *et al.*, 2004]. Yet, the detection is based on change information concerning one property, and is mostly based on file level information.

We propose the use of formal concept analysis to detect groups of entities that changed in the same way in several versions. Formal concept analysis is a technique that identifies sets of elements with common properties based on an incidence table that specifies the elements and their properties [Ganter and Wille, 1999].

To identify how entities changed in the same way, we use historical measurements to detect changes between two versions. We build an Evolution Matrix [Lanza and Ducasse, 2002] annotated with the detected changes, and we use the matrix as an incidence table where history represents the elements and changed in version x represents the xth property of the element.

Also, for building the matrix of changes, we make use of logical expressions which combine properties with thresholds and which run on two versions of the system to detect interesting entities. In this way, we can detect changes that take into account several properties.

Example. ShotgunSurgery appears when every time we have to change a class, we also have to change a number of other classes [Fowler *et al.*, 1999]. We would suspect a group of classes of such a bad smell, when they repeatedly keep their external behavior constant and change the implementation. We can detect this kind of change in a class in the versions in which the number of methods did not change, while the number of statements changed.

We can apply our approach on any type of entities we have defined in the metamodel. In this chapter we show how we detect groups of packages, classes and methods.

Structure of the Chapter

In the next section we briefly define two generic historical measurements. We describe Formal Concept Analysis in a nutshell in Section 8.3 (p.128). We show how we use FCA to detect co-change patterns, and we show how we apply it on different levels of abstractions (Section 8.4 (p.129)). We discuss the results we obtained when applying our approach on a large open source case study (Section 8.5 (p.132)). In Section 8.7 (p.134) we conclude and present the future work, and in Section 8.8 (p.135) we discuss the approach from the point of view of Hismo.

8.2 History Measurements

We use two generic historical measurements to distinguish between different types of changes.

Addition of a Version Property (*A*). We define a generic measurement, called addition of a version property P, as the addition of that property between version $i-1$ and i of the history H:

$$(i > 1) \quad A_i(P,H) = \begin{cases} P_i(H) - P_{i-1}(H), & P_i(H) - P_{i-1}(H) > 0 \\ 0, & P_i(H) - P_{i-1}(H) \leqslant 0 \end{cases} \tag{8.1}$$

Evolution of a Version Property (*E*). We define a generic measurement, called evolution of a version property P, as being the absolute difference of that property between version $i-1$ and i:

$$(i > 1) \quad E_i(P,H) = |P_i(H) - P_{i-1}(H)| \tag{8.2}$$

We instantiate the above mentioned measurements by applying them on different version properties of different types of entities:

— Method: *NOS* (number of statements), *CYCLO* (McCabe cyclomatic number [McCabe, 1976]).

— Class: *NOM* (number of methods), *WNOC* (number of all subclasses).

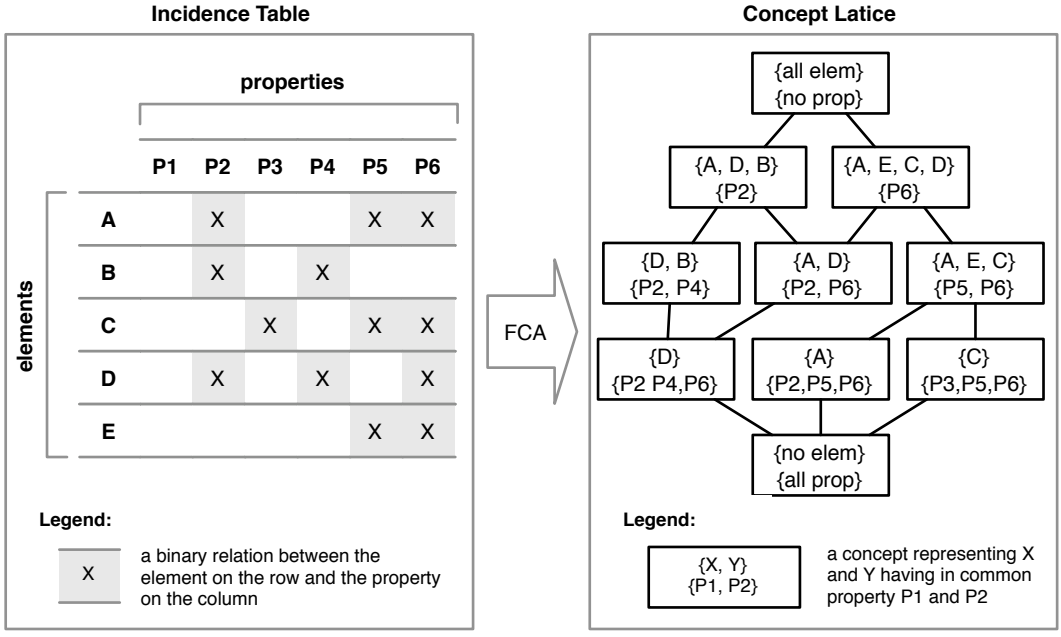


Figure 8.1: Example of applying formal concept analysis: the concepts on the right are obtained based on the incidence table on the left.

— Package: *NOCls* (number of classes), *NOM* (number of methods).

The *E* measurement shows a change of a certain property, while the *A* measurement shows the additions of a certain version property.

8.3 Concept Analysis in a Nutshell

Formal concept analysis is a technique that identifies meaningful groupings of elements that have common properties.

Figure 8.1 (p.128) gives a schematic example of the technique. The input is specified in the form of a so called incidence table which encodes binary relations between the set of elements and the set of properties. The output is a lattice of concepts, where each concept is a tuple of a set of elements and a set of common properties. For example, elements A and D have two properties in common: P2 and P6.

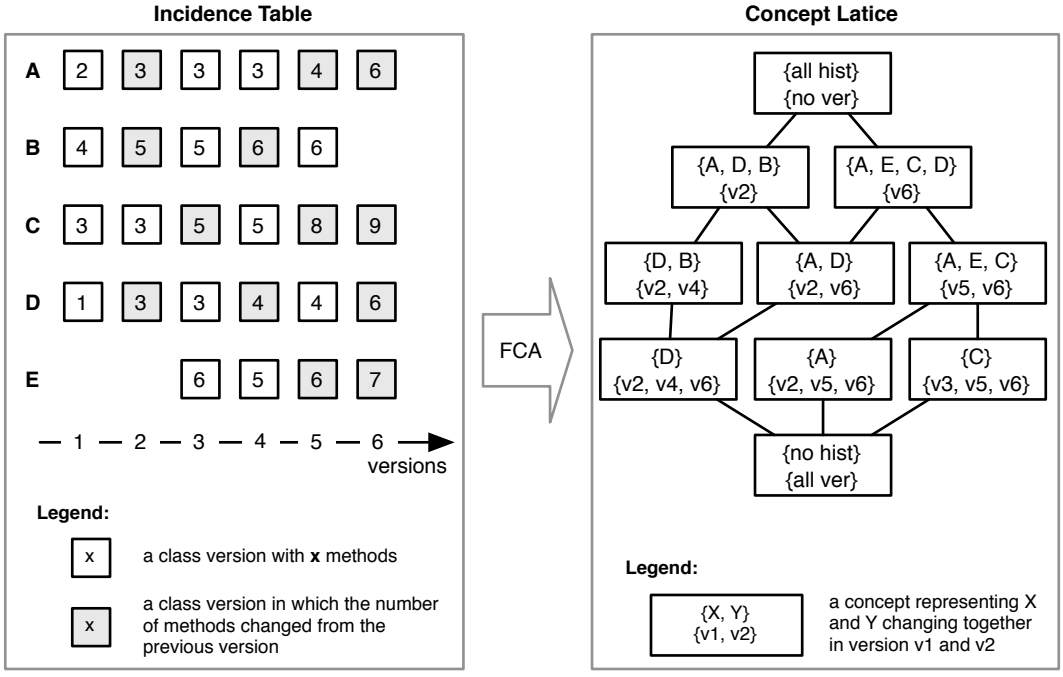


Figure 8.2: Example of applying concept analysis to group class histories based on the changes in number of methods. The Evolution Matrix on the left forms the incidence table where the property P_i of element X is given by "history X changed in version i."

Formal concept analysis is a generic technique working with elements and properties in general. To apply it in a particular context we need to map our interests on the elements and properties. We present in the next section the mapping we use to detect co-change patterns.

8.4 Using Concept Analysis to Identify Co-Change Patterns

On the left side of Figure 8.2 (p.129) we display an example of an Evolution Matrix in which each square represents a class version and the number inside a square represents the number of methods in that particular class version. A grayed square shows a change in the number of methods of a class version as compared with the

previous version ($E_i(NOM) > 0$).

We use the matrix as an incidence table, where the histories are the elements and the properties are given by "changed in version i". Based on such a matrix we can build a concept lattice. On the right side of figure we show the concept lattice obtained from the Evolution Matrix on the left.

Each concept in the lattice represents all the class histories which changed certain properties together in those particular versions. In the given example, class history A and D changed their number of methods in version 2 and version 6.

We do not only want to detect entities that change one certain property in the same time, but we want to detect entities that change several properties, and/or do not change other properties. For example, to detect parallel inheritances it is enough to just look at the number of children of classes; but, when we want to look for classes which need to change the internals of the methods in the same time without adding any new functionality, we need to look for classes which change their size, but not the number of methods.

We encode this change detection in expressions consisting of logical combination of historical measurements. These expressions are applied at every version on the last two versions. In the example from Figure 8.2 (p.129), we used as expression $E_i(NOM) > 0$ and we applied it on class histories.

In the following sections we introduce several expressions applicable on packages, classes and respectively methods.

8.4.1 Method Histories Grouping Expressions.

Parallel Complexity. A set of methods are effected by *Parallel Complexity* when a change in the complexity in one method involves changes in the complexity of other methods. As a measure of complexity we used the McCabe cyclomatic number. Classes with parallel complexity could reveal parallel conditionals.

$$ParallelComplexity : (A_i(CYCLO) > 0) \tag{8.3}$$

Parallel Bugs. We name a change a bug fix, when no complexity is added to the method, but the implementation changes. When we detect such repetitive bug

fixes in more methods in the same versions, we group those methods in a *Parallel Bugs* group. Such a group, might give indications of similar implementation which could be factored out. As an implementation measure we used number of statements.

$$ParallelBugs : (E_i(NOS) > 0) \wedge E_i(CYCLO) = 0) \tag{8.4}$$

8.4.2 Class Histories Grouping Expressions

Shotgun Surgery. The *Shotgun Surgery* bad-smell is encountered when a change operated in a class involves a lot of small changes to a lot of different classes [Fowler *et al.*, 1999]. We detect this bad smell, by looking at the classes which do not change their interface, but change their implementation together.

$$ShotgunSurgery = (E_i(NOM) = 0 \wedge E_i(NOS) > 0) \tag{8.5}$$

Parallel Inheritance. *Parallel Inheritance* is detected in the classes which change their number of children together [Fowler *et al.*, 1999]. Such a characteristic is not necessary a bad smell, but gives indications of a hidden link between two hierarchies. For example, if we detect a main hierarchy and a test hierarchy as being parallel, it gives us indication that the tests were developed in parallel with the code.

$$ParallelInheritance = (A_i(WNOC) > 0) \tag{8.6}$$

Parallel Semantics. Methods specify the semantics of a class. With *Parallel Semantics* we detect classes which add methods in parallel. Such a characteristic could reveal hidden dependencies between classes.

$$ParallelSemantics = (A_i(NOM) > 0) \tag{8.7}$$

8.4.3 Package Histories Grouping Expression

Package Parallel Semantics. If a group of classes is detected, as having parallel semantics, we would want to relate the containing packages as well. *Package Parallel Semantics* detects packages in which some methods have been added, but no classes have been added or removed.

$$PackageParallelSemantics = (E_i(NOCls) = 0) \wedge (A_i(NOM) > 0) \tag{8.8}$$

8.5 Validation

For our experiments we chose 41 versions of JBoss[1]. JBoss is an open source J2EE application server written in Java. The versions we selected for the experiments are at two weeks distance from one another starting from the beginning of 2001 until the end of 2002. The first version has 632 classes, the last one has 4276 classes (we took into consideration all test classes, interfaces and inner classes).

In the followings we will only discuss the *ParallelInheritance* results we obtained on JBoss.

After applying the mechanism described above, we obtained 68 groups of class histories which added subclasses in the same time. Manual inspection showed there were a lot of repetitions (due to the way the concept lattice is built), and just a limited number of groups were useful. Furthermore, inside a group not all classes were relevant for that particular group.

For example, in 19 versions a class was added in the JBossTestCase hierarchy (JBossTestCase is the root of the JBoss test cases). Another example is ServiceMBeanSupport which is the root of the largest hierarchy of JBoss. In this hierarchy, classes were added in 18 versions. That means that both JBossTestCase and ServiceMBeanSupport were present in a large number of groups, but they were not necessarily related to the other classes in these groups.

[1] See http://www.jboss.org.

ClassHistories	Versions
org::jboss::system::ServiceMBeanSupport24 org::jboss::test::JBossTestCase	19 20 27 28 29 30 32 33 34 37 38 39 40 41
javax::ejb::EJBLocalHome javax::ejb::EJBLocalObject	24 41 28 30 32 36 37 38 23

Figure 8.3: Parallel inheritance detection results in JBoss.

These results showed that applying only concept analysis produced too many false positives. That is why we added a filtering step. The filtering step consists in identifying and removing from the groups the entities that changed their relevant properties (*i.e.*, according to the expression) more times than the number of properties detected in a group:

$$FilteringRule = \frac{groupVersions}{totalChangedVersions} > threshold \tag{8.9}$$

In our experiments, we chose the threshold to be $3/4$. For example, if JBossTestCase was part of a group of classes which changed their number of subclasses in 10 versions, we would rule the class out of the group. We chose an aggressive threshold to reduce the number of false positives as much as possible, in the detriment of having true negatives.

After the filtering step, we obtained just two groups. In Figure 8.3 (p.133) we show the class histories and the versions in which they changed the number of children.

In the first group we have two classes which changed their number of children 15 times: ServiceMBeanSupport and JBossTestCase. The interpretation of this group is that the largest hierarchy in JBoss is highly tested. A similar observation was made in Chapter 6 (p.87). With that occasion we detected that the JBossTestCase hierarchy is heterogeneous from the age of the classes point of view.

The second group detects a relationship between the EJB interfaces: EJBLocalHome and EJBLocalObject. This is due to the architecture of EJB which requires that a bean has to have a Home and an Object component.

8.6 Related Work

The first work to study the entities that change together was performed by Gall *et al.* [Gall *et al.*, 1998]. The authors used the change information to define a proximity measurement which they use to cluster related entities. The work has been followed up by the same authors [Gall *et al.*, 2003] and by Itko *et al.* [Itkonen *et al.*, 2004].

Shirabad *et al.* employed machine learning techniques to detect files which are likely to need to be changed when a particular file is changed [Shirabad *et al.*, 2003].

As opposed to the previous approaches, Zimmerman *et al.* placed their analysis at the level of classes and methods [Zimmermann *et al.*, 2004]. Their focus was to provide a mechanism to warn developers that: "Programmers who changed these functions also changed . . . ". Their approach differs from ours because they only look at syntactic changes, while we identify changes based on the semantics of the changes. Furthermore, our approach takes into consideration different changes in the same time.

Davey and Burd proposed the usage of concept analysis to detect evolutionary concepts, but there was no implementation evidence [Davey and Burd, 2001].

Detection of problems in the source code structure has long been a main issue in the quality assurance community. Marinescu [Marinescu, 2002] detected design flaws by defining detection strategies. Ciupke employed queries usually implemented in Prolog to detect "critical design fragments" [Ciupke, 1999]. Tourwe *et al.* also explored the use of logic programming to detect design flaws [Mens *et al.*, 2002]. van Emden and Moonen detected bad smells by looking at code patterns [van Emden and Moonen, 2002]. These approaches differ from ours because they use only the last version of the code, while we take into account historical information. Furthermore, van Emden and Moonen proposed as future research the usage of historical information to detect Shotgun Surgery or Parallel Inheritance.

8.7 Summary of the Approach

Understanding how a system changes can reveal hidden dependencies between different parts of the system. Moreover, such dependencies might reveal bad

smells in the design.

Analyzing the history of software systems can reveal parts of the system that change in the same time and in the same way. We proposed the usage of formal concept analysis, a technique that identifies elements with common properties based on an incidence table specifying binary relations between elements and properties.

To detect the changes in a version, we used expressions that combine different properties to detect complex changes. By applying these queries on every version we obtained an Evolution Matrix annotated with the change information which we then used as input for a concept analysis machine. In other words, we used as elements histories and as properties we used the knowledge of "changed in version i". The results were groups of histories that change together and the versions in which they changed.

An important contribution of our approach is given by the automatic filtering of the raw results of the concept analysis machine: a history is relevant to a concept, if it was not changed in much more versions than the ones in the concept.

According to our algorithm the effectiveness of the approach is highly affected by the value of the threshold. When the threshold is high (*i.e.,* close to 1) we aggressively remove the false positives but we risk missing true negatives. Further work is required to identify the best value for the threshold.

In the future we would also like to apply our approach on more case studies and analyze in depth the results we obtain at different levels of abstraction.

8.8 Hismo Validation

Having history as a first class entity, allowed a straight forward mapping to the elements of the incidence table. To identify the xth property, we computed for the xth version the expression detecting the change. Having historical properties made it easy to encode the expressions.

Below we give the OCL code expressed on Hismo for the ShotgunSurgery expression defined for a ClassVersion:

context ClassVersion

-- returns true if the the number of methods did not change
-- and the number of statements changed with respect to the previous version
derive hasShotgunSurgerySymptom:
(self.ENOM = 0) &
(self.ENOS > 0)

Chapter 9

Van: The Time Vehicle

Implementing Hismo to Combine Analysis Tools

Every successful trip needs a suitable vehicle.

In the end, a meta-model is just a specification. The claim of this dissertation is that a common meta-model allows for the combination of analyses. In this chapter we detail VAN*, our software evolution analysis tool which implements* Hismo *as the underlying meta-model.* Hismo *offers a space in which both time and structure are represented in a uniform way. We show how we can use the same tools to build analyses both on time and on structure. We also show how several analyses can be combined.*

9.1 Introduction

From our experience, when it comes to understanding systems, it is not enough to have prefabricated report generators, but it is crucial to have the ability to interactively investigate the system under discussion. The investigation becomes more difficult as more data is involved, because the more data we need to analyze, the more analyses we need to apply, and different analyses are implemented by different people in different tools. To make all these tools work together, we need a generic infrastructure that integrates them in a dynamic way.

MOOSE is an environment for integrating such tools [Ducasse *et al.*, 2005; Nierstrasz *et al.*, 2005]. The philosophy of MOOSE is to build one space in which all entities are described explicitly and in which all analysis tools coexist. MOOSE grown to be a generic environment for reverse engineering exactly because it changed continuously to integrate all the tools built on top of it in a coherent infrastructure.

Some of the tools built on MOOSE are:

CODECRAWLER – CODECRAWLER is a visualization tool implementing polymetric views [Lanza and Ducasse, 2003; Ducasse and Lanza, 2005]. It is based on a graph notion where the nodes and edges in the graph can wrap the entities in the model. For example, the *Hierarchy Evolution Complexity View* presented in Chapter 6 (p.87) is a polymetric view.

CONAN – CONAN is a concept analysis tool that manipulates concepts as first class entities [Arévalo *et al.*, 2004; Arévalo *et al.*, 2005]. Its target is to detect different kinds of patterns in the model based on combining elements and properties. An example of the usage of concept analysis is shown in Chapter 8 (p.125).

CHRONIA – CHRONIA is an implementation of a CVS protocol to allow direct connectivity to CVS repositories, and it implements several CVS analyses [Gîrba *et al.*, 2005a]. An example, is presented in Chapter 7 (p.105).

HAPAX – HAPAX implements information retrieval techniques for analyzing the semantical information residing in the names of the identifiers and in the comments from the source code [Kuhn *et al.*, 2005].

TRACESCRAPER – TRACESCRAPER analyzes the dynamic traces from different perspectives. For example it offers measurements and visualizations for dynamic traces [Greevy and Ducasse, 2005a]. TRACESCRAPER uses VAN to

analyze the evolution of dynamic traces.

VAN – VAN is our software evolution analysis tool. It implements Hismo and several analyses presented in this dissertation. It interacts with several other tools for implementing different analyses.

In this chapter we pay special attention to the internal implementation of VAN and its influence on Hismo, We also show how the capabilities of MOOSE allow for the combination of tools and how the central meta-model plays a major role. For example, VAN interacts with other tools like CODECRAWLER and CONAN to implement some of the analyses presented in this dissertation.

Structure of the Chapter

In the next section we provide an overview of position of VAN in the overall architecture of MOOSE. Section 9.3 (p.139) shows how we use the same tools to manipulate both structure and history. We present how we used other tools in Section 9.4 (p.141), and we conclude in Section 9.5 (p.145).

9.2 Architectural Overview

Figure 9.2 (p.141) shows the position of VAN in the overall architecture of MOOSE. The default meta-model of MOOSE is FAMIX, a language independent meta-model [Demeyer *et al.*, 2001]. MOOSE has a repository that can store multiple models providing the necessary infrastructure for holding and managing multiple versions.

At the core of VAN is the implementation of Hismo . In our implementation, Hismo is built on top of FAMIX. The first step towards using the historical analyses is to manually set the history. After selecting the versions we want to analyze, all the histories are computed (*e.g.,* ClassHistories, MethodHistories) based on names.

9.3 Browsing Structure and History

MOOSE offers generic tools for manipulating entities and properties. The GENERIC BROWSER allows one to manipulate a list of entities in an spreadsheet like way,

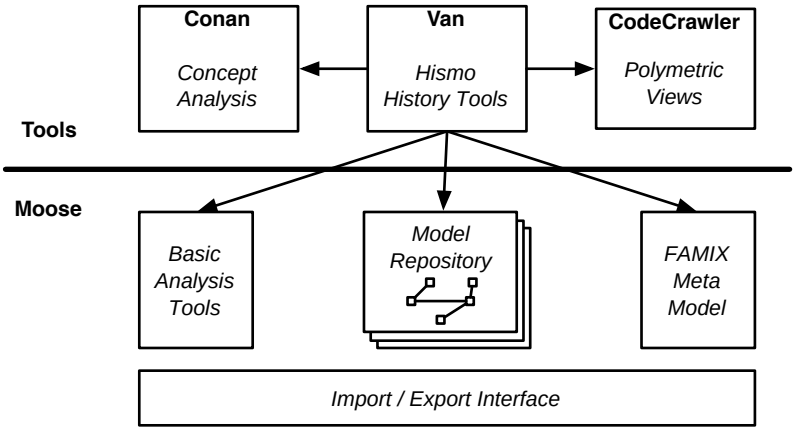


Figure 9.1: VAN and MOOSE. MOOSE is an extensible reengineering environment. Different tools have been developed on top of it (*e.g.,* VAN is our history analysis tool). The tools layer can use and extend anything in the environment including the meta-model. The model repository can store multiple models in the same time. Sources written in different languages can be loaded either directly or via intermediate data formats.

and it also allows for expressions of detection strategies (see Chapter 5 (p.67)) to obtain a sub-group. Each entity has properties, and MOOSE provides an ENTITY INSPECTOR to view these properties.

As in Hismo histories are a explicit entities, we can manipulate them exactly like we do with snapshot entities. Figure 9.3 (p.142) emphasizes the parallelism between manipulating structure and history: on the top part we show snapshot entities, and on the bottom part we show historical entities. On each part, we display a GROUP BROWSER and an ENTITY INSPECTOR. The differences are marked with bold labels on the figure. For example, in the GROUP BROWSER on the top, we display a group of Classes sorted by NOM, while in the ENTITY INSPECTOR on the bottom we display a group of ClassHistories ordered by ENOM.

Chapter 5 (p.67) presents history-based detection strategies as extensions to regular detection strategies. On the bottom part of each GROUP BROWSER there is an editor for detection strategies. For example, on the top part of the figure, we have selected the GodClass detection strategy to be applied on the group of classes. Similarly, on the bottom part, we have selected the Stable GodClass detection strategy.

In Chapter 6 (p.87) we present the *Hierarchy Evolution Complexity View*, a poly-

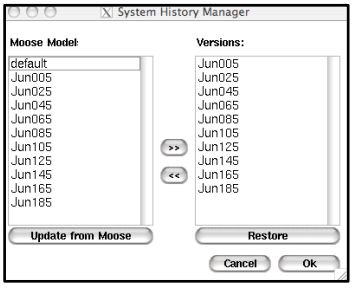


Figure 9.2: VAN gives the historical semantic to the MOOSE models.

metric view showing ClassHistories and InheritanceHistories. CODECRAWLER implements the polymetric views and displays them based on a graph representation of the data. Originally, CODECRAWLER used the polymetric views to display structural entities. One of these views was the System Complexity View, which displays classes as nodes and inheritances as edges. Figure 9.4 (p.143) shows the parallelism between the two views.

MOOSE aims to integrate the tools built on top by providing a registration mechanism that allows the tool to register itself to the context that it knows how to manipulate. This information is used to generate the *menu* for an entity. For example, CODECRAWLER registers the System Complexity View to a group of classes. In this way, every time we select a group of classes we can invoke CODECRAWLER to visualize them in a System Complexity View. Having history as an explicit entity, we use the registration mechanism to allow tools to manipulate histories. For example, we register the *Hierarchy Evolution Complexity View* for a group of ClassHistories.

9.4 Combining Tools

VAN uses CODECRAWLER for visualizing the *Hierarchy Evolution Complexity View*. As described before, CODECRAWLER relies on a graph model and it typically maps entities to nodes and relationships to edges. Using Hismo, the mapping was straight forward: ClassHistories map to nodes and InheritanceHistories map to edges.

We also use CONAN, a Formal Concept Analysis tool, to detect co-change patterns (see Chapter 8 (p.125). In this case too, the bridge was straight forward, because

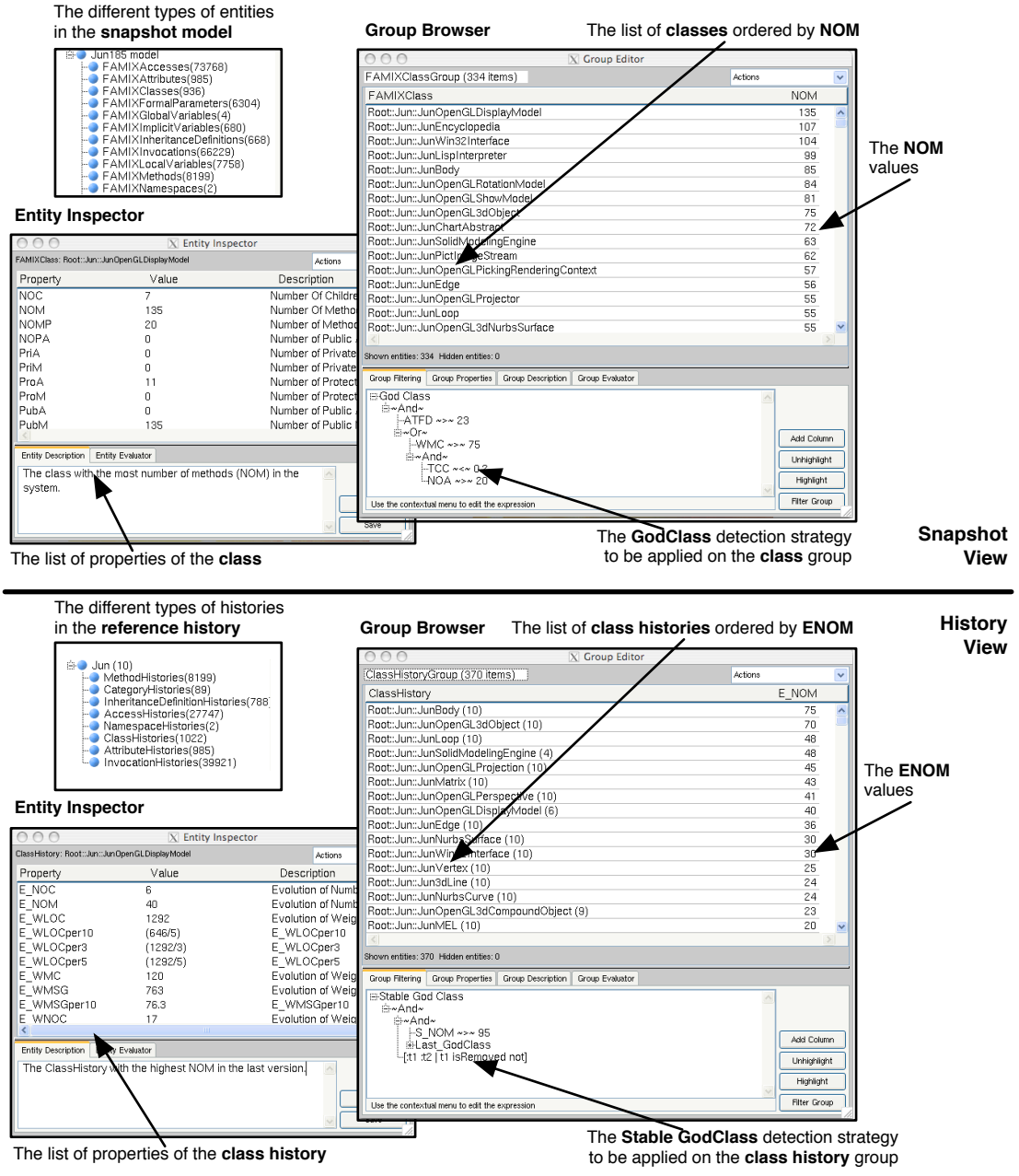


Figure 9.3: Screenshots showing the Group Browser the Entity Inspector. On the top part, the windows display snapshot entities, while on the bottom part they display historical entities.

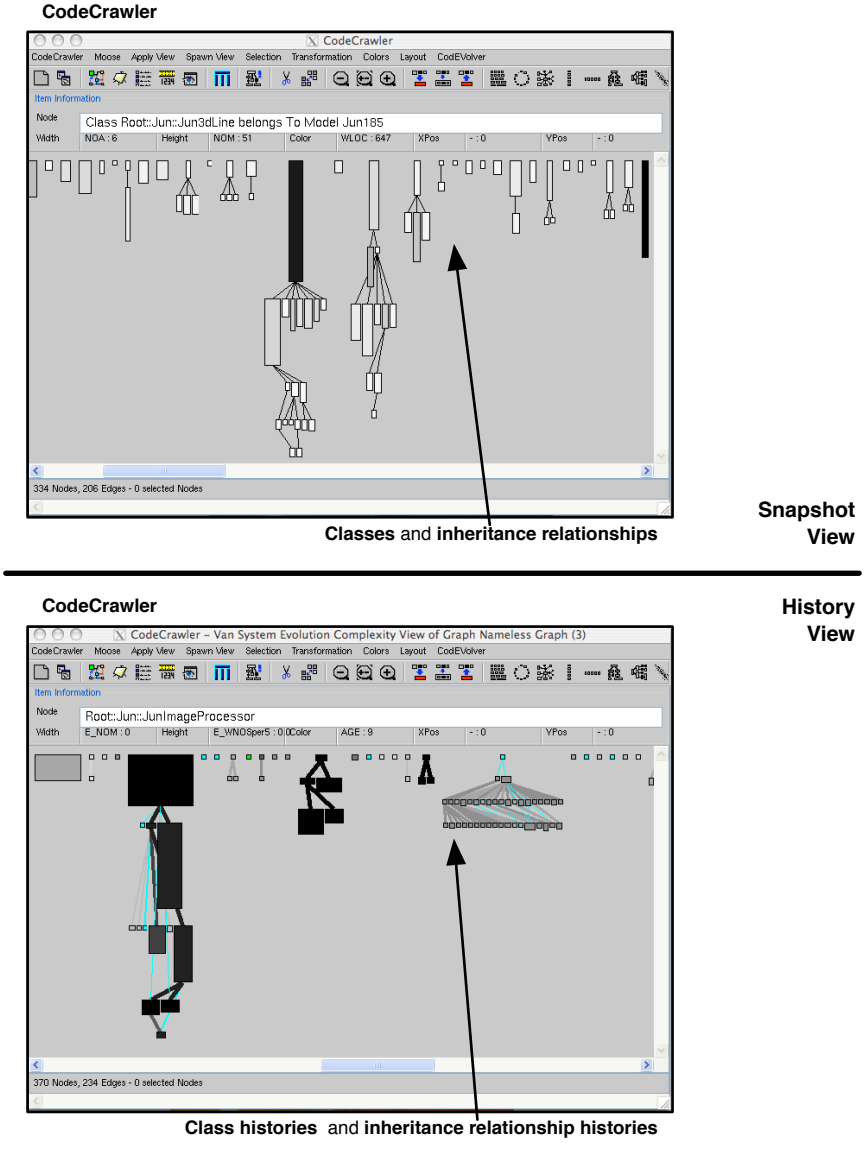


Figure 9.4: Screenshots showing CODECRAWLER. On the top part, it displays class hierarchies in a System Complexity View, while on the bottom part it displays class hierarchy histories in a *Hierarchy Evolution Complexity View*.

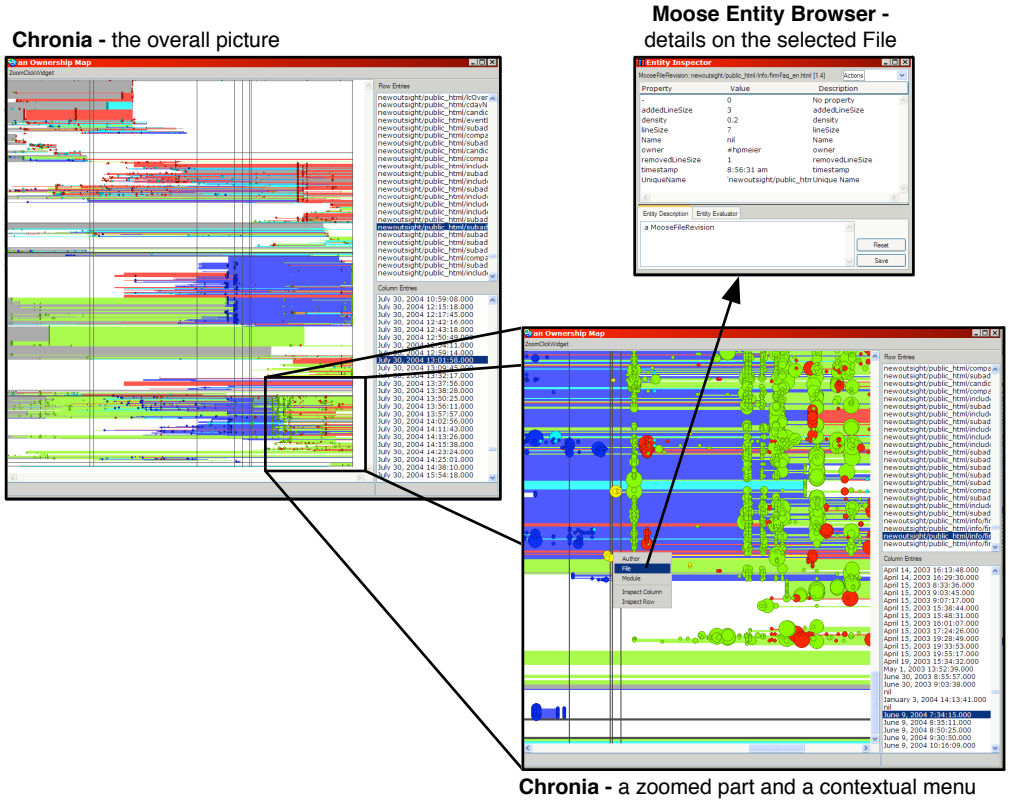


Figure 9.5: Screenshots showing CHRONIA in action: interaction is crucial.

CONAN manipulates entities and properties. We provided histories as entities and "changed in version i" as properties.

Chapter 7 (p.105) introduces the *Ownership Map* visualization to display how authors changed the files in a CVS repository. We implemented the visualization in CHRONIA. To do so, we first implemented a CVS-like meta-model and we implemented the visualization on top. Due to the size of the data, it was not enough to just build a static visualization tool, but we needed an interactive tool. Figure 9.5 (p.144) shows the interactive nature of the tool. In particular, the explicitness of the meta-model allows us to use the MOOSE default tools. For example, we can inspect the FileHistory from the visualization in the ENTITY INSPECTOR.

Another tool built on Van was QUALA [Bühler, 2004]. QUALA aimed to detect

phases in history, by defining a phase as being a first class entity. QUALA used CODECRAWLER to display the detected phases.

9.5 Summary

MOOSE is an environment for reengineering, and several tools are built on top of it. The philosophy of MOOSE is to integrate reengineering tools by making the meta-model explicit. VAN is a version analysis tool implementing Hismo and several evolution analyses. By making history an explicit entity, we used the mechanisms of MOOSE to integrate tools for analyzing software evolution.

Chapter 10

Conclusions

What we can governs what we wish.

Understanding software evolution is important as evolution holds information that can be used in various analyses like reverse engineering, prediction, change impact, or in developing general laws of evolution.

We have reviewed various approaches used to exploit the history of data regarding the evolution of software systems. These approaches typically focus on only some traits of the evolution and most of them do not rely on explicit an meta-model, and because of that, it is difficult to compare and combine the results.

In this dissertation we argue for the need for an explicit meta-model that allows for the expression and combination of software evolution. Based on our literature survey we have gathered requirements for such a meta-model: (1) different abstraction and detail levels, (2) comparison of property evolutions, (3) combination of different property evolutions, (4) historical selection, (5) historical relationships, and (6) historical navigation.

Our solution is to model evolution explicitly. We introduced Hismo, a meta-model centered around the notion of history as an encapsulation of evolution. We argued that Hismo is not tightly dependent on any particular structural construct and that it can be obtained by transforming the snapshot meta-model. In this way, our approach is not even restricted to software analysis, but can be applied to

other fields as well.

As a validation of our meta-model we showed several analyses build on Hismo:

History properties show how to summarize the evolution into properties attached to histories. Historical measurements like Evolution of Number of Methods, are an example of such properties. Such properties allow for historical comparison.

Yesterday's Weather is a complex historical measurement based on the assumption that the latest changed parts also change in the near future. It shows how relevant it is to start reverse engineering from the latest changed parts of the system.

History-based detection strategies are expressions that combine measurements at the snapshot level with historical measurements. We show how using them we can refine the detection of design flaws.

Hierarchy Evolution Complexity View combines the manipulation of historical relationships with historical properties to offer a means to understand the evolution of class hierarchies as a whole.

We built the *Ownership Map* visualization to detect how developers drive software evolution. The analysis is placed at the file level and it shows how developers changed the system. We made use of historical relationships to order the files to provide for a meaningful visualization.

Co-change analysis attracted extensive research. Besides showing how the current state-of-the-art research can be expressed in terms on Hismo, we also proposed a novel approach that makes use of concept analysis to detect *co-change patterns*. We showed how we can group histories to detect patterns like parallel inheritance.

Several other analyses were built using our approach, but were not covered in this dissertation:

Phase detection. Buehler developed an approach to detect phases in histories [Bühler, 2004]. A phase is a selection of a history in which each version complies to a certain expression. For example, he detected growth phases by looking for versions that present additions as compared with the previous version.

Trace evolution. Greevy *et al.* analyzed the evolution of execution traces to characterize the evolution of features [Greevy *et al.*, 2005]. The authors recovered

the mapping between the features and the code by analyzing the execution traces of the features [Greevy and Ducasse, 2005b]. They modeled the Trace as first class entity, and they analyzed the evolution of the traces by defining TraceHistories, and by using historical measurements. For example, they detect how classes changed from being used in only one feature, to being used in several features.

Refactorings detection. Dig and Johnson used version information to detect refactorings based on different heuristics [Dig and Johnson, 2005]. The goal was to analyze how many of the changes that break the API are due to refactorings. They performed the experiment using our VAN tool.

We implemented Hismo and the presented history analyses in our tool called VAN. VAN is built on top of the MOOSE reengineering environment. MOOSE philosophy recognizes that successful reengineering needs to combine several techniques. Several tools have been built over the years in the MOOSE environment, VAN being one of them. We showed how the implementation of Hismo made it possible to combine several tools for accomplishing the required analysis.

10.1 Discussion: How Hismo Supports Software Evolution Analysis

The above applications show how Hismo satisfies the desired activities:

Different abstraction and detail levels. Hismo takes into account the structure of the system. We gave examples of history measurements which take into account different semantics of change (*e.g.,* changes in number of methods, number of statements) on different entities (*e.g.,* packages, classes, methods). As we show Hismo can be applied on any software entities which can play the role of a Snapshot. In this dissertation we present analyses at different levels of abstractions: system, class, method or file.

Furthermore, having detailed information about each version, we can also define the version properties (*e.g.,* number of methods) in terms of history (*e.g.,* "number of methods in version i"). Thus, all the things we can find out from one version can be found out having the entire history. This allows for combining structural data with evolution data in the same expression. For example, we can detect the harmless *God Classes* by detecting those that did not add or remove methods (Chapter 5 (p.67)).

Comparison of property evolutions. History describes the evolution. History measurements are a way to quantify the changes and they allow for the comparison of different entity evolutions. For example, the Evolution of Number of Methods lets us assess which classes changed more in terms of added or removed methods.

Combination of different property evolutions. Some of the evolution analyses need to combine property evolutions with structural properties. For example, the *Hierarchy Evolution Complexity View* combines different property evolutions (Chapter 6 (p.87)), and the history-based detection strategies combines property evolutions with structural information (Chapter 5 (p.67)).

Historical selection. Given a history we can filter it to obtain a sub-history. As the defined analyses are applicable on a history, and a selection of a history is another history, the analyses described in this dissertation can be applied on any selection. For example, *Yesterday's Weather* (Chapter 4 (p.47)) applies the measurements on selections of histories to identify latest changed parts or early changed parts.

Historical relationships. In the same way we reason about the relationships between structural relationships, we can reason about the historical relationships. In Chapter 6 (p.87) we present *Hierarchy Evolution Complexity View* as an example of how to reason about the inheritance histories between class histories.

Historical navigation. Based on the structural relationship "a Package has Classes" we can build the relationship "a PackageHistory has ClassHistories". This can be generalized to any structural relationship and thus, at the history level we can ask a PackageHistory to return all ClassHistories – *i.e.,* all Classes which ever existed in that Package. An example of the usefulness of this feature is given in Chapter 4 (p.47), where we show how we can implement *Yesterday's Weather* for a SystemHistory.

10.2 Open Issues

On the Types of Data Taken Under Study

Most of the analyses deal with source code alone. Yet, reverse engineering needs information about the physical and social environment in which the software was

developed.

Chapter 7 (p.105) presents an example of how to use the author information to recover development patterns. A promising research has been carried out in the Hipikat project [Čubranić and Murphy, 2003]. Other interesting projects correlate the bug information with the change information [Fischer *et al.*, 2003a; Fischer *et al.*, 2003b].

For example an interesting path was explored by the work on analyzing the semantics of code to recover domain concepts [Kuhn *et al.*, 2005]. We would like to employ similar techniques to correlate these concepts with the author and bug information to identify what type of work each concept required.

We believe that to elevate the quality of the analysis, we need to take more types of data into account. In this direction, we believe that Hismo opens the possibility for such an undertaking because it provides the modeling solution.

On How History Could Influence Forward Engineering

We modeled history as being a collection of versions. We took this decision because our focus was reverse engineering and the current state-of-the-art in versioning tools only gave us individual versions.

In fact, a better solution would be to manipulate the entire spectrum of changes as they happen in the development environment, and not just manipulate arbitrary snapshots. That is, a better solution would be to analyze the actual evolution, rather than rebuilding the evolution out of individual versions and recovering the changes based on diff analysis. Although the current data was not available, we believe that this is the next step in managing software evolution.

On the Different Ways to Recover and Manipulate Entity Identity

We refer to entity identity as being the mechanism through which two versions are decided to be part of the same history. In the applications presented here, we reduced entity identity to name identity. That is, when two entities of the same type have the same name in two different versions, they are considered to be part of the same history.

According to this algorithm, when an entity is renamed, its identity is lost. In Chapter 3 (p.29), we discussed two heuristics to deal with this problem [Antoniol

and Di Penta, 2004; Zou and Godfrey, 2003]). These heuristics recover the identity based on some internal properties of the entity. A possibility to complement these heuristics would be to allow the reverse engineer to manually check the results and to take different decisions like merging the histories.

Entity identity is a problem because versioning system do not consider evolution to be a continuous process, but a discrete one. In this case too, keeping the changes as they happened (including their semantics) would make the analysis easier and more precise.

On the Manipulation of Branches

In this dissertation, we modeled history as a sequence of versions which implies a linear version alignment. In the future, we would like to investigate the implications of modeling history as a partially ordered set of versions to represent time as a graph. Such a model would allow manipulation of branches.

Closing Words

We cannot find the flow of things unless we let go.

This entire work is about providing for a model to better understand software evolution. Models are the way we understand the world. They provide a finite representation and they answer questions instead of the real world by allowing for executable reasonings to be expressed on them.

Yet, models are not an end, they are a means. No matter how good a model is, it is still a finite representation, while the real world is infinite.

Modeling is just a step in the process of understanding. To understand the world in its entirety, we need to go beyond models. We need to resonate with the all its details.

October 23, 2005
Tudor Gîrba

Appendix A

Definitions

Entity

— An *entity* is something that has separate and distinct existence in objective or conceptual reality [Soanes, 2001].

Snapshot, Version, Evolution and History

— A *snapshot* is the structure of an entity at a particular moment in time.
Examples of snapshots are classes, methods *etc.*

— A *version* is a snapshot of an entity placed in the context of time.
For example, in the context of CVS, version is denoted by revision.

— The *evolution* is the process that leads from one version of an entity to another.

— A *history* is the reification which encapsulates the knowledge about evolution and version information.

Following the above definitions, we say that we use the history of a system to understand its evolution. Furthermore, the evolution refers to all changes from a version of an entity to another. Sometimes, however, we need to refer to only the change of a particular property of that entity. That is why we define:

— *Property evolution* denotes how a particular property evolved in an entity.

— *Historical property* denotes a characteristic of a history.

For example, the age of a file in CVS is a historical property.

Model and Meta-model

— A *model* is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system [Bézivin and Gerbé, 2001].

— A *meta-model* is a specification model for a class of systems under study where each system under study in the class is itself a valid model expressed in a certain modeling language [Seidewitz, 2003].

We use different terms for different types of models and meta-models:

— A *snapshot meta-model*, or a *structural meta-model* is a meta-model which describes the structure of a class of systems at a certain moment in time.

  Examples of snapshot meta-models are UML or FAMIX.

— An *evolution model* is a simplified view on the evolution of a system.

  Examples of evolution models include the date sequence of each release, a chart showing team allocation over time for a given set of modules, the modifications performed etc.

— An *evolution meta-model* is a meta-model which describes a family of evolution models.

  For instance, in each versioning systems there is an evolution meta-model that specifies which kind of information is kept about evolution.

— A *history meta-model* is an evolution meta-model which models history as a first class entity.

Entity identity

— *Entity identity* denotes the mechanism through which two versions are decided to be part of the same history.

Bibliography

[Antoniol and Di Penta, 2004] Giuliano Antoniol and Massimiliano Di Penta. An automatic approach to identify class evolution discontinuities. In *IEEE International Workshop on Principles of Software Evolution (IWPSE04)*, pages 31–40, September 2004.

[Antoniol *et al.*, 2004] Giuliano Antoniol, Massimiliano Di Penta, Harald Gall, and Martin Pinzger. Towards the integration of versioning systems, bug reports and source code meta-models. In *Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 83–94, 2004.

[Aoki *et al.*, 2001] Atsushi Aoki, Kaoru Hayashi, Kouichi Kishida, Kumiyo Nakakoji, Yoshiyuki Nishinaka, Brent Reeves, Akio Takashima, and Yasuhiro Yamamoto. A case study of the evolution of jun: an object-oriented open-source 3d multimedia library. In *Proceedings of International Conference on Software Engineering (ICSE)*, 2001.

[Arévalo *et al.*, 2004] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, November 2004.

[Arévalo *et al.*, 2005] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Discovering unanticipated dependency schemas in class hierarchies. In *Proceedings of CSMR '05 (9th European Conference on Software Maintenance and Reengineering)*, pages 62–71. IEEE Computer Society Press, March 2005.

[Arévalo, 2005] Gabriela Arévalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. PhD thesis, University of Berne, January 2005.

[Ball and Eick, 1996] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, pages 33–43, 1996.

[Bennett and Rajlich, 2000] Keith Bennett and Vaclav Rajlich. Software maintenance and evolution:a roadmap. In *ICSE — Future of SE Track*, pages 73–87, 2000.

[Bertin, 1974] Jacques Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

[Bézivin and Gerbé, 2001] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of Automated Software Engineering (ASE 2001)*, pages 273–282. IEEE Computer Society, 2001.

[Bieman and Kang, 1995] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings ACM Symposium on Software Reusability*, April 1995.

[Brooks, 1987] Frederick P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987.

[Buckley *et al.*, 2005] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal on Software Maintenance and Evolution: Research and Practice*, pages 309–332, 2005.

[Bühler, 2004] Thomas Bühler. Detecting and visualizing phases in software evolution. Diploma thesis, University of Bern, September 2004.

[Burch *et al.*, 2005] Michael Burch, Stephan Diehl, and Peter Weißgerber. Visual data mininng in software archives. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 37–46, St. Louis, Missouri, USA, May 2005.

[Burd and Munro, 1999] Elizabeth Burd and Malcolm Munro. An initial approach towards measuring and characterizing software evolution. In *Proceedings of the Working Conference on Reverse Engineering, WCRE '99*, pages 168–174, 1999.

[Capiluppi *et al.*, 2004] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. Evolution of understandability in OSS projects. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 58–66, 2004.

[Capiluppi, 2003] Andrea Capiluppi. Models for the evolution of OS projects. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 65–74, 2003.

[Chidamber and Kemerer, 1994] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[Chuah and Eick, 1998] Mei C. Chuah and Stephen G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, pages 24–29, July 1998.

[Ciupke, 1999] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of TOOLS 30 (USA)*, pages 18–32, 1999.

[Collberg *et al.*, 2003] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86. ACM Press, 2003.

[Conway, 1968] Melvin E. Conway. How do committees invent ? *Datamation*, 14(4):28–31, April 1968.

[Davey and Burd, 2001] John Davey and Elizabeth Burd. Clustering and concept analysis for software evolution. In *Proceedings of the 4th international Workshop on Principles of Software Evolution (IWPSE 2001)*, pages 146–149, Vienna, Austria, 2001.

[Demeyer *et al.*, 1999] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE '99 (6th Working Conference on Reverse Engineering)*. IEEE, October 1999.

[Demeyer *et al.*, 2000] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–178, 2000.

[Demeyer *et al.*, 2001] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[Demeyer *et al.*, 2002] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Dig and Johnson, 2005] Daniel Dig and Ralph Johnson. The role of refactorings in API evolution. In *Proceedings of 21st International Conference on Software Maintenance (ICSM 2005)*, pages 389–398, September 2005.

[Draheim and Pekacki, 2003] Dirk Draheim and Lukasz Pekacki. Process-centric analytical processing of version control data. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 131–136, 2003.

[Ducasse and Lanza, 2005] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, January 2005.

[Ducasse *et al.*, 2004] Stéphane Ducasse, Tudor Gîrba, and Jean-Marie Favre. Modeling software evolution by treating history as a first class entity. In *Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 71–82, 2004.

[Ducasse *et al.*, 2005] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, 2005.

[Eick *et al.*, 1992] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[Eick *et al.*, 2001] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[Eick *et al.*, 2002] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *Software Engineering*, 28(4):396–412, 2002.

[Fischer *et al.*, 2003a] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 90–99, November 2003.

[Fischer *et al.*, 2003b] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems.

In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, September 2003.

[Fowler *et al.*, 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Fowler, 2003] Martin Fowler. *UML Distilled*. Addison Wesley, 2003.

[Gall *et al.*, 1997] Harald Gall, Mehdi Jazayeri, René R. Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance 1997 (ICSM '97)*, pages 160–166, 1997.

[Gall *et al.*, 1998] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance 1998 (ICSM '98)*, pages 190–198, 1998.

[Gall *et al.*, 2003] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, 2003.

[Ganter and Wille, 1999] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[Gîrba and Lanza, 2004] Tudor Gîrba and Michele Lanza. Visualizing and characterizing the evolution of class hierarchies. In *WOOR 2004 (5th ECOOP Workshop on Object-Oriented Reengineering)*, 2004.

[Gîrba *et al.*, 2004a] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of ICSM 2004 (20th International Conference on Software Maintenance)*, pages 40–49. IEEE Computer Society Press, 2004.

[Gîrba *et al.*, 2004b] Tudor Gîrba, Stéphane Ducasse, Radu Marinescu, and Daniel Raţiu. Identifying entities that change together. In *Ninth IEEE Workshop on Empirical Studies of Software Maintenance*, 2004.

[Gîrba *et al.*, 2004c] Tudor Gîrba, Jean-Marie Favre, and Stéphane Ducasse. Using meta-model transformation to model software evolution, 2004. 2nd International Workshop on Meta-Models and Schemas for Reverse Engineering (ATEM 2004).

[Gîrba *et al.*, 2005a] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of*

International Workshop on Principles of Software Evolution (IWPSE), pages 113–122. IEEE Computer Society Press, 2005.

[Gîrba *et al.*, 2005b] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance*, pages 2–11, 2005.

[Godfrey and Tu, 2000] Michael Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 131–142. IEEE Computer Society, 2000.

[Gold and Mohan, 2003] Nicolas Gold and Andrew Mohan. A framework for understanding conceptual changes in evolving source code. In *Proceedings of International Conference on Software Maintenance 2003 (ICSM 2003)*, pages 432–439, September 2003.

[Greevy and Ducasse, 2005a] Orla Greevy and Stéphane Ducasse. Characterizing the functional roles of classes and methods by analyzing feature traces. In *Proceedings of WOOR 2005 (6th International Workshop on Object-Oriented Reengineering)*, July 2005.

[Greevy and Ducasse, 2005b] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society Press, 2005.

[Greevy *et al.*, 2005] Orla Greevy, Stéphane Ducasse, and Tudor Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance)*, pages 347–356. IEEE Computer Society Press, September 2005.

[Grosser *et al.*, 2002] David Grosser, Houari A. Sahraoui, and Petko Valtchev. Predicting software stability using case-based reasoning. In *Proceedings of the 17th IEEE International Conference on Automated Software Engienering (ASE '02)*, pages 295–298, 2002.

[Gulla, 1992] Bjorn Gulla. Improved maintenance support by multi-version visualizations. In *Proceedings of the 8th International Conference on Software Maintenance (ICSM 1992)*, pages 376–383. IEEE Computer Society Press, November 1992.

[Hassan and Holt, 2004] Ahmed Hassan and Richard Holt. Predicting change propagation in software systems. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 284–293. IEEE Computer Society Press, September 2004.

[Holt and Pak, 1996] Richard Holt and Jason Pak. GASE: Visualizing software evolution-in-the-large. In *Proceedings of Working Conference on Reverse Engineering (WCRE 1996)*, pages 163–167, 1996.

[Hunt and McIlroy, 1976] James Hunt and Douglas McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[Itkonen *et al.*, 2004] Jonne Itkonen, Minna Hillebrand, and Vesa Lappalainen. Application of relation analysis to a small java software. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 233–239, 2004.

[Jain *et al.*, 1999] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

[Jazayeri *et al.*, 1999] Mehdi Jazayeri, Harald Gall, and Claudio Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.

[Jazayeri, 2002] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[Krajewski, 2003] Jacek Krajewski. QCR - A methodology for software evolution analysis. Master's thesis, Information Systems Institute, Distributed Systems Group, Technical University of Vienna, April 2003.

[Kuhn *et al.*, 2005] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference On Reverse Engineering (WCRE 2005)*, pages ??–??, November 2005. to appear.

[Lanza and Ducasse, 2002] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *Proceedings of LMO 2002 (Langages et Modèles à Objets*, pages 135–149, 2002.

[Lanza and Ducasse, 2003] Michele Lanza and Stéphane Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

[Lanza and Ducasse, 2005] Michele Lanza and Stéphane Ducasse. Codecrawler — an extensible and language independent 2d and 3d software visualization tool. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 74–94. Franco Angeli, 2005.

[Lanza *et al.*, 2006] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. to appear.

[Lanza, 2003] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[Lehman and Belady, 1985] Manny Lehman and Les Belady. *Program Evolution – Processes of Software Change*. London Academic Press, 1985.

[Lehman *et al.*, 1997] Manny Lehman, Dewayne Perry, Juan Ramil, Wladyslaw Turski, and Paul Wernick. Metrics and laws of software evolution – the nineties view. In *Metrics '97, IEEE*, pages 20–32, 1997.

[Lehman *et al.*, 1998] Manny Lehman, Dewayne Perry, and Juan Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 208–217, 1998.

[Lehman, 1996] Manny Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[Lorenz and Kidd, 1994] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[MacKenzie *et al.*, 2003] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd., 2003.

[Marinescu, 2001] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of TOOLS*, pages 173–182, 2001.

[Marinescu, 2002] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Department of Computer Science, "Politehnica" University of Timişoara, 2002.

[Marinescu, 2004] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of ICSM '04 (International Conference on Software Maintenance)*, pages 350–359. IEEE Computer Society Press, 2004.

[McCabe, 1976] T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[Mens and Demeyer, 2001] Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution)*, pages 83–86, 2001.

[Mens *et al.*, 2002] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of SEKE 2002*, pages 289–296. ACM Press, 2002.

[Meyer, 1988] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mihancea and Marinescu, 2005] Petru Mihancea and Radu Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *Proceedings of European Conference on Software Maintenance (CSMR 2005)*, pages 92–101, 2005.

[Mockus and Votta, 2000] Audris Mockus and Lawrence Votta. Identifying reasons for software change using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 120–130. IEEE Computer Society Press, 2000.

[Mockus and Weiss, 2000] Audris Mockus and David Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2), April 2000.

[Mockus *et al.*, 1999] Audris Mockus, Stephen Eick, Todd Graves, and Alan Karr. On measurements and analysis of software changes. Technical report, National Institute of Statistical Sciences, 1999.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Girba. The story of Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 1–10. ACM, 2005. Invited paper.

[OCL 2.0, 2003] Uml 2.0 object constraint language (ocl) specification, 2003. http://www.omg.org/docs/ptc/03-10-14.pdf.

[Parnas, 1994] David Lorge Parnas. Software Aging. In *Proceedings of ICSE '94 (International Conference on Software Engineering)*, pages 279–287. IEEE Computer Society / ACM Press, 1994.

[Pinzger *et al.*, 2005] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[Raţiu *et al.*, 2004] Daniel Raţiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of CSMR 2004 (European Conference on Software Maintenance and Reengineering)*, pages 223–232, 2004.

[Raţiu, 2003] Daniel Raţiu. Time-based detection strategies. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, September 2003. Supervised by Tudor Gîrba and defended at Politehnica University of Timisoara, Romania.

[Ramil and Lehman, 2001] Juan F. Ramil and Meir M. Lehman. Defining and applying metrics in the context of continuing software evolution. In *Proceedings of the Seventh International Software Metrics Symposium (Metrics 2001)*, pages 199–209, 2001.

[Riel, 1996] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[Rochkind, 1975] Marc Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[Sahraoui *et al.*, 2000] Houari A. Sahraoui, Mounir Boukadoum, Hakim Lounis, and Frédéric Ethève. Predicting class libraries interface evolution: an investigation into machine learning approaches. In *Proceedings of 7th Asia-Pacific Software Engineering Conference*, 2000.

[Seidewitz, 2003] Ed Seidewitz. What models mean. *IEEE Software*, 20:26–32, September 2003.

[Shirabad *et al.*, 2003] Jelber Sayyad Shirabad, Timothy C. Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *International Conference on Software Maintenance (ICSM 2003)*, pages 95–104, 2003.

[Soanes, 2001] Catherine Soanes, editor. *Oxford Dictionary of Current English*. Oxford University Press, July 2001.

[Sommerville, 2000] Ian Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[Stasko *et al.*, 1998] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[Taylor and Munro, 2002] Christopher M. B. Taylor and Malcolm Munro. Revision towers. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50. IEEE Computer Society, 2002.

[Tufte, 1990] Edward R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[Čubranić and Murphy, 2003] Davor Čubranić and Gail Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[Čubranić, 2004] Davor Čubranić. *Project History as a Group Memory: Learning From the Past*. Ph.D. thesis, University of British Columbia, December 2004.

[van Emden and Moonen, 2002] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conf. Reverse Engineering*, pages 97–107. IEEE Computer Society Press, October 2002.

[Van Rysselberghe and Demeyer, 2004] Filip Van Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, September 2004.

[Viégas *et al.*, 2004] Fernanda Viégas, Martin Wattenberg, and Kushal Dave. Studying cooperation and conflict between authors with history flow visualizations. In *In Proceedings of the Conference on Human Factors in Computing Systems (CHI 2004)*, pages 575–582, April 2004.

[Voinea *et al.*, 2005] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005)*, pages 47–56, St. Louis, Missouri, USA, May 2005.

[Ware, 2000] Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[Wu *et al.*, 2004a] Jingwei Wu, Richard Holt, and Ahmed Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89. IEEE Computer Society Press, November 2004.

[Wu *et al.*, 2004b] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 90–99. IEEE Computer Society Press, November 2004.

[Xing and Stroulia, 2004a] Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, pages 123–128, 2004.

[Xing and Stroulia, 2004b] Zhenchang Xing and Eleni Stroulia. Understanding class evolution in object-oriented software. In *Proceedings of the 12th International Workshop on Program Comprehension (IWPC 2004)*, pages 34–43, 2004.

[Xing and Stroulia, 2004c] Zhenchang Xing and Eleni Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 242–251. IEEE Computer Society Press, 2004.

[XMI 2.0, 2005] Xml metadata interchange (xmi), v2.0, 2005. http://www.omg.org/cgi-bin/doc?formal/05-05-01.

[Ying *et al.*, 2004] Annie Ying, Gail Murphy, Raymond Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):573–586, 2004.

[Zimmermann and Weißgerber, 2004] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories*, pages 2–6, 2004.

[Zimmermann *et al.*, 2003] Thomas Zimmermann, Stephan Diehl, and Andreas Zeller. How history justifies system architecture (or not). In *6th International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 73–83, 2003.

[Zimmermann *et al.*, 2004] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, 2004.

[Zou and Godfrey, 2003] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, pages 146–154, November 2003.

Modeling History

to Understand Software Evolution

Tudor Gîrba is born in Romania, on August 8, 1977, and he is a Romanian citizen.

2005
Consultant at Sweng. Software Engineering GmbH, Berne, Switzerland

2002-2005
PhD student and assistant at Software Composition Group, University of Berne, Switzerland

2001-2002
Software engineer and coach at Sava Technologies SRL, Timișoara, Romania

2001-2002
Co-Founder of the LOOSE Research Group, Politehnica University of Timișoara, Romania

1997-2000
Programmer and designer at Piron, an independent group of game developers

1996-2001
Student at the Politehnica University of Timișoara, Romania

Over the past three decades, more and more research has been spent on understanding software evolution. The development and spread of versioning systems made valuable data available for study. Indeed, versioning systems provide rich information for analyzing software evolution, but it is exactly the richness of the information that raises the problem. The more versions we consider, the more data we have at hand. The more data we have at hand, the more techniques we need to employ to analyze it. The more techniques we need, the more generic the infrastructure should be.

The approaches developed so far rely on *ad-hoc* models, or on too *specific* meta-models, and thus, it is dif ficult to reuse or compare their results. We argue for the need of an explicit and generic meta-model for allowing the expression and combination of software evolution analyses. We review the state-of-the-art in software evolution analysis and we conclude that:

> *To provide a generic meta-model for expressing software evolution analyses, we need to recognize the evolution as an explicit phenomenon and model it as a first class entity.*

Our solution is to encapsulate the evolution in the explicit notion of *history* as a sequence of *versions*, and to build a meta-model around these notions: Hismo. To show the usefulness of our meta-model we exercise its different characteristics by building several reverse engineering applications.

This dissertation offers a meta-model for software evolution analysis yet, the concepts of history and version do not necessarily depend on software. We show how the concept of history can be generalized and how we can obtain our meta-model by transformations applied on structural meta-models. As a consequence, our approach of modeling evolution is not restricted to software analysis, but can be applied to other fields as well.