Detecting and Visualizing Phases in Software Evolution

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Thomas Bühler

October 2004

Supervised by:

Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik
Universität Bern

The address of the author:

Thomas Bühler
Riedliweg 43
CH-3053 Münchenbuchsee
`buehler@iam.unibe.ch`

Abstract

Understanding the evolution of an object-oriented system based on various versions of source code requires analyzing a vast amount of data since an object-oriented system is a complex structure rather than a collection of classes.

Our work provides an approach to understand such an evolution by detecting and visualizing phases in the evolution, *i.e.*, abstractions of time spans where the encapsulated versions all comply with an expression. Our approach is applicable on any level, *i.e.*, not only on system level, but for example also on class level. Our approach furthermore contains a set of measurements on phases that characterize them.

Phases help understand an evolution because on the one hand because they enable studying an evolution on a higher level. On the other hand, phases can be detected with multiple expressions at the same time. This results in concurrent phases which enables studying an evolution from different perspectives at the same time.

Acknowledgements

First of all, I want to thank Tudor Gîrba for his kind supervision and Prof. Dr. Oscar Nierstrasz for giving me the opportunity to carry out my master thesis in his group in such an uncomplicated way. I also want to thank Prof. Dr. Stéphane Ducasse for helping me with Quala and reviewing my work.

Special thanks go to Prof. Dr. Michele Lanza for guiding and supporting me in my student project and in my diploma work and for all those discussions that gave me the confidence of being on the right track. He guided and helped me on a voluntary base despite being so busy himself. Thank you very much Michele.

Thanks also go to all the students from the pool for sharing those discussions and lunches with me. Special thanks go to Markus Kobel, David Vogel, Adrian Lienhard and Galogero Butera.

Most of all, I want to thank the people beyond university, whose lives were affected by this work: My parents and my brother for supporting me making this possible. And last but not least my shining star Joanna Zipsin-Caputo for supporting and encouraging me in spite of all those storms. Thank you, I could not have done this without you.

Thomas Bühler
October 2004

Contents

Chapter 1

Introduction

Software systems are now ubiquitous. Virtually all electrical equipment now includes some kind of software; software is used to help manufacturing industry, schools and universities, health care, finance and government; many people use software of different kinds for entertainment and education [Som00].

A typical software system is modified and extended a number of times during its lifetime to keep it operational. In fact, the majority of software engineers today are not involved with the production of new systems but are busy with changing and extending existing software systems [Jon98]. Lehman et al. stated in the first of their *laws of program evolution dynamics* that a software system that is used in a real-world environment necessarily must change or it becomes progressively less useful in that environment [LB85, Leh96, LPR$^+$97]. There are mainly two reasons why software has to change: On the one hand the software must evolve to meet changing customer needs and on the other hand to fit in changing environments.

Software therefore has to be developed in a way that it can evolve after its initial development. But what are the characteristics that make software easier or harder to change? To answer this question, we first have to be able to gain an understanding of the evolution of a software system. Therefore, means are needed to analyze the history of a system, *i.e.,* the development of a software system and its changes. Various kinds of information can be used, for example results from interviewing developers, feedback from customers such as bug reports, the source code, etc. We however restrict ourselves to analyzing source code, *i.e.,* the source code of multiple versions. Furthermore, we restrict ourselves to object-oriented systems. Thus, the problem addressed in this work is **understanding how an object-oriented system evolved into its current state based on the source code of its versions**.

The main problem that arises is the vast amount of data that has to be analyzed. Research results must scale up to industrial applications for them to be useful [JH99]. A case study of 50 versions of code of a certain software system, each consisting of about 500 classes would call for analyzing and comparing 250'000 classes. Furthermore, an object-oriented software system is a complex structure consisting of classes, methods, attributes and different kinds of relationships between them rather than simply a set of classes. Understanding its evolution thus requires analyzing the change of such a complex structures over time. Analyzing discrete changes are only of little help because in general a vast amount of changes are made at the same time.

Two techniques which can be used to reduce complexity are software metrics and software visualization. The latter technique helps in terms of essentially simplifying the study of multiple aspects in parallel by visually displaying them ("one picture conveys a thousand words"). Metrics can help to assess the complexity of software entities and to discover artifacts with unusual measurement values.

We use both software metrics and visualization techniques. Based on software metrics we detect abstractions of time intervals in a history of a software entity which we name *phases* and in which, from a certain perspective, all encapsulated versions have an identical evolution. These phases are then displayed so the observer can visually and interactively get a quick understanding of the analyzed history. Our approach is not based on filtering out information. Instead, we define means to create abstractions which help to quickly get a coarse understanding of the analyzed history and which can be used as a vantage point for further inspections on a more detailed level.

Furthermore, we define various kinds of measurements on the created abstractions. On the one hand, we define general measurements such as the duration of a phase in hours, the number of encapsulated versions, etc. On the other hand we define measurements that are based on the kind of the abstracted changes such as the amplitude of a phase or the certainty of a phase. The same mechanism can then be used to define measurements to further classify the changes.

Document structure

- In Chapter 2 we present several different approaches to tackle software evolution.

- In Chapter 3 we present our approach of detecting phases with phase descriptions. We start by presenting our approach in short, then introduce its prerequisites FAMIX, HISMO, software metrics, and detection strategies, and finally present our approach. Therefore we first introduce the notions phase and phase description, then measurements on phases and phase descriptions and finally a way to visualize the evolution with detected phases. This chapter includes a template to define phase descriptions.

- This template is then used in Chapter 4 and 5 to define two catalogs of phase descriptions. The first one is applicable on system level, *i.e.,* on the evolution of a system. The second catalog is applicable on class level.

- In Chapter 6 we validate our approach on the evolution of two systems, Jun and SmallWiki. First, we present the visualization of detected phases in a part of Jun's evolution and explain how this visualization can be read. Then we analyze the entire life cycle of SmallWiki on three levels. First, we analyze the measurements based on phases and phase descriptions, then visually discern stages in SmallWiki's evolution and finally demonstrate how phases can be used interactively to get a detailed understanding of the changes.

- In Chapter 7 we use phases on the class level by analyzing the visualization of five classes that evolved differently.
- In Chapter 8 we close this work by presenting a summary and possible future work.
- In Appendix A we describe the tool we developed in the context of this work.
- Appendix B contains the definitions of all applied detection strategies.
- Appendix C contains the description of the software metrics we applied.

Chapter 2

State of the Art

In this chapter we present some developments in the field of software evolution. We start with a brief summary of the pioneering work of Lehman et al. and then a set of newer approaches grouped by their research goal. First, we present different approaches that define quality-related measurements based on history information, then approaches that use the history information to detect error prone items in software systems, and finally approaches that define ways to use software visualization to understand software evolution.

2.1 Metrics and Laws of Software Evolution

Since the 1970's research has been spent on building a theory of evolution by formulating laws based on empirical observations. The observations are based on interpreting evolution charts which represent some property on the vertical (*e.g.*, the number of modules or the number of changed modules) and time on the horizontal axis. The major work in this area was carried out by Lehman et. al [LB85, LPR$^+$97, LPR98]. Their studies were captured in a series of laws of software evolution that are presented in Table 2.1. From the point of view of software engineering, such statements must be accepted as an external regulating and constraining force. To overcome them requires expertise in organizational dynamics, management, sociology, etc., not just software technology [LPR$^+$97]. The laws all relate specifically to E-type systems, that is, to software systems that solve a problem or implement a computer application in the real world [Leh96].

This approach has recently been applied on a case study on the kernel of Linux[1], an open source operating system. Godfrey et. al studied the evolution of the Linux kernel at both the system level as well as within each of the major subsystems and compared the results to Lehman's laws of software evolution [GT00]. They found that at the system level the growth of the Linux kernel has been superlinear which is a violation of Lehman's fifth law.

[1]http://www.linux.org

	Name	Description
1	Continuing Change	E-type systems must be continually adapted else they become progressively less satisfactory.
2	Increasing Complexity	As an E-type evolves its complexity increases unless work is done to maintain or reduce it.
3	Self Regulation	The E-type system evolution process is self regulating with distribution of product and process measures close to normal.
4	Conservation of Organizational Stability	The average effective global activity rate in an evolving E-type system is invariant over product lifetime.
5	Conservation of Familiarity	As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
6	Continuing Growth	The functional content of an E-type systems must be continually increased to maintain user satisfaction over their lifetime.
7	Declining Quality	The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
8	Feedback System	E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 2.1: Laws of Software Evolution

2.2 Time Based, Quality-Related Measurements

In this section, we briefly present five approaches that define ways to extract quality related measurements from the history of a software. These approaches address a different objective than our work.

Metrics to Assess the Maintainability Based On the Calling Structure

Burd and Munro present a number of metrics to assess the **maintainability** of code [BM99]. The availability of such metrics has the potential to assess if and how maintenance changes have effected the **comprehensibility** of the code. Furthermore they claim that the metrics could potentially allow an initial assessment of a number of proposed maintenance changes and thus allow the selection of a strategy that offers the best overall evolutionary path.

The proposed metrics are based on so called **dominance trees** which provide an abstraction to a standard call graph reducing the calls to a single link between functions within a code module. The proposed metrics are based on identifying those instances in the dominance tree where more than a single call is present and therefore a number of functions need to be included within a mental model during a maintenance intervention. It is then investigated what effect software evolution has on the relations in a dominance tree that are present within specific code modules. Changes to these relations are then tracked over time to give an indication of the changing complexity of the code.

Assessing the Evidence of Code Decay from Change Management Data

Eick et al. present a number of measurements that index code decay [EGK$^+$01]. Code is defined as being **decayed if it is more difficult to change than it should be**. It is stressed that decay is distinct from the ability of the software to meet requirements and that software that is decaying may nevertheless be increasing in value. Eick et al. then present so called **code decay indices**, that are both quantified and observable in a version management data base and represent symptoms, risk factors and prognoses of code decay. They are based on a version management system that tracks changes at multiple levels, *e.g.,* on module and file level. Examples for such indices are the number of changes to a module in a certain time interval or the span of a change (the number of files the change touches). Applying these measurements on the entire change management history of a large, fifteen-year old real-time software system for telephone switches, they found strong **evidence that code does decay**, *i.e.,* their analysis demonstrates 1) the increase over time in the span of the changes (number of files touched per change), 2) the decline in modularity of subsystems, 3) contributions of several factors to fault rates in modules of the code, and 4) that span and size of changes are important predictors of the effort to implement a change. Eick et al. anticipate that all projects of sufficiently large scale will exhibit decay to some extent: **code decay is a generic phenomenon**.

Understanding Conceptual Changes in Evolving Source Code

Gold and Mohan defined a framework to understand the conceptual changes in an evolving system [GM03]. Their approach is based on **concepts** which are “defined as descriptive terms at a higher level of abstraction than the source code, nominated by the maintainer to describe some abstractions of interest”. The authors then characterize changes to concepts based on a framework to describe the possible changes to concepts. For example, the concept change type *Split* occurs when a concept becomes separated into two or more other concepts. These concepts changes are then related to actions on the part of the maintainer and to comprehensibility issues in the future. The goal is to allow the development of methods to automatically assess comprehensibility and guide the maintainer to improving code quality. Furthermore, the long-term aim of this project is to develop a predictive model for code quality given the change type and changes made.

Software Evolvability

Stephen Cook et al. present an approach to understanding software evolution that is based around the **quantifiable concept of evolvability** [CJH]. The authors define evolvability as the capability of software products to be evolved to continue to serve their customers in a cost effective way. The concept of evolvability brings together factors from three main areas: software product quality, software evolution processes, and the organizational environment in which software is used. In this approach, not only source code is considered, but also abstract representations, architectures and designs of software products, which enables the measurement of evolvability to be conducted as early as possible in the software life cycle process.

Predicting Software Stability using Case-Based Reasoning

An approach with a similar goal is presented by Grosser et al. [GSV02]. They aim at assessing the **stability** in object-oriented systems which they define as the ease with which a software system or a component can evolve while preserving its design as much as possible. Their approach has thus a similar goal as software evolvability but instead of defining a traditional model for their measurement, they use **case-based reasoning** based on the hypothesis that two software items which show same or similar characteristics will also evolve in a similar way. Software entities are then, with the use of software metrics, represented as cases. To asses the stability of a new case, a set of known cases is used to find the known cases that match best. Those are then used to predict the stability of the new case.

2.3 Approaches Aiming at Identifying Specific Entities in the Current Version

In this section, we briefly summarize four approaches that use history information to detect entities in the current version. Three of the presented approaches detect, based on heuristics or observations, entities that show flaws.

Detecting Refactorings via Change Metrics

Demeyer, Ducasse, and Nierstrasz use, in the context of reverse engineering, multiple versions of a software to detect where the implementation has changed [DDN00]. They propose a set of heuristics to detect specific changes, *i.e.*, refactorings, by applying object-oriented metrics to successive versions of a software system. Those detection heuristics are based on change metrics which are the difference of a metric measurement between two successive versions. However, identifying which refactorings have been applied when going from one version to another by itself does not imply that we can actually deduce how and why the implementation has drifted from its original design.

Yesterday's Weather

Yesterday's Weather [GDL04] is an analysis based on the retrospective empirical observation that the classes which have changed most recently also suffer important changes in the near future. It is thus an approach for identifying key classes for reverse engineering activities based on the assumption that the parts which change are those that need to be understood first. The approach consists of identifying, for each version of a subject system, the classes that were changed the most in the recent history of a software system and checking if these are also among the most changed classes in successive versions. The number of versions in which this supposition holds is then divided by the total number of analyzed versions to obtain the values of Yesterday's Weather. It thus indicates the predictability which classes will be changed in the future and thus are key classes for reverse engineering.

Time Based Detection Strategies

Marinescu detected flaws in object-oriented design by applying so-called detection strategies which are measurement-based rules on a single version of a system [Mar01, Mar02]. Ratiu enlarged this concept by taking the history of the detected design fragments into account to increase the accuracy of the problem detection: If in the past the flaw proved not to be harmful then it is less dangerous. This enlarged concept is captured in so called time-based detection strategies [Raţ03, RDGM04].

Product Release History

Another approach in understanding evolution is presented in [GJKT97]. It is based on examining the structure of several major and minor releases of a large telecommunication switching system based on information stored in a database of product releases. The historical evolution of the structure is tracked and the adaptions made are related to the structure of the system. The goal of this work is to identify modules or subsystems that should be subject to restructuring or reengineering activities.

The structural information in the data base is extracted and stored by preprocessors during compilation. It is then analyzed on a macro-level, *i.e.*, by investigation of structural information about each release (such as version number of system modules indicating major or minor releases).

2.4 Understanding Evolution using Visualization

In this section, we present three approaches that provide means to understand the evolution of software with visualization techniques. First, we present the evolution matrix which basically displays every single class in every version in an object-oriented system. Then we present an approach to visualize the evolution of class hierarchies based on rectangles, lines, and colors. Finally, we present the approach of Jazayeri to use color to depict the amount of changes applied to a specific version.

The Evolution Matrix

The evolution matrix [Lan01,Lan03] combines software visualization and software metrics to visualize the evolution of the classes of a software system in a matrix. The evolution matrix displays multiple versions of a system at class level. Each column of the matrix represents one version of the software while each row represents the different versions of the same class (two classes are considered the same if they have the same name). Within the columns, the classes are sorted alphabetically in case they appear the first time in the system. Otherwise they are placed at the same vertical position as their predecessors. The classes themselves are represented with rectangles. The size of the rectangles is given by different measurements applied on the class version.

The evolution matrix allows one to read different kind of information: First, the size of the system in a particular version (in terms of number of classes) is the height of the corresponding column. Secondly, the added and removed classes can easily be detected since added classes are displayed in a new row that starts at the bottom of the column and removed classes leave empty spaces. And thirdly, the overall shape of the evolution matrix is an indicator for the evolution of the whole system. A growth phase is indicated by an increase in the height of the matrix, while during a stabilization phase (no classes are being added or removed) the height of the matrix remains the same.

Besides characterizing the evolution at system level, the evolution matrix provides based on the measurements that are used to determine the dimensions of the rectangles information about the classes themselves. Based on this information, a classification of different evolution patterns has been made.

Visualizing and Characterizing the Evolution of Class Hierarchies

Gîrba and Lanza present a visualization approach to understand the evolution of class hierarchies [GL04]. Class hierarchies provide a grouping of classes based on their similar semantics. Thus, understanding a hierarchy as a whole reduces the complexity of understanding big systems. The authors introduce the notion of a history as a first class entity and define measurements which summarize the evolution of an entity or a set of entities. These measurements are then used to define polymetric views [LD03,Lan03], *e.g.*, the *Class Hierarchy History Complexity View*. Two-dimensional nodes are used to represent entities, edges to represent relationships. This simple visualization is enriched with up to 5 metrics on the node characteristics and 2 metrics on the edge characteristics. In the *Class Hierarchy Complexity View* for example, the color of the class history nodes and the width of the inheritance edges is used to display their age. The authors stress that polymetric views are intrinsically interactive.

Color Visualization

Jazayeri applied another approach using visualization to analyze evolution [JGR99, Jaz02] which is based on color. A history of a release is displayed in a color percentage bar which contains different colors. The colors represent different version numbers of certain parts of the release. This allows the observer to quickly observe the amount of changes form one release to the next. Large variations in color indicate a release that is undergoing a big amount of change.

Chapter 3

Detecting Phases

In this chapter we present our approach to support understanding the evolution of software based on its source code. That is, we comparatively analyze source code at different points in time focusing on the changes that have been made. We name source code at one particular point in time a *version*, and a sequence of versions a *history*. Formulated with these terms, we present an approach to understand a history based on comparatively analyzing the changes made from one version to another. It is based on detecting phases where from one particular point of view the same kind of changes have been made. That is, we detect sequences of versions in a history where the versions have, from one particular point of view the same characteristics.

A simple example is to detect growth phases in the history of a software system, *i.e.*, phases where code has been added. Assuming that we have a measurement that indicates the size of the system at one version, we can detect those phases where the size measurement raised, as exemplified in Figure 3.1.

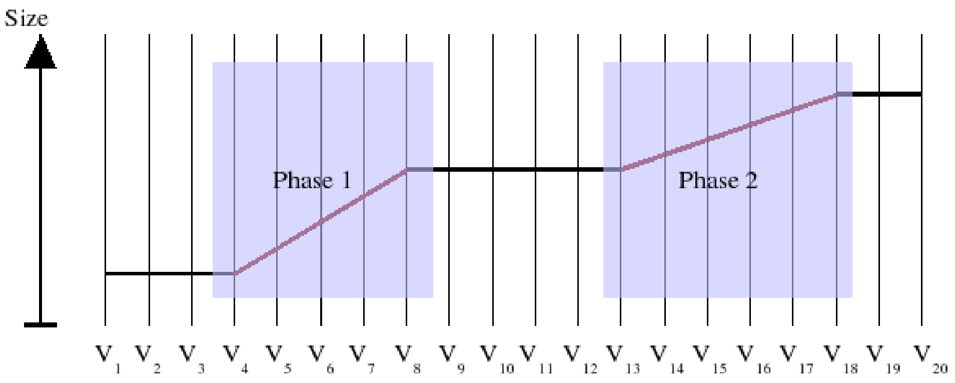

Size
Phase 1
Phase 2
V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8 V_9 V_{10} V_{11} V_{12} V_{13} V_{14} V_{15} V_{16} V_{17} V_{18} V_{19} V_{20}

Figure 3.1: Visualization of the concept of a phase. Each vertical line represents one version while the line leading from the left to the right depicts the progression of the size of the system.

Phases are detected with an expression we name *phase description* . This phase description is applied to every version of a history. If it evaluates to true for a particular version, this version is part of a phase. In the previous example, we use an expression that evaluates to true if a version has a bigger size than its predecessor version.

In a history, we can detect phases with multiple phase descriptions at the same time and hence study the relationship between different kind of phases, *i.e.,* phases detected with different phase descriptions. For example, we can detect phases where the system grew and phases where classes have been removed at the same time. Studying the temporal relationship between the detected phases, we might for example discover time spans where the system grew but at the same time classes have been removed or time spans where the system grew and no classes have been removed. To study such temporal relationships of phases, we use a way to visualize detected phases described later on.

For further analysis of detected phases, we introduce a set of measurements that characterize phases. First, we present measurements that characterize single phases such as the duration of a phase. We then present measurements that are dependent from the phase description a phase is detected with. Then, we present measurements that give information of all phases detected by one phase description.

The source code however is not directly analyzed. Instead, we use the FAMIX meta-model to build models of the source code at multiple points in time. These FAMIX models are then used to build a historical model with the HISMO meta-model. To analyze such a model, we use software metrics which characterize a model and its contained entities with numbers or other symbols. We also use detection strategies, which are expressions to detect fragments of a model based on measurements. The FAMIX and the HISMO meta-model, software metrics and detection strategies are thus prerequisites for our approach. We therefore present them in the next sections before we then come to our approach of detecting and visualizing phases.

3.1 The FAMIX Meta-Model

FAMIX [DTD01, TDDN00] is a meta-model for a language independent representation of object-oriented source code. It is an entity-relationship model that models object-oriented code at program entity level, *i.e.,* entities and relationships such as classes, methods, invocations and accesses are being modeled, but not complete abstract syntax trees. Additional to the entities themselves, FAMIX defines a set of well defined attributes for every entity. Figure 3.2 shows the core entities and relations of the FAMIX model which is set up as an object-oriented hierarchy. The complete model consists of more entities and relationships such as invocations and formal parameters.

FAMIX has been design to fulfill the following requirements [TDD00]:

- **Language independence -** FAMIX is a meta-model to model object-oriented

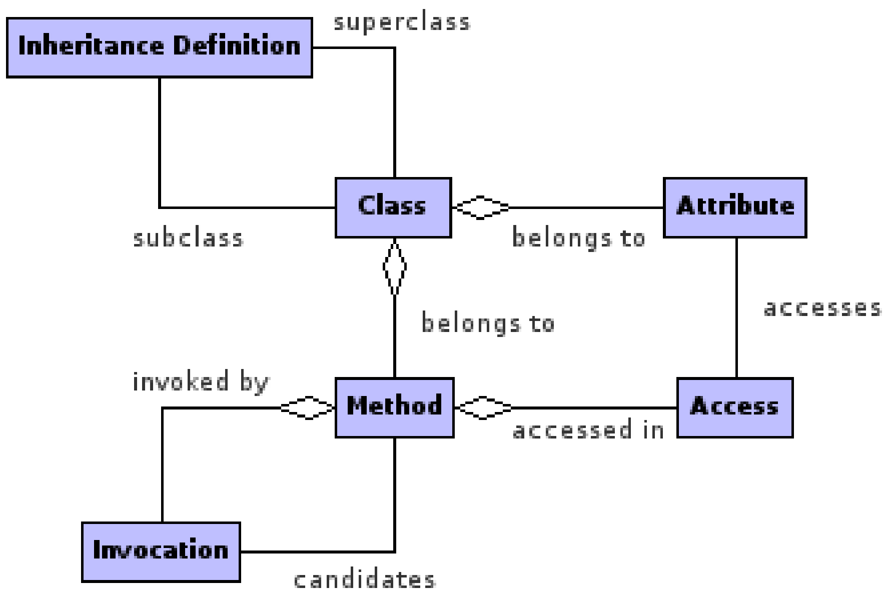


Figure 3.2: Core of the FAMIX meta-model

source code in a language independent way. This opens up the possibility to reason about languages on a more abstract level.

- **Extensibility** - FAMIX allows for adding new abstractions and specific attributes to existing abstractions. This extensibility is for example needed to model language-specific information.
- **Information Exchange** - The information is stored in a flat, streaming-friendly way and it uses a unique naming scheme that is valid over multiple transfers.

3.2 The History Meta-Model HISMO

The FAMIX meta-model provides a way to model object-oriented source code at a single point in time. To study software evolution, we need to analyze multiple FAMIX models which represent source code at different points in time. Therefore, we model sequences of multiple FAMIX models and their temporal relationships with the the history meta-model HISMO [DGF04].

The history meta-model is a meta-model which treats historical information just like any other kind of information. It is centered around the two notions of a version and a history which are defined as follows:

Definition 3.1 (Version) *A version is a snapshot of an entity at a certain point in time [GDL04].*

Definition 3.2 (History) *History is a sequence of versions and encapsulates knowledge about evolution and version information [DGF04].*

According to these definitions, a history encapsulates a sequence of versions of the same kind of entity. Also, the relationship at version level has a correspondent at history level (*e.g.*, as a *Class* has more *Methods*, a *ClassHistory* has more *MethodHistories*). A history does not have a direct relation with a version entity. Instead, it has a direct relation with multiple versions which then each have a direct relation with a model entity.

The meta-model HISMO can be applied on any meta-model. In our work, we base it on the source code meta-model FAMIX. A version encapsulates thus a FAMIX entity and a history accordingly a sequence of versions of this entity. A reduced schema of the HISMO structure is depicted in Figure 3.3.

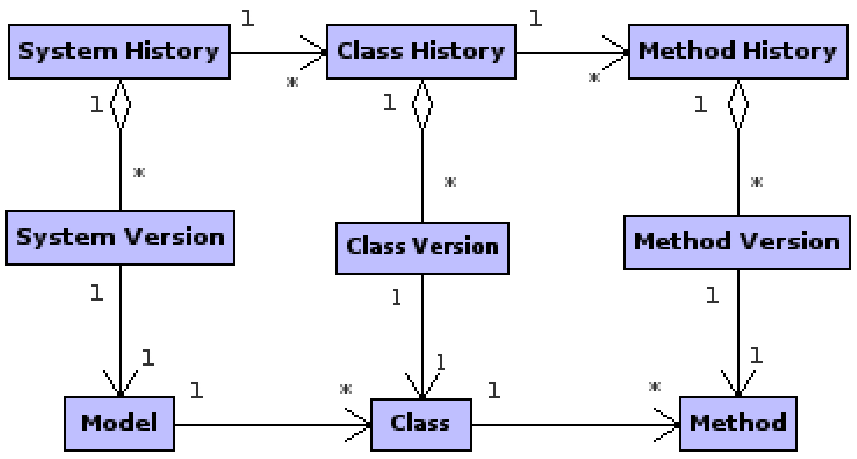


Figure 3.3: The History Meta-Model based on FAMIX

With HISMO, we can study a history as a whole, but we can also compare two distinct versions. Furthermore, HISMO provides change information at different levels of abstraction, *e.g.*, at system level, package level, class level, method level, etc.

In this work, we present an approach to detect phases in the evolution of a software entity. Formulated with the notion of a history, we present an approach to detect phases in the history of a software entity.

3.3 Software Metrics

In the previous sections, we presented how we use the meta-models FAMIX and HISMO to build a model of object-oriented source code at multiple points in time. In order to achieve our goal, detecting phases in a history, we need means to further analyze such a model. For that, we do not directly analyze source code and instead use software metrics.

Software metrics measure certain properties of software entities by mapping them to numbers (or to other symbols) according to well-defined, objective measurement rules. The measurement results are used to describe, judge, or predict characteristics of the software system with respect to the property that has been measured.

Software metrics can be divided into structural metrics and historical metrics. Structural metrics measure characteristics on software entities, in our case FAMIX entities, while historical measurements measure differences of structural measurements of different versions of a software entity.

3.3.1 Structural Software Metrics

Generally, structural software metrics can be divided into two categories [LK94]:

1. **Design Metrics** are measurements of the static state of the project's design. Design metrics are used to assess the size and in some cases the quality and complexity of software. They tend to be more locally focused and more specific, thereby allowing them to be used effectively to directly examine and improve the quality of the product's components.

2. **Project Metrics** deal with the dynamics of a project, with what it takes to get to a certain point in the development life cycle and how to know to be there. They can be used in a predictive manner for example to estimate staffing requirements. Being at a higher level of abstraction, they are less prescriptive and more fuzzy but are more important from an overall project perspective.

In this work, we restrict ourself to design metrics. Furthermore, we focus on metrics that can be computed from the source code.

We define and use a **metrics as a function** of an entity. As an example, $NOM(ClassA)$ stands for the **n**umber **o**f **m**ethods in ClassA. In the past, a vast amount of structural metrics have been defined. In our work, we use only a small set of relatively simple metrics. They are presented in Appendix C.

3.3.2 Historical Software Metrics

In this work, we restrict ourselves to change metrics. A change metric is the difference of a measurement of two consecutive versions:

Definition 3.3 (Change Metric) *The change metric δM of a metric M of a version V_i is defined as*

$$\delta M(V_i) := \begin{cases} M(V_i) - M(V_{i-1}) & if\ V_i\ has\ a\ predecessor\ version\ V_{i-1} \\ 0 & else \end{cases}$$

where V_{i-1} is the predecessor version of V_i.

A change metric thus indexes the change of a measurement of an entity. It can be positive if the measurement in the new version is higher or negative if it is lower. It is 0 if the measurement in both versions is identical or the version has no predecessor.

3.4 Detection Strategies

In our work, we use detection strategies to select entities in a model which comply with a certain characteristic which is expressed with software metrics. Detection strategies are defined as follows:

Definition 3.4 (Detection Strategy) *A detection strategy is the quantifiable expression of a rule by which design fragments that are conforming to that rule can be detected in the source code [Mar02].*

The term “quantifiable expression of a rule” means that the rule must be properly expressible by software metrics. Hence, a detection strategy is a generic mechanism for detecting a set of design fragments based on metrics.

Detection Strategies as Functions

We define a detection strategy as a function which returns, based on the input set of entities, the set of entities with which the underlying expression evaluates to true. As an example, the set of data classes among the set of classes S detected by the detection strategy $DataClass$ can be expressed as $DataClass(S)$.

Furthermore, we use the notion of $|S|$ to denote the size of the set S, *i.e.*, the number of elements in the set S. For example, the number of data classes among the set of classes S is $|DataClass(S)|$.

Detection Mechanism

The detection mechanism of detection strategies is based on the principle of data filters and composition operators.

Definition 3.5 (Data Filter) *A data filter is a mechanism (a set operator) through which a subset of data is retained from an initial set of measurement results, based on the particular focus of the measurement [Mar02].*

Data filters can be classified in two main categories: marginal and interval filters. Marginal Filters can be further divided into semantical and statistical filters where the threshold value is determined with statistical methods. Semantical filters contain a threshold value and a direction. They can be further divided into absolute semantical filters (*e.g.,* HigherThan, LowerThan) and relative semantical filters (*e.g.,* TopValues, BottomValues). Statistical filters also use a threshold value but it is derived with statistical methods from the initial data set. The classification of filters made in [Mar02] is shown in Table 3.4.

Type of Filter	**Limit Specifiers**		**Filter Examples**
Marginal	Semantical	Relative	TopValues(10)
			BottomValues(5%)
		Absolute	HigherThan(20)
			LowerThan(6)
	Statistical		BoxPlot
Type of Filter	**Specification**		**Filter Example**
Interval	Composition of two marginal filters with semantical limit specifiers of opposite polarities		Between(20,30) := HigherThan(20) ∧ LowerThan(30)

Table 3.1: Classification of Data Filters

The key issue in filtering data is to reduce the initial data set in order to detect those design fragments that have special properties captured by the metric. Since a detection strategy is the "quantifiable expression of a design rule", it must be able to quantify entire design rules. A data filter only supports the interpretation of individual metric results. To quantify entire design rules, a mechanism which provides a correlated interpretation of multiple result sets is needed:

Definition 3.6 (Composition Operators) *The operators used to compose a set of metrics into an "articulated rule" are called composition operators [Mar02].*

Composition operators can be seen from a logical and from a set point of view. From the logical, a composition operator reflects a "joint-point" of sub-expressions of the representation of a design rule. The set point of view helps understanding how the final result set is built since a composition operator combines filtered sets with common set operators. In Marinescu [Mar02], three composition operators are presented: AND, OR, and BUTNOT. We enriched this set with the XOR operator.

Thus, a detection strategy first filters sets of design fragments using data filters. The resulting filtered sets are then combined with composition operators. This combination results then in a new set of design fragments which can again be filtered and combined with other sets.

Detection strategies are mainly used to detect design flaws which violate certain design rules [JF88] or design heuristics [Rie96]) or correspond to bad smells

[FBB+99]. But their usage is not restricted to detecting design problems. They are a generic mechanism for analyzing a source code model using metrics [Mar02, RDGM04] and can actually be used to detect any kind of design fragments. *E.g.*, in this work, detection strategies are used to detect added leaf classes, removed super classes and classes where certain refactorings have been applied. The list of detection strategies we applied is presented in Appendix B.

Example - Detecting Extended Classes

Let us assume we want to detect classes that have been extended, *i.e.*, where methods or attributes have been added. Thus, we are looking for classes C where $\delta NOM(C) > 0$ or $\delta NOA(C) > 0$, where NOM and NOA are the metrics that measure the number of methods respectively the number of attributes of a class. The detection strategy $ExtendedClass$ which detects such classes can be defined as follows:

$$ExtendedClass(S) := S' \left| \begin{array}{l} S' \subseteq S,\ \forall C \in S' \\ \delta NOM(C) > 0 \vee \delta NOA(C) > 0 \end{array} \right.$$

It is thus a function that detects the subset of classes S' out of the set of classes S which all have an increased number of methods or attributes.

3.5 Phase and Phase Description

After presenting the prerequisites, we now define our approach of detecting phases in a history. The core of it is the two notions of a phase description and a phase:

Definition 3.7 (Phase Description) *A phase description is an expression to detect sequences of consecutive versions which all comply with it.*

Definition 3.8 (Phase) *A phase is a sub-history that encapsulates a non-empty set of consecutive versions which all comply with a phase description.*

Definition of Phase Descriptions

The definition of a phase description is like a detection strategy based on data filters and composition operators. The applied filters and composition operators are the same as presented in Section 3.4.

Phase descriptions are defined as mathematical functions, taking a history, *i.e.*, a set of versions as their input. They compute a set of phases, that is, a set of coherent parts of the input history.

Example

An exemplary problem is to detect growth phases in the history H of a system. For that, we assume that the system grows from version v_{i-1} to its successor version v_i if there are more classes in version v_i, that is, if $\delta NOCL(v_i) > 0$. Thus, the growth phases in the history H can be detected with the following phase description.

$$Growth(H) := \left\{ H' \,\middle|\, H' \subseteq H, \forall v \in H' \; \delta NOCL(v) > 0 \;\right\}$$

In words, this phase description is a function that takes the history H as its input and returns all phases H' that encapsulate versions v that contain more classes than their predecessor.

Discussion of the Terms

A phase is, according to its definition, an abstraction of multiple consecutive versions and thus represents a coherent part in a history. Since a phase encapsulates multiple versions of a history into a new entity, it enables regarding part of a history with the same characteristic as a unit and thus enables analyzing histories on a more abstract level. For example, a phase in a system history might encapsulate a time span where the system has been growing, certain flaws have been introduced, bugs have been corrected, etc. A phase might be a complete history, *i.e.*, encapsulate all versions, but at the minimum it consists of one version.

A phase is the detection result of a phase description. However, if a phase description detects a sequence of more than one consecutive versions, there are several alternative ways to encapsulate those versions in phases. For example, if two consecutive versions are detected, they might be summarized into one single phase or into two phases with each containing one single version. In this work, we use phases to encapsulate the longest coherent sequence unless we mention it otherwise.

From the conceptual point of view, a phase is a regular history: Neither the fact that the encapsulated versions are consecutive nor that they all comply with a phase description is a contradiction to the notion of history. Nevertheless the phase description a phase is detected with has a specific detection goal. The detected phase can be regarded as a suspect that, with a certain probability, conforms to this goal. This view, a phase as a suspect of the detection goal of a phase description, has lead us to the definition of a phase. Considering phases, we always refer to their underlying detection goal. For example, if we talk about growth phases, we refer to a suspect time interval which is detected with a specific growth phase description.

Measurements on Phases

For further inspections on obtained phases, we define a set of measurements on a history. Since a phase is a history, these measurements are thus also defined on phases.

Phase Measurement	Description
Length	Number of versions a history encapsulates
Duration	Duration of a history in hours
Density	Duration divided through the length. The density thus indicates how many hours the encapsulated versions are in average apart from each other

The measurement length is identical with the age of a history which is defined in [GL04]. We gave this measurement a different name to avoid confusions with duration.

Measurements on Phase Descriptions

For further analysis on the applied phase descriptions we define a set of measurements that are based on the set of phases detected by a certain phase description. They are therefore defined as a function that takes a phase description PD applied on a history H as its input.

$$NOP(PD(H)) := |PD(H)|$$

$PD(H)$ is the set of phases detected by the phase description PD in the history H). $NOP(PD(H))$ indicates the size of this set. In other words, it is the **n**umber **o**f **p**hases detected by the phase description PD in the history H. Note that this measurement is dependent on the way phases are built out of the by PD detected versions.

$$VersionCoverage(PD(H)) := \frac{\sum_{P \in PD(H)} Length(P)}{Length(H)}$$

The version coverage of a phase description PD applied on a history H is thus the fraction of the number of versions that are encapsulated by any of the phases $PD(H)$ and the length of the history H. It is thus a value between 0 and 1 that indicates proportionally to what extent the phases detected by PD cover a history H. A value of 0.5 for example indicates that every second version of H is part of a phase detected by PD.

$$TimeCoverage(PD(H)) := \frac{\sum_{P \in PD(H)} Duration(P)}{Duration(H)}$$

The time coverage is a similar measurement as the version coverage. It indicates proportionally to what extent the time span encapsulated by the entire history H is covered by any of the phases $PD(H)$.

$$AverageM(PD(H)) := \frac{\sum_{P \in PD(H)} M(P)}{NOP(PD(H))}$$

AverageM(PD(H)) is a generic measurement that is defined as the average of the measurement M of all phases $PD(H)$. For example, $AverageLength(PD(H))$ is the average length of all phases $PD(H)$ detected by the phase description PD in the history H .

Attributes on Phase Descriptions

We define two optional attributes of a phase description that have the same semantics across all phase descriptions: **certainty** and **amplitude**. That is, the meaning and the range of these measurements is defined independently of the description of a phase, but the way it is computed is based on the underlying phase description.

The definition of the certainty and amplitude is shown in Table 3.5.

Name	**Range**	**Description**
Certainty	[0 - 1]	The certainty is a measurement that indicates how clear it is that a detected phase corresponds to the detection goal of its phase description.
Amplitude	[0 - ∞]	The amplitude is a positive measurement that indicates the extent of the changes the underlying phase description captures .

Table 3.2: Attributes of a Phase Description

Let us consider the previous example of the growth phase description again. A useful definition of the *amplitude* of a phase is the amount the system grew in the time span the phase encapsulates, that is, the sum of the $\delta Size(V)$ over all versions V the phase encapsulates. The *certainty* could be defined as a fraction which divides the amount of removed code through the amount of added code. A high value would thus indicate that almost the same amount of code has been removed and added and that it is thus not clear whether the system really grew.

Equally to the certainty and the amplitude, further additional measurements on phase descriptions can be defined. For example, we could define a classification number attribute which gives further information about the characteristics of a phase. This classification number could be either a single number or a multi-dimensional vector. In the example of the growth phases, we could define a measurement that indicates to what extent the added classes are subclasses of a framework.

Combining Phase Descriptions

For further investigations, multiple phase descriptions can be combined with composition operators since they are like detection strategies expressions. For example, one might be interested in phases that comply with two or more or to one of multiple phase descriptions. For that, a new phase description has to be created which uses a composition operator to combine phase descriptions. In this work, we use the composition operator AND, OR, XOR and NOT as presented in Section 3.4.

Comparison of the concepts phase description and detection strategy

A phase description is basically similar to a detection strategy: Both are expression to detect a set of entities that comply with it. But there are differences between the two concepts:

1. A detection strategy aims at detecting design fragments, *i.e.*, parts in the design in one version. In contrast, a phase description detects versions in the history of one design fragment.

2. In contrast to a phase description, the result of a detection strategy is the computed set of entities. A phase description uses the resulting set of versions to create phases summarizing consecutive versions.

3.6 Visualization of Phases

Up to now, we presented how phases can be detected using phase descriptions and how they can be further described with measurements of phases. To get a quick understanding of the computed data and thus recover the evolution at a higher abstraction level, we visualize phases using software visualization. Visualizing simplifies studying relationships between phases respectively their phase descriptions and enables filtering out phases that visually stand out since good visual display allows the human brain to study multiple aspects of complex problems in parallel. Our visualization allows us to visually detect when phases of different phase descriptions arise in parallel and when phases have striking measurements.

Building on the concept of polymetric views [LD03, Lan03], we chose to visualize phases with rectangles. The **position** of the left side indicates at what point in time the underlying phase starts - the right side when it ends. A phase that is positioned left to another phase encapsulates thus earlier versions. The vertical position of a phase corresponds to its phase description. All phases detected with one phase description are aligned horizontally; in the beginning of each line we put the name of the corresponding phase description.

The **width** of a phase's rectangle displays the phase's duration. If the amplitude is defined in a phase's underlying phase description, it is used to determine the **height** of a rectangle. Otherwise, a default value is used. In this case, all the rectangles that correspond to this phase description have the same height.

Thus, the width has for all phases the same definition while the height may be defined differently for every phase description. However, phases with the same phase description can be compared.

The **colors** are used to display the certainty and an additional measurement, for example a classification number, if defined. If both are defined, the corresponding figures have two stripes. The color of the upper stripe shows the certainty while the lower one shows the classification number. Both of these measurements must be defined as values between 0 and 1. This range is proportionally mapped into a shade of gray: White corresponds to 0 and black to 1. If only one of those measurements is defined, the rectangle only has one color stripe of which the defined measurement determines the color. If none is defined, the rectangle is simply filled black.

A phase encapsulates conceptually a time-interval where the versions with a phase description. The phase description detects sequences of versions that comply with it, generally based on change metrics. A phase with length one indicates a specific change from one initial version to the detected version. The initial version of the change is however conceptually not included in the phase. For the visualization, we include the initial version of the change, *i.e.,* the preceding version of the first detected version of a phase. Thus, a phase with length one is displayed as a rectangle which leads from the initial, not detected version to the detected version.

Figure 3.4, which shows the visualization of a single phase, exemplifies this principle.

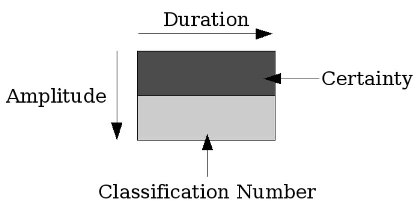


Figure 3.4: Visualization of One Phase

The entire visualization displays the complete history from left to right: The left border represents the earliest version while the right border stands for the latest. Every version of that history is represented with a vertical line of which the horizontal position is determined by the point in time of the version. The distance between two lines thus indicates how much time passed between the two corresponding versions. Note that we consider real time stamps instead of numbering the versions. An example of an entire visualization is provided in Figure 3.5.

Note that this visualization technique is, as it is implemented, interactive. Just by looking at the resulting picture, it is of limited use. Indeed, the viewer must interact with the visualization to get finer grained and more detailed information of the displayed phases. How the user can do so is described in Appendix A.

Example

Figure 3.5 shows the visualization of the phases of three different phase descriptions in the history of a system. Each vertical line represents one system version. We can already see that there are areas where the lines are closer together which means that less time passed from one version to another.

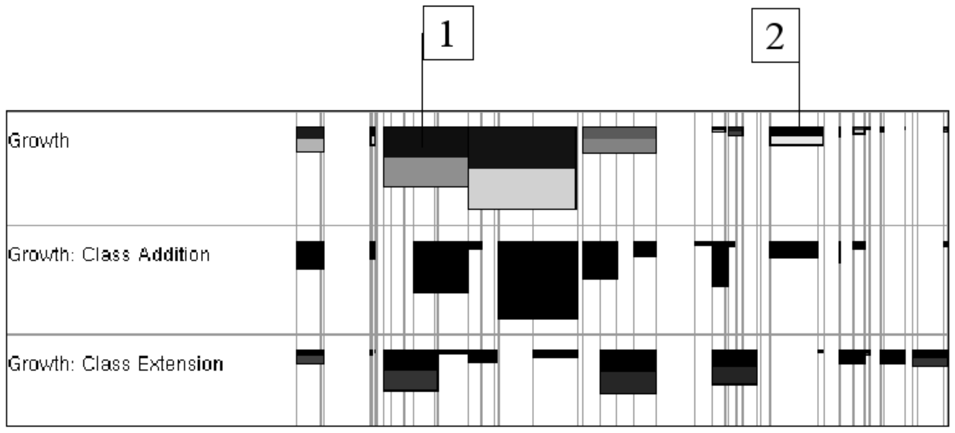


Figure 3.5: Example Visualization of Phases

The 'Growth' phase description detects phases where the size of the system increased. Two of its phases are marked with numbers. The amplitude of the phase marked with '1' is about three times higher than the amplitude of the one marked with '2' since its figure is about three times higher. This means that the phase '1' encapsulates a much stronger growth. The duration of both phases is however about the same since the corresponding rectangles have about the same width. Their density is considerably different, because the phase '1' encapsulates more versions. Both phases have a certainty value close to one since the color of the upper stripe is in both phases almost black. It is thus in both phases clear that the system really grew, even though the phase '2' encapsulates only small growth.

The 'Class Addition' phase description detects phases where classes have been added while the third phase description, the 'Class Extension' phase description detects phases where classes have been extended, *i.e.*, where methods or attributes have beed added. On the visualization, we can now visually discover time spans where classes have been added but no classes have been extended if a version is part of a 'class addition phase' but not of a 'class extension' phase. Explicitly, these phases could be detected by combining the 'class addition' with the 'class extension' phase descriptions with a *BUTNOT* composition operator. By analogy we can discover phases where classes have been added but no classes extended, phases where 'class addition' and 'class extension' phases arise at the same time.

In those time spans where 'class addition' and 'class extension' phases arise in parallel, we can discover phases where the amplitude of one phase is considerably higher than the other one. For example, if there is a class addition phase with a high amplitude and a class extension phase with a low amplitude in parallel, we know that the encapsulated change consisted mostly of the addition of new classes but also a few classes have been extended.

3.7 A Template to Describe Phase Descriptions

In the following two chapters, we define two catalogs of phase descriptions. In Chapter 4 we present a catalog of phase descriptions that is applicable on system level and in Chapter 5 one applicable on class level. These catalogs contain phase descriptions that have been useful in our analyzes.

We describe the provided phase descriptions with a template that consists of the three paragraphs motivation, definition, and discussion which are explained below.

Motivation. This paragraph describes the motivation for the definition of the underlying phase description and what information a detected phases offers.

Definition. The paragraph definition first contains a textual explanation of the applied detection mechanism and possibly defined measurements and then the formal definition.

Discussion. The discussion paragraph contains a list of various points. The most often listed points are:

- **Concurrent Phases -** Phases of another phase description that are necessarily detected in the same time interval.
- **Asynchronous Phases -** Phases of another phase description that can not be detected in the same time interval.
- **False Positives -** A description of situations that may be detected as part of a phase but are not.
- **False Negatives -** A description of situations that a phase description does not detect but conform to its detection goal.

The discussion paragraph provided only in the catalog applicable the system level and not in the one applicable on class level. The mentioned points are listed in most phase descriptions, *i.e.*, where they are relevant. Additionally, the discussion paragraph contains in some phase descriptions additional points.

3.8 Discussion

Summarizing, we presented an approach to detect phases, *i.e.*, time intervals where the versions match with the same phase description and thus, from a certain point of view, show the same kind of changes. This detection mechanism was enriched with several kind of measurements, *i.e.,* 1) general measurements on phases such as the length of a phase, 2) phase description specific measurements on phases, and 3) measurements on phase descriptions such as the time coverage. Then, we introduced a way to visualize the phases based on their point in time and multiple measurements.

The detection of phases can be used in various ways and can be used in following tasks:

- **Interpreting measurements.** The provided measurements can be used to gain overall information of the analyzed history. This information can be used as an initial assessment of a specific system or to categorize histories of different software entities. Note that our approach can be used on different levels, *e.g.,* on the system level, subsystem level, class level, etc. Thus, we can use phases to analyze the history of an entire system, of a subsystem, of a single class, etc.

- **Visually analyze correlations between phases**. The presented way to visualize phases lets us visually analyze when phases appear concurrently. Since the measurements of the phases are displayed as well, we can visually compare the measurements of different phases. This technique can be used to describe the analyzed history, *e.g.,* one could say “here we can see growth accompanied by refactoring activities while... ”.

- **Get the details analyzing single phases**. Single phases are an abstraction of a time span with specific changes. To understand the exact changes, one can analyze single phases. The phase indicates when and with a certain probability (depending on the definition of the underlying phase description) how certain changes where made. What exactly changed can then be made out by inspecting the phase. This technique is especially useful when different concurrent phases are inspected at the same time.

The presented approach thus bridges the gap between high level analysis of entire histories and low-level analysis that aim at understanding single changes. Furthermore, it is applicable on any kind of history. For example, it can be used to analyze the history of a class or of a subsystem.

Chapter 4

Phase Description on System Histories

This chapter contains a catalog of phase descriptions that detect phases in a system history. A detected phase encapsulates a set of consecutive versions of a complete system (or subsystem). The presented phase descriptions are grouped in three categories: The growth and reduction, maintenance and refactoring phase description categories.

The first category aims at describing growth and reduction in a system history. On these phase descriptions, we define the certainty and the amplitude measurements. Furthermore, we define an additional measurement named classification number that further characterizes the changes made in the time span encapsulated in a phase. We define these three measurements in a way they can be used to compare phases with different phase descriptions.

The second category, named maintenance, contains phase descriptions that aim at detecting small changes. The first phase description aims at detecting the correction of bugs, the second the cleaning of methods, and the third changes that do not appear in the applied measurements. In the maintenance phase descriptions we do not define the amplitude, the certainty or other measurements.

The last category contains the refactoring phase descriptions. In this category, we present 7 phase descriptions that aim at detecting phases where specific refactorings [FBB$^+$99] have been applied. In all of them, we define the amplitude but we forgo the definition of the certainty measurement and any additional measurements.

4.1 Growth and Reduction Phase Descriptions

In this section we present three phase descriptions to describe growth in a history of a system. A simple way to measure growth from one version to another is to only take the change of the number of classes into account, *i.e.*, to define that a system grows from one version to another if it contains more classes than before.

There can also be growth without new classes having been added, that is, if classes have been extended with new methods and possibly also new attributes. In this work we apply an empirical formula to determine growth in a system history.

We consider that functionality has been added if there were new classes added or existing classes extended. We consider a class as extended if there were methods or attributes added to it (or both). To measure the amount of the added functionality we define the measurement *Addition* of a version V of a system as follows:

$$\textbf{Addition(V)} := |NewClass(V)| + \frac{1}{2}|ExtendedClass(V)| \tag{4.1}$$

It is thus the sum of the number of added classes and half of the number of extended classes (from one version to another). The number of added and extended classes is determined by the detection strategies *NewClass* and *ExtendedClass* which are defined in Appendix B.7 and B.1. The number of the extended classes is weighted half to the number of new classes based on the assumption that if the amount of functionality of one new class is added by extending existing classes, it requires extending more than one class.

By analogy to the measurement Addition, we define measurement Reduction:

$$\textbf{Reduction(V)} := |RemovedClass(V)| + \frac{1}{2}|ReducedClass(V)| \tag{4.2}$$

This definition is based on the detection strategies *RemovedClass* and *ReducedClass* which are defined in Appendix B.8 and B.2. The detection strategy *ReducedClass* detects a class version if it contains less methods, attributes or both.

Based on these two definitions we define a measurement to measure the growth between two versions of a system:

$$\textbf{Growth(V)} := Addition(V) - Reduction(V) \tag{4.3}$$

According to this definition, we define that a system grows from one version to another if $Growth(V) > 0$, shrinks if $Growth(V) < 0$ and remains the same size if $Growth(V) = 0$. Based on that we present the **system growth** and the **system reduction** phase descriptions that indicate when and to what extent a system grows respectively has been reduced.

There might be functionality added to a system version V even if the system does not grow, *i.e.*, if $Addition(V) > 0$ but $Reduction(V) > Addition(V)$. To detect these phases we provide the next phase description, the **general growth** phase description. It aims at detecting when and to what extent functionality has been added to a system. It thus, compared to the system growth phase description, disregards the amount of removed functionality. By analogy, we present the **general reduction** phase description which detects the removal of functionality disregarding the amount of added functionality.

Additionally we present the **new hierarchies growth** and the **hieraries removal** phase descriptions which aim at detecting the addition respectively the removal of complete inheritance hierarchies.

For all growth and reduction phase descriptions we define the amplitude and the certainty measurement. Furthermore, we define an extra measurement named classification number. The certainty and the classification number are defined identically in all growth and reduction phase descriptions and thus offer means to compare phases of different phase descriptions.

The **classification number** splits the growth into addition and extension of classes, that is, it indicates with a proportional value to what extent the growth respectively the reduction consists of each. The **certainty** indicates how clear it is that the system really grew in a phase. For that it compares the amount of removed and added functionality. The **amplitude** is defined slightly differently in the system growth and system reduction phase descriptions than in the others: It is the sum of $Growth(v)$ over all versions v in a phase while in the other phase descriptions it is the sum of $Addition(v)$ in all versions v in a phase. Note that although the definition of the amplitude is not equal in all growth phase descriptions, it can still be used in comparisons.

The presented formulas use the detection of new and removed classes. Instead, they could also use the less exact possibility of using the change metric $\delta NOCL$. But in that case, we would not discover any changes in a situation where at the same time an identical amount of classes has been added and removed. With our approach, we would receive a positive $Addition$ measurement and therefore detect a *general growth* phase in this situation. We would also receive a positive $Reduction$ measurement and hence detect a concurrent *general reduction* phase.

4.1.1 System Growth Phase Descriptions

Motivation. The system growth phase description aims at detecting phases in the history of a system where the system has been growing based on the definition of the $Growth$ measurement. We define the amplitude of a system growth phase as the size of the growth. A system growth phase with a larger amplitude thus encapsulates a bigger growth of the system than one with a smaller amplitude. Furthermore we compare the amount of added and removed functionality to determine the certainty of a phase. A similar amount of removed and added code results in a low certainty value while a certainty values close to 1 indicates that the amount of removed functionality is negligible compared to the amount of added functionality in a system growth phase. If in a system growth phase no functionality is removed at all, *i.e.*, no classes have been removed or reduced, the certainty is 1. In this case It is thus clear that the size of the system has been increased. To further characterize system growth phases, we define the classification number which is a measurement to proportionally indicate to what percentage the added functionality consists of added or extended classes. A value of 0 means that new classes have been added but no existing classes have been extended. A value of 1 in contrast means that existing classes have been extended but no new classes have been added.

Definition. The detection of system growth phases is based on the formula 4.3 only: A version v belongs to a system growth phase if $Growth(v) > 0$.

The amplitude of a system growth and a system reduction phase P is defined as the sum of the growth measurement of every version v encapsulated in phase P.

The certainty basically compares the amount of added and removed functionality summed up over a phase. It is the ratio of the amplitude of a phase P and the sum of the $Addition(v)$ measurement for each of its versions v. The certainty value of a system growth phase is between $\frac{1}{2}$ and 1, since $Growth(v)$ is higher than 0 but does not exceed $Addition(v)$.

The classification number is the ratio of the amount of added classes in every version of a phase divided through $Addition(v)$ of every version v in a phase.

$$
\begin{aligned}
&SystemGrowth(S) := \left\{ S' \;\middle|\; S' \subseteq S,\ \forall v \in S'\ Growth(v) > 0 \ \right\} \\
&Amplitude(P) := \sum_{\forall v \in P} |Growth(v)| \\
&Certainty(P) := \sum_{\forall v \in P} \frac{Growth(v)}{Addition(v)} \\
&ClassificationNumber(P) := \sum_{\forall v \in P} \frac{|NewClass(v)|}{Addition(v)}
\end{aligned}
$$

Discussion

- **Concurrent Phases -** General growth phase
- **Asynchronous Phases -** System reduction, code cleaning, code correction and idle phases

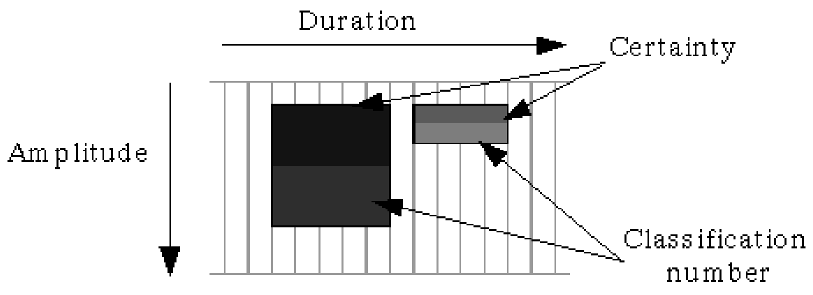


Figure 4.1: Exemplification of System Growth Phases

Exemplification. In Figure 4.1 we see the visualization of two different system growth phases. The width indicates the duration in versions. Accordingly, the left phase consists of 5 versions while the right one is one version shorter.

The height displays the amplitude, *i.e.,* how much the system grew in a phase. Therefore the left phase encapsulates a stronger growth than the right one.

The certainty values of each phase is depicted in the color of the upper stripe. The almost black upper stripe of the left figure displays thus a high certainty values which means that almost no functionality has been removed compared to the amount of added functionality. The right phase in contrast has a brighter upper stripe which means that it is less clear that the system grew in this phase because a similar amount of functionality has been removed and added. Thus other changes than the addition of new classes or methods are encapsulated in this phase.

The classification number is used to determine the lower color. According to the definition of the characteristic, a dark shade of gray as in the left phase indicates that the growth consists mostly of class addition while a light color as in the right figure indicates mostly of class extension. Black would indicate pure class addition (*i.e.,* there are no classes extended but classes added) while white indicates pure class extension.

4.1.2 System Reduction

Motivation. This phase description contradicts the system growth phase description. It aims at detecting when and to what extent the overall system has been reduced. The amplitude of a system reduction phase indicates how much the system has been reduced in this phase while the certainty value indicates how clear it is that the system has been reduced by considering the amount of added functionality. The classification number is a number that indicates to what extent the removed functionality consisted of removed classes and the reduction of classes.

Definition. A system reduction phase is detected if $Growth < 0$.

The amplitude of a system reduction phase P is defined as the sum of the growth measurement of every version v encapsulated in phase P.

The certainty compares the amount of added and removed functionality summed up over a phase. It is the ratio of the amplitude of a phase P and the sum of the $Reduction(v)$ measurement for each of its versions v. The certainty value of a system reduction phase is therefore between $\frac{1}{2}$ and 1 (in a system reduction phase, $Reduction(v) > 0$ and $|Growth(v)| \leq Reduction(v)$ for every encapsulated version v).

The classification number is the ratio of the amount of removed classes in every version of a phase divided through $Reduction(v)$ of every version v in a phase.

$$SystemReduction(S) := \left\{ S' \,\middle|\, S' \subseteq S,\ \forall v \in S'\ Growth(v) < 0 \right\}$$
$$Amplitude(P) := \sum_{\forall v \in P} |Growth(v)|$$
$$Certainty(P) := \sum_{\forall v \in P} \frac{|Growth(v)|}{Reduction(v)}$$
$$ClassificationNumber(P) := \sum_{\forall v \in P} \frac{|RemovedClass(v)|}{Reduction(v)}$$

Discussion

- **Conjunct Phases -** General reduction phase.
- **Asynchronous Phases -** System growth and all maintenance phases.

4.1.3 General Growth

Motivation. The general growth phase description aims at detecting phases where functionality has been added disregarding the change of the overall size of the system. Its definition is based on the definition of the $Addition$ measurement. Detecting growth independently from the overall system size gives together with system growth phases more detailed information about coding activities. General growth phases may overlap one or more system growth phases or they can cover versions that are not part of system growth phases. The latter case indicates that the system did not grow, but yet new functionality has been added. The former case demands further analysis of those versions that are part of the general but not of the system growth phases. These can be detected with a combination of the two phase descriptions.

The amplitude of a general growth phase indicates the amount of the added functionality in the phase. It is defined slightly different than in the system growth phase description, *i.e.*, it disregards the amount of removed functionality and only considers the amount of added functionality. The certainty and classification number are defined identically in all growth phases.

Definition. A phase is detected if there are classes added or extended. Extended and added classes are detected with the appropriate detection strategies *ExtendedClass* and *NewClass* that are described in Appendix B.1 and B.7. A class is considered as extended if there have been methods or attributes added (or both).

The definition of the amplitude is basically equivalent to the system growth phase description; It is the sum of $Addition(v)$ of all versions v in a phase. The certainty and the classification number are defined identically as in the system growth phase description and are thus not further discussed here.

$$GeneralGrowth(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|NewClass(v)| > 0) \vee \\ (|ExtendedClass(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} Addition(v)$$

$$Certainty(P) := \sum_{\forall v \in P} \frac{Growth(v)}{Addition(v)}$$

$$ClassificationNumber(P) := \sum_{\forall v \in P} \frac{|NewClass(v)|}{Addition(v)}$$

Discussion

- **Concurrent Phases -** None.

- **Asynchronous phases -** All maintenance phases.
- **False positives -** Renaming of a classes, a methods or an attributes shows in our model up as the removal the addition of entities. The general growth phase description only considers the addition and thus detects renaming changes.

4.1.4 General Reduction

Motivation. This phase description contradicts the general growth phase description. It aims at detecting reduction in the system history disregarding the change of the overall size. The amplitude measures the amount of removed functionality while the certainty values indicates how clear it is that the system has been reduced by comparing the amount of removed and concurrently added functionality. The classification number is a number that indicates to what extent the removed functionality consisted of removed and reduced classes. A class is considered as reduced if methods or attributes have been removed.

Definition. This phase description detects a phase if are any classes removed or reduced. Removed and Reduced classes are detected with the appropriate detection strategies (see Appendix B.2 and B.8).

The definitions of the amplitude, certainty and classification number are analog to those of the general growth phase description. Instead of considering added and extended classes, the numbers of removed and reduced classes are used.

$$GeneralReduction(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|RemovedClass(v)| > 0) \vee \\ (|ReducedClass(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} Reduction(v)$$

$$Certainty(P) := \sum_{\forall v \in P} \frac{|Growth(v)|}{Addition(v)}$$

$$ClassificationNumber(P) := \sum_{\forall v \in P} \frac{|ReducedClass(v)|}{Reduction(v)}$$

Discussion

- **Concurrent Phases -** None.
- **Asynchronous Phases -** Maintenance phases.
- **False positives -** Renaming of a classes, a methods or an attributes shows in our model up as the removal the addition of entities. The general reduction phase description only considers the addition and thus detects renaming changes.

4.1.5 New Hierarchies Growth

Motivation. With the two previous growth phase descriptions and the defined measurements we could among other things tell when and and how many classes were added to the system. But we could not tell yet what kind of classes have been added. This phase description aims at detecting the addition of new inheritance hierarchies, *i.e.*, phases where new superclasses and inherited classes have been added.

Definition. A new hierarchies phase is detected if there have been new superclasses added and new leaf classes added. A new superclass is an added class that has subclasses (see the corresponding detection strategy *ExtendedClass* in Appendix B.5). Accordingly, a new leaf class is a class that does not have subclasses. New leaf classes are detected with the detection strategy *NewLeafClass* which is defined in Appendix B.6.

The amplitude, certainty and classification number are identically defined to the general growth phase description and thus not further explained here.

$$NewHierarchiesGrowth(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|NewSuperclass(v)| > 0) \wedge \\ (|NewLeafClass(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} Addition(v)$$

$$Certainty(P) := \sum_{\forall v \in P} \frac{Growth(v)}{Addition(v)}$$

$$ClassificationNumber(P) := \sum_{\forall v \in P} \frac{|ExtendedClass(v)|}{Addition(v)}$$

Discussion

- **Concurrent Phases -** General growth phase.
- **Asynchronous Phases -** All maintenance phases.
- **False Positives -** False positives might appear if 1) an already existing superclass is renamed and 2) if the added leaf classes are not subclasses of the added superclasses. In the latter case, instead of adding a complete hierarchy, an abstraction to already existing classes is created. This situation is captured in the split into subclass and in the split into superclass phase descriptions.

- **Improved certainty -** A more informative way to define the certainty would be to compare the number of added superclasses to the total number of added classes.
- **Detection Flaw -** The detection mechanism of this phase description disregards the inheritance relationships between the removed classes which is the cause for one of the cases of false positives.

4.1.6 Class Hierarchy Removal

Motivation. This phase description aims at detecting phases where entire inheritance hierarchies have been removed. It searches for versions where superclasses and leaf classes have been removed. The amplitude of a system reduction phase indicates how much the system has been reduced in this phase. The certainty value indicates how clear it is that the system has been reduced by comparing the amount of added and removed functionality. The classification number is a measurement that indicates to what extent the removed functionality consisted of removed classes and the reduction of classes. The amplitude, the certainty and the classification number measurement are defined equal to the general reduction phase description.

Definition. This phase description detects a phase if there were superclasses and leaf classes removed in a version. Removed superclasses are detected with the detection strategy *RemovedSuperclass* while the removed leave classes are discovered with the *RemovedLeafClass* detection strategy. The amplitude, certainty and classification number are defined equally as in the general reduction phase and are thus not discussed here.

$$HierarchiesRemoval(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|RemovedSuperclass(V)| > 0) \wedge \\ (|RemovedLeafClass(V)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} Reduction(v)$$

$$Certainty(P) := \sum_{\forall v \in P} \frac{|Growth(v)|}{Reduction(v)}$$

$$ClassificationNumber(P) := \sum_{\forall v \in P} \frac{|ReducedClass(v)|}{Reduction(v)}$$

Discussion

- **Conjunct Phases -** General reduction phase.
- **Asynchronous Phases -** Maintenance phases.
- **False Positives -** False Positives might appear if on the one hand complete hierarchies have been renamed or if on the other hand the removed leaf classes are not subclasses of the removed superclasses. The latter case indicates that in one hierarchy one more superclasses but no leaf classes have been removed. This situation is a sign for redesigning activities which is captured in

the *merge with superclass* and in the *merge with subclass* refactoring phase description.

- **Detection Flaw -** The detection mechanism of this phase description disregards the inheritance relationships between the removed classes what is the cause for one of the cases of false positives.

4.2 Maintenance Phase Descriptions

The notion of maintenance generally covers the complete process of changing a system after it has been deployed [Som00]. This notion thus corresponds rather to a stage in the life cycle of a system than to a specific kind of changes. We however use the term maintenance in the latter sense: Our maintenance phase descriptions aim at detecting phases where only small and specific changes have been made or when we could not detect any changes at all.

The detection mechanism of these phase descriptions are based on simple heuristics. Generally, their detection expression excludes other, “bigger” changes such as the addition of new classes. The maintenance phase descriptions aim at detecting phases where only “maintenance changes” have been applied instead of generally detecting “maintenance changes”.

Therefore, if a maintenance phase is detected, no “big” changes have been applied, and the small changes, *i.e.*, the reason for those small changes have to be understood. These however can hardly be understood from analyzing software measurements but only from reading source code. Detected maintenance phases thus do not explain changes from one version to another. However, they tell us that only certain small changes have been made and serve as a vantage point indicating in which versions and in which classes and methods those changes have been made. The reverse engineer can then compare the source code of single class versions and understand the applied changes.

Note that the maintenance phase descriptions encapsulate the most extreme shift from analyzing a large amount of versions of an entire system to single versions to single changes in a method.

We present the following three maintenance phase descriptions:

- **Code Correction** phase description - to detect the correction of errors.
- **Code Cleaning** phase description - to detect the removal of superfluent code.
- **Idle** phase description - to detect phases where no changes are visible in the applied software metrics.

4.2.1 Code Correction

Motivation. This phase description aims at detecting phases where errors in methods were corrected. It searches for versions where there were only a few methods corrected and no other changes made based on the assumption that if a class has the same amount of methods and attributes but statements have been added, the change was correction of error. The amplitude indicates in how many classes errors have been corrected.

Definition The detection is based on the *MSGOnlyExtendedClass* detection strategy (defined in Appendix B.3) which detects class versions that have more methods or attributes but more messages compared to its previous version. A code correction phase is detected if there were no classes added, removed, extended or reduced, but one or more classes are detected by the *MSGOnlyExtendedClass* detection strategy.

$$CodeCorrection(S) := \left\{ S' \,\middle|\, \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|MSGOnlyExtendedClass(v)| > 0)\wedge \\ (|ExtendedClass(v)| = 0)\wedge \\ (|ReducedClass(v)| = 0)\wedge \\ (|NewClass(v)| = 0)\wedge \\ (|RemovedClass(v)| = 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} |MSGOnlyExtendedClass(v)|$$

Discussion

- **Asynchronous Phases -** All, *i.e.,* no other phases appear concurrently.
- **False Positives -** False positives may be detected if other changes have been made to a particular class that in total only show up as a addition of statements. An exemplary case is if one method is added and another smaller one removed.
- **False Negatives -** Situations may not be discovered if other changes have been applied at the same time. An example is the addition of one method.
- **Detection Improvement -** The detection mechanism works on the class level, *i.e.,* it detects classes where only a few messages have been removed. The detection would be better on the method level, *i.e.,* to detect classes where there are methods where only a few statements have been removed. With our definition a situation where in a class errors have been corrected and other code has been removed would, in contrast to the improved expression, not be recognized.

4.2.2 Code Cleaning

Motivation. This phase description also aims at phases where there were only small changes in methods made. In contradiction to the code correction phase description, it looks for classes where superfluous code has been removed. The reason why the code has been removed is what actually leads to an understanding of the applied changes. To understand that reason, the reverse engineer can use the detected phases as vantage points for further inspections.

Definition. This phase description like the previous one implies that no classes have been added, removed, extended or reduced. The detection is then based on the detection strategy *MSGOnlyReducedClass* (defined in Appendix B.4) which searches for classes that have not been extended or reduced in terms of NOA and NOM but where some statements in one or more of its methods have been removed.

Detection Expression

$$CodeCleaning(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|MSGOnlyReducedClass(v)| > 0) \wedge \\ (|ExtendedClass(v)| = 0) \wedge \\ (|ReducedClass(v)| = 0) \wedge \\ (|NewClass(v)| = 0) \wedge \\ (|RemovedClass(v)| = 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} |MSGOnlyReducedClass(v)|$$

Discussion

- **Concurrent Phases -** None.
- **Asynchronous Phases -** All.
- **False Positives -** False positives may be detected if other changes have been made to a particular class that in total only show up as a removal of statements. An exemplary case is if one method is added and another larger one removed.
- **False Negatives -** Situations may not be discovered if other changes have been applied at the same time. An example is the addition of one method.
- **Detection Improvement -** The detection mechanism works on the class level. *i.e.*, it detects classes where only a few messages have been removed. The detection would be better on the method level, *i.e.*, to detect classes where there are methods where only a few statements have been removed. With our definition, a situation where in a class errors have been corrected and other code has been removed would, in contrast to the improved expression, not be recognized.

4.2.3 Idle

Motivation. This phase description detects phases where there were no changes made. Note that the detection is based on measurements only. This means that in an idle phase some changes have been made but they do not show up in the applied measurements.

Definition. This phase description detects a version as part of a phase if there were no classes added or removed, no classes have been extended or reduced and $Growth = 0$.

$$Idle(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|NewClass(v)| = 0)\wedge \\ (|RemovedClass(v)| = 0)\wedge \\ (|ExtendedClass(v)| = 0)\wedge \\ (|ReducedClass(v)| = 0)\wedge \\ (|MSGOnlyExtendedClass(v)| = 0)\wedge \\ (|MSGOnlyReducedClass(v)| = 0) \end{array} \right\}$$

Discussion

- **Concurrent Phases -** None.
- **Asynchronous Phases -** All.
- **False positives -** Situations may be discovered if several changes have been applied that counter-balance in terms of measurements. This situation might be visible in phases of other phase descriptions.

4.3 Refactoring Phases Descriptions

In object-oriented development processes with their emphasis on iterative development [GR95] change is an essential ingredient of system design. Demeyer, Ducasse and Nierstrasz [DDN00] claim that to really understand evolving software, the changes themselves are the critical factor. Changes are in object-oriented development accomplished by means of so-called refactorings [Opd92, FBB$^+$99]. Fowler defines refactoring as a noun as follows:

> **Refactoring** (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [FBB$^+$99].

Fowler stresses that refactoring 1) improves the design of software, 2) makes software easier to understand, 3) helps finding bugs and 4) helps programming faster. On a high level of abstraction, detecting refactorings in the evolution of a software thus indicates when and how much effort has been spent on improving the software's design quality. On a lower level, identifying which refactorings have been applied helps us understand how and - to a certain extent why - the system has evolved [DDN00].

In this section, we present an exemplary set of phase descriptions to detect when and to what extent specific refactorings have been applied in an evolution. In contrast to the growth and reduction phase descriptions, the refactoring phase descriptions detect specific changes using detection heuristics. Refactoring phase descriptions serve basically in two different ways: On the one hand they offer the possibility to discover and localize the application of specific changes and serve as vantage points for further investigations. On the other hand they proved to be helpful on a higher grained level: The frequency of the occurrence of specific refactoring phases already gives valuable information independent from the semantics of the encapsulated changes.

The presented refactoring phase descriptions are all based on heuristics but they vary in their level of exactitude. Some phase descriptions simply compare the amount of added and removed classes while other phase descriptions use detection strategies to detect specifically changed classes based on detection strategies that detect certain methods of that class. The exact identification of the application of specific refactorings is a research area of its own.

The detection mechanism is basically identical in all presented refactorings phase descriptions: Classes where the refactoring has been applied are detected with one or more detection strategies. A phase is then detected if the detection strategy detected one or more entities. The size of the resulting set of detected entities is then used to determine the amplitude of the phase.

4.3.1 Class Renaming

Motivation. The renaming of classes is either a sign for the improvement of the naming or for a better or a new understanding of the created key abstractions. Both cases indicate that work has been done on improving the code.

Definition. The renaming of a class shows in our model up as the removal of a class and the addition of a new one. The renaming of classes can therefore be discovered if about the same amount of classes has been removed and added at the same time. A phase is detected if the fraction of the amount of removed and added classes is between two threshold values. We chose to use $\frac{4}{5}$ and $\frac{5}{4}$.

$$ClassesRenaming(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|NewClass(v)| > 0) \wedge \\ (|RemovedClass(v)| > 0) \wedge \\ \frac{4}{5} < \frac{|NewClass(v)|}{|RemovedClass(v)|} < \frac{5}{4} \end{array} \right\}$$

Discussion

- **Concurrent Phases -** General growth and general reduction phases.
- **Asynchronous Phases -** All maintenance phases.
- **False Positives -** Situations might erroneously be discovered if the addition and the removal of classes occurs but has another reason.
- **False Negatives -** Class renamings might not be discovered if at the same time other classes are added or removed so that the threshold value is exceeded.
- **Improved Detection -** The detection of class renaming could be improved by looking for the removing and adding of classes with (more or less) the same measurements.

4.3.2 Extract Method

Motivation. The extract method refactoring [FBB+99] is a common refactoring. It consists of extracting a piece of code in one or more methods into a new well-named method. The application of this refactoring results in fine-grained and well-named methods which supports the understandability and maintainability of code for several reasons. Firstly, it increases the chances that other methods can reuse a method. Secondly, it allows higher-level methods to be read more like a series of comments. And thirdly, it simplifies the overriding of methods.

The amplitude of a phase indicates in how many classes this refactoring has been applied.

Definition. A system version is detected as part of an extract method phase if there are classes where methods have been added others were reduced. The detection is based on the detection strategy *ExtractMethod* which detects such classes in a system version. This detection strategy is based on another detection strategy, the *ReducedMethod* detection strategy, which filters reduced methods M in a class($\delta MSG(M) < 0$). The mentioned detection strategies are defined in Appendix B.15 and B.17.

The amplitude of a phase is defined as the amount of classes where the refactoring has been applied (in all versions).

$$ClassRefactoring(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|ExtractMethod(v)| > 0) \end{array} \right\}$$
$$Amplitude(P) := \sum_{\forall v \in P} |ExtractMethod(v)|$$

Discussion

- **Concurrent Phases -** General growth phases.
- **Asynchronous Phases -** All maintenance phases.
- **False Positives -** This phase description may detect false positives if the reduction of the methods is unrelated to the newly added methods.
- **False Negatives -** Situations may not be discovered if 1) the extract method refactoring is countered by the removal of methods and 2) if the methods where code has been extracted from were extended at the same time. The first case can be caused by the refactoring itself if the extracted method could be reused in a way that reduces the overall number of methods in the class.
- **Complex Detection -** The detection is time consuming even though this phase description looks simple: The *ReducedMethod* detection strategy is applied on every method in every class version in every version.

4.3.3 Split into Superclass

Motivation. This phase description searches for refactorings that optimize the class hierarchy by splitting functionality from a class into a newly created superclass. Thus, it looks for the creation of a superclass together with a number of pull-ups of methods or attributes. The description of the refactoring and the detection heuristic is taken from Demeyer, Ducasse and Nierstrasz [DDN00].

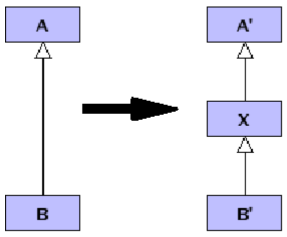


Figure 4.2: Split into Superclass Refactoring

Definition. The detection is based on the detection strategy *SplitIntoSuperclass* (defined in Appendix B.9) that identifies class versions where a superclass has been added and functionality has been pulled up, *i.e.,* class versions which are nested deeper in the inheritance hierarchy and have less methods, attributes or class variables compared to their previous version. A version is part of a split into superclass phase if the detection strategy *SplitIntoSuperclass* identifies one or more classes for that version.

$$SplitIntoSuperclass(S) := \left\{ S' \,\middle|\, \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|SplitIntoSuperclass(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} |SplitIntoSuperclass(v)|$$

Discussion

- **Concurrent Phases -** General growth phases.
- **Asynchronous Phases -** Maintenance phases.
- **False negatives -** The heuristic may fail when the change in the hierachy nesting level of a class is countered by an unrelated removal of a superclass, or when the pull-ups is countered by an equal addition or removal of new functionality. The former case will show up as false positives of the *move to other class* phase description.
- **False positives -** The heuristic may discover situations that do not correspond to a split if part of the class functionality has been rearranged, superimposed by an unrelated addition of a superclass.

4.3.4 Merge With Superclass

Motivation. This phase description aims at detecting refactorings that optimize the class hierarchy by merging a superclass and one or more subclasses. Thus, is looks for the removal of a superclass, together with a push-downs of methods or attributes. The description of the refactoring and the detection heuristic is taken from Demeyer, Ducasse and Nierstrasz [DDN00].

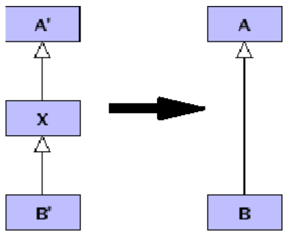


Figure 4.3: Merge with Superclass Refactoring

Definition. The definition of this phase description is based on the detection strategy *MergeWithSuperclass* (defined in Appendix B.10) which identifies classes where a superclass has been removed and functionality has been pushed down. More precisely it detects a class version if its hierarchy nesting level is lower and it contains more methods, attributes or class variables than its previous version. A phase is detected if the detection strategy *MergeWithSuperclass* identifies one or more suspects.

$$MergeWithSuperclass(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|MergeWithSuperclass(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} |MergeWithSuperclass(v)|$$

Discussion

- **Concurrent Phases -** General reduction phases.
- **Asynchronous Phases -** All maintenance phases.
- **False negatives -** This heuristic may fail to detect a split or merge, when the change in HNL is countered by an inverse change higher up in the hierarchy, or when the push-down is countered by an equal removal of functionality.
- **False positives -** This heuristic may discover situations that do not correspond to a merge if part of the class functionality has been rearranged, superimposed by an unrelated removal of a superclass.

4.3.5 Split into Subclass

Motivation. This phase description aims at detecting optimizations of the class hierarchy by splitting functionality from a class into one or more newly created subclasses. Thus, it looks for the creation of new subclasses together with a number of pull-ups of methods and attributes. The split into subclass refactoring complies partially with the extract method refactoring in [FBB$^+$99]. The description of the refactoring and the detection heuristic is taken from Demeyer, Ducasse and Nierstrasz [DDN00].

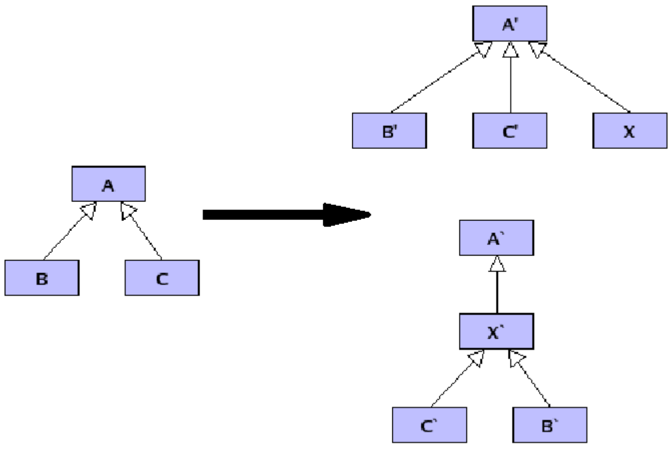


Figure 4.4: Split into Subclass Refactoring

Definition. The detection is based on the *SplitIntoSubclass* detection strategy (defined in Appendix B.11) that identifies classes where one ore more subclasses have been added and functionality has been pulled up. More precisely, it looks for class versions that have more subclasses but less methods, attributes or class variables than the previous version.

$$SplitIntoSubclass(S) := \left\{ S' \middle| \begin{array}{l} S' \subseteq S,\ \forall V \in S' \\ (|SplitIntoSubclass(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} |SplitIntoSubclass(v)|$$

Discussion

- **Concurrent Phases** - General growth phases.
- **Asynchronous Phases** - All maintenance phases.
- **False negatives.** The heuristic may fail to detect a split or a merge, when the refactoring did not involve a change in the number of children of a class,

or when the pull-up is countered by an equal addition or removal of new functionality. Sometimes, these cases show up as false positives of the *move to other class* phase description.

- **False negatives.** The heuristic may discover classes that are not split or merged, most often when class functionality has been added, moved or removed and at the same time unrelated subclasses have been added. Thus, sometimes the false positive does correspond with a false negative of the *move to other class* phase description.

4.3.6 Merge with Subclass

Motivation. This phase description aims at detecting changes that merge superclass with one or more of its children. Thus, it searches removal of subclasses together with a number of push-downs of methods or attributes. The description of the refactoring and the detection heuristic is taken from Demeyer, Ducasse and Nierstrasz [DDN00].

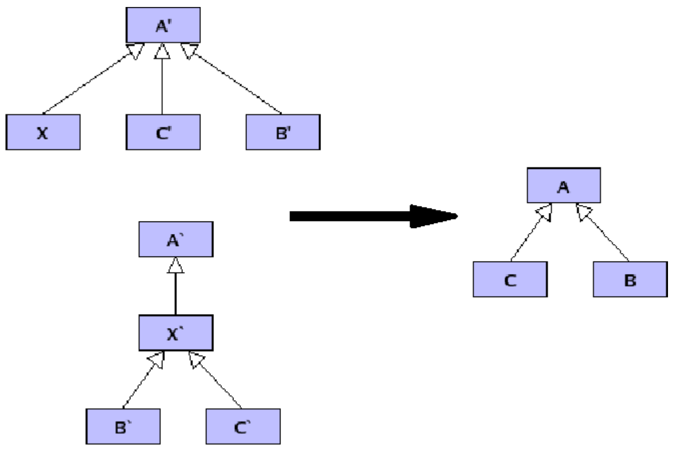


Figure 4.5: Merge with Subclass Refactoring

Definition. The detection of a Merge with Subclass phase is based on the *MergeWithSubclass* detection strategy which is defined in Appendix B.12. This detection strategy identifies class versions that have less or more children and more methods, attributes or class variables compared to it previous version.

Detection Expression

$$MergeWithSubclass(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|MergeWithSubclass(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} |MergeWithSubclass(v)|$$

Discussion

- **Concurrent Phases -** General reduction phases.
- **Asynchronous Phases -** All maintenance phases.
- **False negatives.** The detection heuristic may fail to detect a merge, when the refactoring did not involve a change in NOC, or when the push-down is countered by an equal removal of functionality. These cases sometimes will show up as false positives of the *move to other class* phase description.

4.3.7 Move To Other Class

Motivation. This phase description searches for refactorings that move functionality from one class to another. This other class may be either a subclass, a superclass, or a sibling class (*i.e.,* a class which does not participate in an inheritance relationship with the target class). Accordingly we look for classes where methods, instance variables or class variables have been removed and at the same time for classes where functionally has been added. The detection mechanism for this refactoring was basically taken from [DDN00] and slightly improved.

Definition. The detection is based on the detection strategies *MoveToOtherClassSource* and *MoveToOtherClassDest* (presented in Appendix B.13 and B.14). The former identifies classes where functionality has been removed while the latter identifies classes where functionality has been added. A version is considered as part of a *move to other class* phase if both detection strategies detected some class versions.

$$MoveToOtherClass(S) := \left\{ S' \;\middle|\; \begin{array}{l} S' \subseteq S,\ \forall v \in S' \\ (|MoveToOtherClassSource(v)| > 0) \wedge \\ (|MoveToOtherClassDest(v)| > 0) \end{array} \right\}$$

$$Amplitude(P) := \sum_{\forall v \in P} |MoveToOtherClass(v)|$$

Discussion

- **Concurrent Phases -** General growth and general reduction phases.
- **Asynchronous Phases -** All maintenance phases.
- **False negatives -** The heuristic may fail to detect a move when it was countered by an equal addition or removal of functionality.
- **False positives -** The heuristic may discover cases that do not correspond to a move of functionality but then it detects symptoms of other refactorings. The functionality may be removed instead of moved, or it may have ben replaced.

Chapter 5

Phase Descriptions on Class Histories

This chapter contains 7 basic phase descriptions that detect phases in the history of a class. We present six phase descriptions to detect changes based on one single metric only and one to detect idle phases:

- The **method addition and removal** phase descriptions detect phases where methods have been added respectively removed.
- The **attribute addition and removal** phase descriptions detect phases where attributes have been added respectively removed in a class.
- The **statements addition and removal** phase descriptions detect phases where the number of statements in a class increased respectively decreased.
- The **idle** phase description detects phases where no changes have been made in a class.

These phase descriptions do not aim at detecting specific changes on their own but in combination with others they can be used for that. For example, the split method refactoring would show up in a method addition phase and a concurrent statements removal phase. The coevally occurrence of these phases could be detected either visually or by combining the method addition and the statements removal phase descriptions with an AND composition operator.

Except for the idle phase description, we define the amplitude for each phase description.

5.1 Method Addition and Removal Phase Descriptions

Motivation. The method addition respectively removal phase description detects when and how many methods have been added respectively removed in the history of a class.

Definition. A version is detected as part of a method addition phase if it contains a higher amount of methods than its previous version. Analogical, a method removal phase encapsulate class versions that contain less methods that their predecessor. Note that class variables are disregarded.

The amplitude of a phase indicates how many methods have been added respectively removed in all encapsulated versions.

$$\begin{aligned}
&MethodAddition(H) := \left\{ H' \middle| \; H' \subseteq H, \, \forall v \in H' \, \delta NOM(v) > 0 \; \right\} \\
&MethodRemoval(H) := \left\{ H' \middle| \; H' \subseteq H, \, \forall v \in H' \, \delta NOM(v) < 0 \; \right\} \\
&Amplitude(P) := \sum_{\forall v \in P} |\delta NOM(v)|
\end{aligned}$$

5.2 Attribute Addition and Removal Phase Descriptions

Motivation. The attribute addition and the attribute removal phase descriptions detect when and how many attributes have been added respectively removed in the history of a class. Note that the number of attributes includes both instance and class variables.

Definition. A class version is detected by a attribute addition $\setminus$ removal phase description if it contains more $\setminus$ less attributes than its preceding class version.

The amplitude of a phase indicates how many attributes have been removed respectively added in all encapsulated versions.

$$\begin{aligned}
&AttributeAddition(H) := \left\{ H' \middle| \; H' \subseteq H, \, \forall v \in H' \, \delta NOA(V) > 0 \; \right\} \\
&AttributeRemoval(H) := \left\{ H' \middle| \; H' \subseteq H, \, \forall v \in H' \, \delta NOA(V) < 0 \; \right\} \\
&Amplitude(P) := \sum_{\forall v \in P} |\delta NOA(v)|
\end{aligned}$$

5.3 Statement Addition and Removal Phase Descriptions

Motivation. These phase descriptions detects phases in the history of a class where statements have been added respectively removed.

Definition. A class version is detected by this phase description if it contains more respectively less statements than its predecessor. The number of statements is the sum of all the statements in all methods of a class.

The amplitude of a phase indicates how many statements have been added \ removed in all encapsulated versions.

$$StatementAddition(H) := \left\{H' \middle| \; H' \subseteq H, \, \forall v \in H' \, \delta NOS(V) > 0 \; \right\}$$
$$StatementRemoval(H) := \left\{H' \middle| \; H' \subseteq H, \, \forall v \in H' \, \delta NOS(V) < 0 \; \right\}$$
$$Amplitude(P) := \sum_{\forall v \in P} |\delta NOS(v)|$$

5.4 Idle Phase Description

Motivation. The idle phase description aims at detecting phases where no changes have been made to a class.

Definition. A version is detected as part of a phase if no methods, attributes of statements have been added or removed in a class.

$$Idle(H) := \left\{ H' \middle| \begin{array}{l} H' \subseteq H, \, \forall v \in H' \, \delta NOM(V) = 0 \wedge \\ \delta NOA(V) = 0 \;\; \& \;\; \delta MSG(V) = 0 \end{array} \right\}$$
$$Amplitude(P) := \sum_{\forall v \in P} \delta NOM(v)$$

Chapter 6

Detecting Phases in System Histories

In this chapter, we apply the catalog of phase descriptions on system histories on the evolution of two different applications: First on Jun and then on SmallWiki. In Jun, we analyze 77 consecutive versions. We present the visualization of all detected phases and then demonstrate how this visualization can be read and how it can be used to understand the evolution. In the analysis of SmallWiki, we inspect almost all versions of SmallWiki from its beginning up to the start of the development of SmallWiki2. We present three different usages of the detected phases. First, we present the computed measurements and show how they can be interpreted. Then we show the visualization of the entire evolution of SmallWiki. Based on this visualization we then distinguish four different stages and characterize them. Finally, we use the detected phases as vantage points for more detailed inspections. That is, we show how phases can be used to recover the formation of the core hierarchies of SmallWiki.

6.1 Visualizing Phases in the Evolution of Jun

Jun[1] is a large open source 3D-graphics framework written in VisualWorks [2] Smalltalk. Jun is open source software developed within a for-profit company. However, almost all of Jun was developed by a small group of three to five programmers at a time [AHK$^+$01]. The development of Jun is highly dependent of one chief programmer who is solely responsible for integrating added portions of code created by team members or the community into an officially released Jun version upgrade.

Though the open source community did not provide much source code, it did provide feedback, feature requests and bug notices. The evolution of Jun is however not simply driven by feedback from the community. Several large-scale

[1]see http://www.sra.co.jp/people/aoki/Jun/Main_e.htm for more information
[2]http://www.cincom.com/smalltalk

projects using Jun identified new needs for it, which also guided the evolution of Jun.

The Jun project was started in 1996 and is still under development. It has mainly grown in small, incremental steps with more than one release a week. We analyze Jun from version 40 to version 120 which corresponds to a time span of 8 months. In this time span, Jun grew from 106 to 239 classes.

Preparation Problems. The parsing and compiling of the source code was a problem itself because it is written for obsolete versions of the VisualWorks environment. As a consequence of that, there were faulty models which first had to be identified and put out of consideration. The versions put out of consideration are the versions 41, 42, 63, 68, and 73.

Another problem was caused by different line feeds in the source code. The VisualWorks parser we applied created extra empty lines between every line of code, but not in all methods. This caused faults in the measurements of the LOC (lines of code) metrics. Thus, we set the LOC metric aside. Another solution to this problem would have been to preprocess all the source files and replace all the faulty line feed with one that would have been recognized properly.

The Visualization of Jun. Figure 6.1 depicts the detected phases in the analyzed time span. We focus on three parts of this time span that show a different evolution and demonstrate how the visualization can be used to understand the evolution in these parts.

6.1.1 The Time Span From Version 40 to 44

In the time span from version 40 to 44 (depicted in detail in Figure 6.2), we find a system growth phase that has a length of two and thus encapsulates two version shifts. Note that the versions 41 and 42 are taken out of consideration. Thus, the phase encapsulates the version shifts from version 40 to 43 and from 43 to 44. From the distances between the vertical lines that represent the versions, we can see that the first version shift lasted much longer than the second. This system growth phase has a black upper stripe which indicates a certainty value of 1.0 (the exact measurement value can be obtained by interacting with the visualized phase). According to the definition of the certainty measurement of system growth phases, this indicates that there was only code added and no code removed at all in this phase. More exactly, there were no classes removed and no existing classes reduced, but new classes added and existing ones extended. This is also visible in the fact that there is no parallel general reduction phase which would be detected if code had been removed. The color of the lower stripe of the phase displays the classification number. It is a dark shade of gray which is hardly distinguishable from color in the upper stripe. This indicates that in this phase, some classes have been extended, but mostly new classes have been added. More exactly, it displays

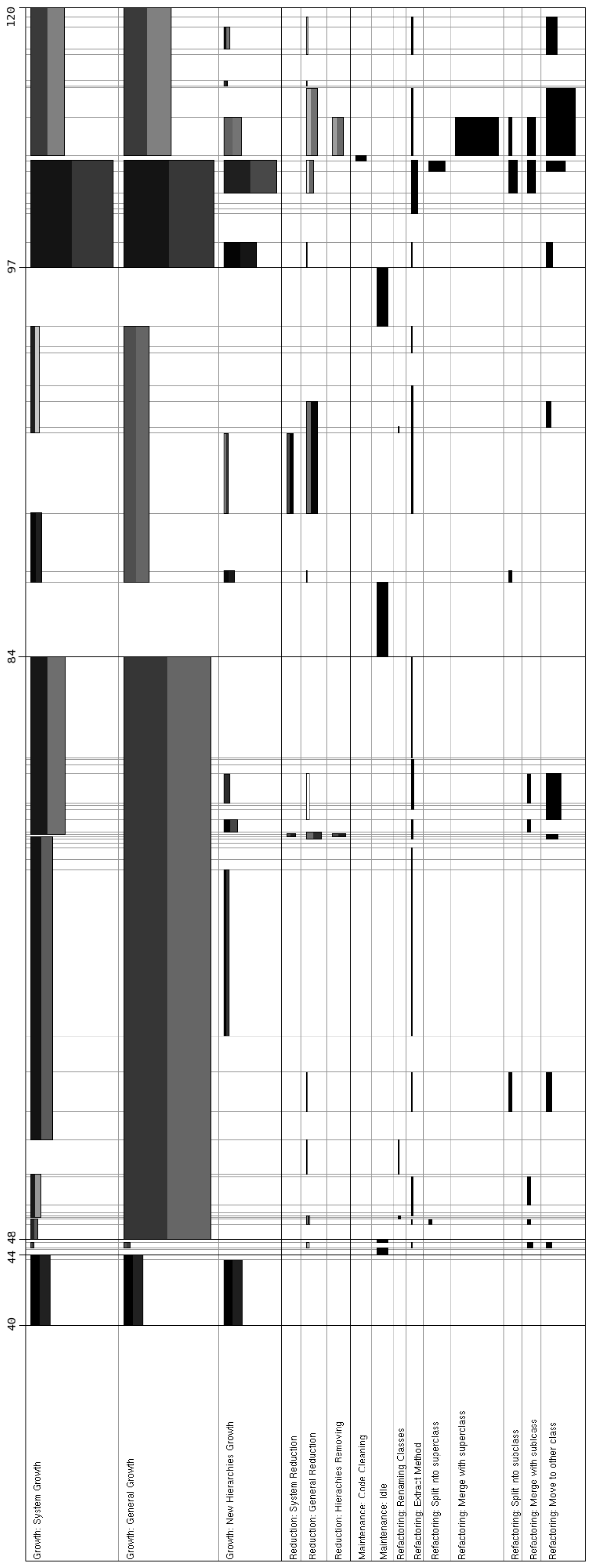


Figure 6.1: Phases in Jun from version 40 to 120

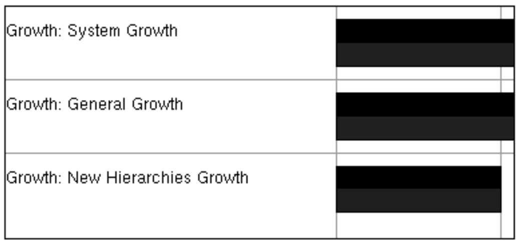


Figure 6.2: Growth phases from version 40 to 44

a classification number of 0.87 which means that the encapsulated growth consists to 87% of the addition of new classes.

In parallel to the considered system growth phase, there is a general growth phase which has identical measurement values. Because of the way these two phase descriptions are defined, there is always a general growth phase parallel to a system growth or a new hierarchies growth phase. That the general growth phase has identical measurements is based on the fact that in this time span there is only new code added and that the phases have the same length. If there was at the same time code removed, the general growth phase would have a higher amplitude since it does not consider the removal, but only the addition of code.

In parallel to the considered system growth phase, there is also a new hierarchies growth phase which indicates the addition of complete new hierarchies. The length of this phase is however 1 smaller, but the amplitude is almost equal since the two rectangles have almost the same height. This lets us conclude that the growth has been made mostly in the first part of the system growth and consists of the addition of at least one hierarchy (super- and leaf classes) and possibly the addition and extension of classes that are not part of this hierarchy. The second versions shift encapsulated in the system and the general growth phase only captures minor changes.

6.1.2 The Time Span form Version 48 to 84

The time span between the versions 48 and 84 (depicted in detail in Figure 6.3) is covered with one single general growth phase. This means that in every version code has been added. But the system does not grow in every encapsulated version since the general growth phase encapsulates time spans that are not covered by a system growth phase. These system growth phases cover most of the depicted time span which shows that the system mostly grows.

The first system growth phase has a length of two. In the first encapsulated version shift, there is no other except the general growth phase in parallel. This indicates that there were only classes extended and leaf classes added and no other changes have been made. In the second encapsulated version shift, there is a parallel general reduction phase which indicates that in this time span there was some

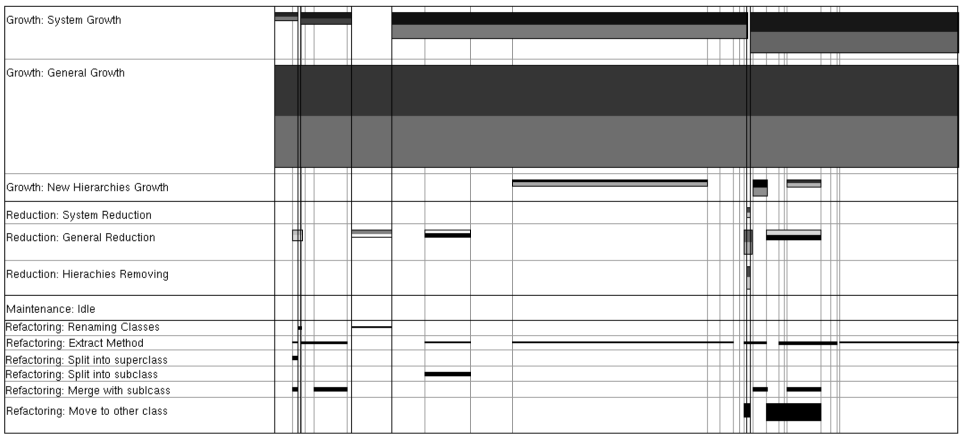


Figure 6.3: Phases in Jun from version 48 to 84

code removed. Additionally, there are three concurrent refactoring phases. Without interpreting the phases in detail, we can conclude that in this phase, the system has been growing but also changes to existing structures have been made.

The first version shift that is not covered by a system growth phase is very short. It is covered by a renaming classes phase and a general reduction phase. Renaming classes shows in our model up as the removal and the addition of the same amount of classes (with different names). Thus, a renaming classes phase always implicates general reduction and a general growth phase in parallel. The fact that these are the only parallel phases means that an exact equal amount of classes has been removed and added which implies that the renaming has been the only change. The renaming class phase has an amplitude of 1 which indicates that one single class has been renamed. To find out what class has been renamed, we can apply the detection strategies *NewClass* and *RemovedClass* on the corresponding versions. Doing so, we find out that the only change that has been made is the remaining of the class *JunBrowseEnhance* into *JunBrowserEnhancement*. The renaming classes phase is however longer than the considered time span, *i.e.*, it overlaps with the next system growth phase. It thus shows that in the next version shifts, more classes have been renamed.

After the discussed renaming phase, there is again a system growth phase. In parallel, there is an extract method and a merge with subclass phase. Thus, besides adding new code, refactorings have been applied.

This system growth phase is again ended by a class renaming phase. This class renaming phase has a concurrent general reduction phase with the same length and duration. Its amplitude is identical and its classification number is 0 (visible in the white lower stripe) which means that in this phase, only classes have been removed and no classes have been reduced. We can thus conclude that in this phase, one class has been added and one class has been removed.

After this renaming phase, there is again a system growth phase. It has a length of 9 and thus encapsulates 9 version shifts. The encapsulated version shifts however do not show identical changes. In the first version shift there was only code added since there are no other concurrent phases. In the following version shift, there are multiple phases with different phase descriptions in parallel. This indicates that this version shift captures more complex changes than the previous one. The following version shift shows again a pure addition of new code. The fourth version shift however shows a parallel new hierarchies growth phase which indicates that there was one or more superclasses and one or more leaf classes added. The remaining version shifts in this system growth phase again consists of pure code addition.

This system growth phase is ended by a short system reduction phase which has a certainty value of 0.71. This shows that in this time span, there was also functionality added but in total more removed. The classification number which is displayed in the color of the lower stripe shows that mostly methods in classes have been removed and only a few classes have been removed. In parallel to this phase, there is a move to other class phase which implies that methods have been moved between classes. These facts can be interpreted as follows: Methods have been moved so that duplicated code could be removed. To understand the changes in detail and to prove our interpretation, we had to compare the source code of the encapsulated versions. The phases and the underlying detection strategies help finding out where to start, *i.e.,* pointing to the methods that have been removed respectively added.

The rest of the discussed general growth phase is covered by another system growth phase from version 71 to 84 . It shows a similar characteristic as the previous system growth phase, *i.e.,* version shifts with pure addition and version shifts with more complex changes.

6.1.3 The Time Span from Version 97 to 120

The time span between the versions 97 and 120 is shown in detail in Figure 6.4. We see that this time span is almost entirely covered by system growth and general growth phases. The high amplitudes of these phases indicates that a lot of code has been added. In numbers, 159 classes and in total 1541 methods have been added. In this time span, there are however also several concurrent phases of different refactoring and reduction phase descriptions. This means that concurrently to the addition of new code, existing structures have been modified indicating that the addition of the new code caused changes in existing code.

For example, parallel to the new hierarchies growth phase between version 104 and 106, there are among others a split into superclass and a merge with subclass phase. This indicates that parallel to the addition of new classes, there were the one hand classes removed in one hierarchy and on the other hand added to another hierarchy. The removing of the classes triggers the general reduction phase.

Between the versions 107 and 110, we find concurrent to the new hierarchies growth phase among others a merge with superclass phase that has a notably high

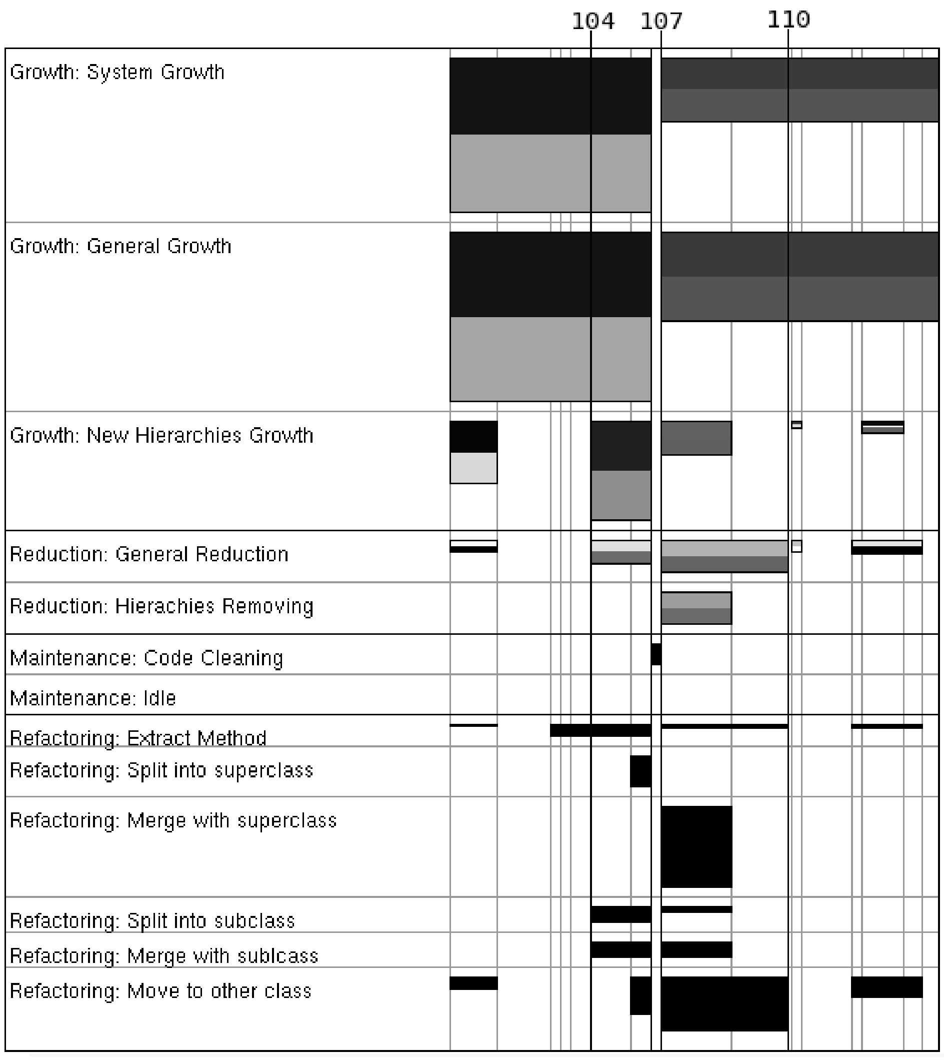


Figure 6.4: Phases in Jun from Version 97 to 120

amplitude. The merge with superclass refactoring detects situations where code has been moved from a superclass into a subclass and the superclass then has been removed. This phase has an amplitude of 16 which implies that the superclass of 16 classes has been removed, showing up in a concurrent hierarchies removal phase. Additional, there is a move to other class what indicates that functionality has been moved between classes. Furthermore, there are a split into subclass and a merge with subclass phase in parallel.

We can conclude that in this time span, many different changes have been made concurrently. We can use the detected phases to quickly find out what parts of the system have been affected in the changes. For that we basically have to "grab" a phase and "look inside". For example, we can find out what hierarchies have been added in the new hierarchies growth phase between version 97 and 99 by applying the detection strategies *NewSuperclass* and *NewLeafClass* on the version 99. With the *NewSuperclass* detection strategy, we obtain that 8 superclasses have been added. The name of all of them starts with "JunVrml". Additionally, there were 20 leaf classes added. The name of 18 of them starts with "JunVrml". This lets us conclude that the main change in this time span was the addition of "Vrml"-functionality [3]. Inspecting in a similar way the entire time span from version 97 to 120, we find out that almost all changes in this time span correspond to this "Vrml"-functionality. This lets us conclude that in the time span from version 97 to 120, a major part of jun has been added and that this addition caused the application of refactorings, that is, changes to existing classes.

6.1.4 Conclusion

We demonstrated how the visualization of phases can be used to get an understanding of the evolution of software system, in this case of Jun. We showed the visualization of a set of versions and explained three parts of this visualization in more detail. We however stayed on a high level and did not aim at understanding the changes in detail.

[3]The **V**irtual **R**eality **M**odelling **L**anguage is a file format for describing interacting 3D objects and worlds to be experienced on the world wide web

6.2 Analyzing the Evolution of SmallWiki

SmallWiki [Ren03] is a fully object-oriented and open source Wiki [4] framework written in VisualWorks Smalltalk. SmallWiki is unlike most other Wiki implementations designed for extensibility. It mainly attains this goal by 1) providing an object-oriented domain model, *e.g.*, the content of a page is stored in a tree of different entities, 2) the possibility to extend this domain model with plug-ins that can be shared and loaded independently, and 3) a high coverage of unit tests.

SmallWiki was mainly created by one single developer; only a few versions have been committed by others. Originally, it has been initiated as a student project at the Software Composition Group of the University of Bern, but was then successfully deployed and moved to a broader scope. Currently, the successor SmallWiki2 is under development, which provides an improved model. We however analyzed the almost complete life cycle of SmallWiki1.

Basic parameters. For this case study, we looked at the almost complete life cycle of SmallWiki1. We analyzed every single version from version 1.1 to 1.313 of, in total 310 versions. This corresponds to a time span of 18 months (from November 6th 2002 to May 18th 2004) in which SmallWiki grew from 8 to 238 classes. The average time span between two versions is about 43 hours. We loaded each of the analyzed versions from the store repository of the Software Composition Group. Based on the loaded and compiled source code we created a FAMIX model for each version. The resulting models are then used to create the HISMO model, that is, a history containing all versions.

Preparation Problems. SmallWiki is based on the libraries Swazoo and Sixx. In early versions, these libraries were included in SmallWiki. At some point in time, these libraries were taken out of SmallWiki and became prerequisites. But since our goal is to analyze the evolution of SmallWiki and not the one of its underlying libraries, we had to disregard those libraries in our models. Concretely, we only considered those classes which are in the name space of SmallWiki since Swazoo and Sixx have their own name space.

Creating the FAMIX models required loading and compiling every single version. This however caused in some of the early versions dependency problems which impact the correctness of the corresponding models. For our analysis, these impacts however do not depict a relevant limitation.

Structure of the case study. In this case study, we demonstrate how phase descriptions can be used to analyze the evolution of an object-oriented system on different levels. First we analyze the defined measurements on phase descriptions on the complete history. This gives us overall information about the evolution of

[4] A Wiki is a collaborative software used to create, edit and manage hypertext pages on a network. It enables the users to author their documents using a simple markup language within a web browser

SmallWiki. Then we present the visualization. By viewing this visualization on a high level, we then detect stages in the evolution of SmallWiki. On the finest level, we demonstrate how the detected phases can be used to get detailed information, *i.e.*, we show how phases can be used to extract the formation of the architecture.

6.2.1 Properties of Phase Descriptions

In a first step, we analyze the measurements we defined based on phases respectively on phase descriptions. Thus, we detect all phases in the complete history and then compute the measurements. These measurements, presented in Table 6.1, could be used to get information about the phase descriptions themselves. We however restrict ourselves to demonstrate what information is revealed about the evolution of SmallWiki:

Growth outspread over most versions. The system growth phases cover more than half of the versions and more than 62% of the analyzed time span. This is not surprising since SmallWiki grew from 8 to 238 classes in the analyzed time span. However, it tells us that the growth is outspread over most of its versions and is not concentrated in only a few. The coverage values of the general growth phases are even higher: in almost 70% of the versions and about 80% of the analyzed time span there was functionality added. The difference of the coverage values of these two phase descriptions tells us that there are time spans where code has been added but the system shrunk. These phases could be detected with the following combination: $GeneralGrowth\ AND\ NOT(SystemGrowth)$.

Growth phases last longer than other phases. The growth phases coverage in time are higher than the coverage in versions. Since the time coverage is a relative measurement, this means that growth phases generally last longer than other phases. Outstanding is the big difference of the coverage values of the new hierarchies growth phase description, *i.e.*, the time coverage is about 4 times higher than the version coverage. This tells us that phases where new hierarchies have been introduced capture particular time consuming implementation tasks. This is backed up with the high average density and average duration measurements.

New hierarchy growth phases capture time consuming, big changes. New hierarchies growth phases have in average the highest amplitude than all other growth phases and thus capture the biggest addition of code. But they have an average length of slightly more than 2 which is shorter than any other growth phases. That is, they capture the biggest changes of all growth phases in the smallest amount of versions. The new hierarchies growth phases however have the longest duration and the highest density. Thus, in those phases, the time between two versions was the longest.

Phase Description	**#Phases**	**Versions Coverage**	**Time Coverage**	**ϕLength**	**ϕDensity**	**ϕDuration**	**ϕAmplitude**	**ϕCertainty**	**ϕCNumber**
System Growth	65	51.9%	62.3%	3.48	85.52	128.43	8.20	0.89	0.44
General Growth	59	69.0%	80.1%	4.63	107.57	182.09	14.78	0.70	0.46
New Hierarchies Growth	21	8.4%	32.0%	2.18	191.25	195.18	17.23	0.76	0.80
System Reduction	43	19.0%	15.6%	2.37	34.97	48.49	2.98	0.77	0.36
General Reduction	68	44.8%	58.4%	3.04	54.52	115.09	6.87	0.48	0.37
Hierarchies Removing	11	4.8%	5.3%	2.36	42.82	65.09	16.09	0.56	0.88
Code Correction	19	6.5%	6.6%	2.05	46.24	46.53			
Code Cleaning	5	1.9%	0.5%	2.20	9.40	13.20			
Idle	40	16.8%	11.4%	2.30	33.05	38.05			
Renaming Classes	14	4.5%	10.5%	2.00	100.78	100.79	5.36		
Extract Method	63	69.7%	77.5%	4.43	87.95	165.03	1.67		
Split into superclass	10	3.2%	24.5%	2.00	328.50	328.50	7.50		
Merge with superclass	7	2.3%	4.2%	2.00	80.00	80.00	10.71		
Split into subclass	13	5.8%	4.6%	2.38	40.62	47.77	8.85		
Merge with subclass	22	7.7%	29.7%	2.09	180.50	180.86	7.73		
Move to other class	61	30.3%	43.7%	2.54	84.71	96.08	11.26		

Table 6.1: Phase Descriptions Overview in SmallWiki

The shape of this addition is further characterized with the classification number. which is higher than in any other growth phase description. This reveals that in new hierarchy growth phases the class addition proportion of the added functionally is higher than in other growth phases. The other growth phase descriptions have much higher classification numbers and thus have a much higher “degree of class extension”. Thus, new hierarchy growth phases encapsulate a big addition of functionality, which comprises only a small amount of versions but a long time span and captures mostly the addition of new classes.

Hierarchy removal phases differ from new hierarchies phases. The hierarchy removal phase description is opposed to the new hierarchy growth phase description, that is, it captures the removal instead of the addition of entire inheritance hierarchies. However, the measurements are considerably different. Hierarchy removal phases in average have about the same length (*i.e.*,, last the same amount of versions), but have a density which is almost five times lower (and accordingly have a lower duration). Thus, the time between two versions is almost 5 times shorter than in the new hierarchies growth phase. The amplitude of the hierarchy removal phases however is in average approximately the same. Hierarchy removal phases thus represent big subtraction of functionality which lasted only a fraction of the time needed to implement. Subtracting functionality however implied adjusting remaining parts and moving code between classes. This is shown by the fact that the average certainty of the hierarchy removal phases is considerably lower than in the new hierarchies phases. The big difference of the version coverage and the time coverage in the new hierarchies growth phase description is furthermore not present in the hierarchies removal phase description. Note that the comparison between new hierarchy phases and hierarchy removal phases is based on the adequate definitions of the amplitude, certainty and classification number, which is not implied by the concept of phases and phase descriptions.

System reduction phases differ from system growth phases. System Reduction phases only cover 16% of the analzed time, but there are more than 40 phases. This means that in about 16% of the analyzed time span, SmallWiki has been reduced. However, the reduction phases last compared to system growth phases short and have a lower amplitude and density. This reveals that the removing of parts has been done often, but in average, there was not much removed.

Further information of the shape of the encapsulated change is revealed by the classification number which is lower than in the system growth phases. The definition of the classification number is in both phase descriptions equivalent. We can thus conclude that the class removal portion is lower in the system reduction phases than the class addition portion in the system growth phases. The classification number of the system reduction phases, which is the lowest at all, reveals that mostly classes have been reduced, that is, methods in classes have been removed. Concluding, we can say that the system reduction phases cover short but intensive removing of superfluent code.

System reduction phases differ from hierarchies removal phases. The system reduction and the hierarchies removal phases have approximately the same length which means that they encapsulate in average approximately the same amount of versions. The hierarchies removal phases however have an amplitude which is in average about 5 times higher and thus capture bigger changes. The classification number of the hierarchies removal phase is 0.88 which means that mostly classes have been removed in these phases and that the amount of reduced classes is small. The classification number of the system reduction phase is however more than 2 times smaller which means that the amount of removed classes is small compared to the amount of reduced classes in these phases. Thus, the hierarchies removal phases capture not only bigger changes but also different kind of changes than system reduction phases.

Removal essential. The general reduction phases cover almost 60% of the analyzed time span, *i.e.,* in almost 60% of the life cycle (40% of the versions). This indicates that the removing of functionality, *i.e.,* methods, attributes or entire classes, enfolds the bigger part of the evolution of SmallWiki and is not concentrated in a few versions. It is an essential part of the evolution of SmallWiki even though it grew in the analyzed time span. General reduction must often overlap with system growth and general growth phases, since the time coverage values of all three phase descriptions is higher than 50%.

General reduction and general growth as opposites. The general growth and the general reduction phase description are defined oppositional to each other. However, general growth phases last considerably longer (in terms of version and in terms of time) and also have a higher average density. This is a sign for the dominance of the growth in the analyzed time span.

Short maintenance phases. Maintenance Phases have compared to other phases a short duration and also a low density. Generally, they have a lower time coverage than version coverage. Thus, the captured small changes could be done relatively quick. Remarkable are the idle and the code cleaning phase description. There are many detected idle phases (40) with a low time coverage. The idle phases encapsulate the application of small changes that did not show up in our measurements at all. The low time coverage and the small duration now indicate that these changes were not time consuming. The code cleaning phases stick out because of their low average density and low average duration measurements. Phases where only superfluous statements in methods have been removed are in fact the shortest we detected in the evolution of SmallWiki.

Refactoring Phases. Refactoring phases aim at detecting the occurrence of specific changes. We see that most refactoring phases cover only a small amount of time and versions but some stick out. Especially the extract method refactoring phases cover the bigger part of the analyzed evolution. The split into superclass refactoring phases cover only a small amount of versions but a much bigger amount of time, which is backed up by the very high average density. These phases thus represent especially time consuming activities. The same shows up in the merge with subclass phases, but to a lower extent. The move to other class phases cover like the extract method phases an outstanding percentage of the analyzed time and versions.

Generally, Table 6.1 reveals that the move to other class and especially the extract method capture a bigger amount of versions and of the analyzed time span than the other refactorings. The split into superclass and the merge with subclass have a notably higher coverage in time than in versions which indicates that they capture more time consuming changes. The extract method phases which cover more than 75% of the analyzed time span have however a low amplitude which means that the underlying refactoring has been applied in most version shifts but in average only a few times.

6.2.2 Visualizing Phases in SmallWiki

After analyzing the evolution of SmallWiki by interpreting measurements based on phases and phase descriptions, we now show the visualization of the detected phases in Figure 6.5 as described in Section 3.6. This figure shows the entire analyzed time span in two parts: the lower part is the continuation of the upper part. Based on this visualization we can study

- when how much time passed between two versions,
- where phases of different phase descriptions arise,
- when they have high amplitudes (visible in the height of the rectangles),
- when they have high certainty values (visible in a dark color),
- when they have high density values (and thus include many version in a short time),
- when they arise concurrent to phases of other phase descriptions,
- what relationships they have to other phase descriptions,
- etc...

We see that the versions, *i.e.*, the vertical lines representing the single versions, are in the beginning much closer together than towards the end. Especially one phase where there are no versions in a long time span stands out. This time span

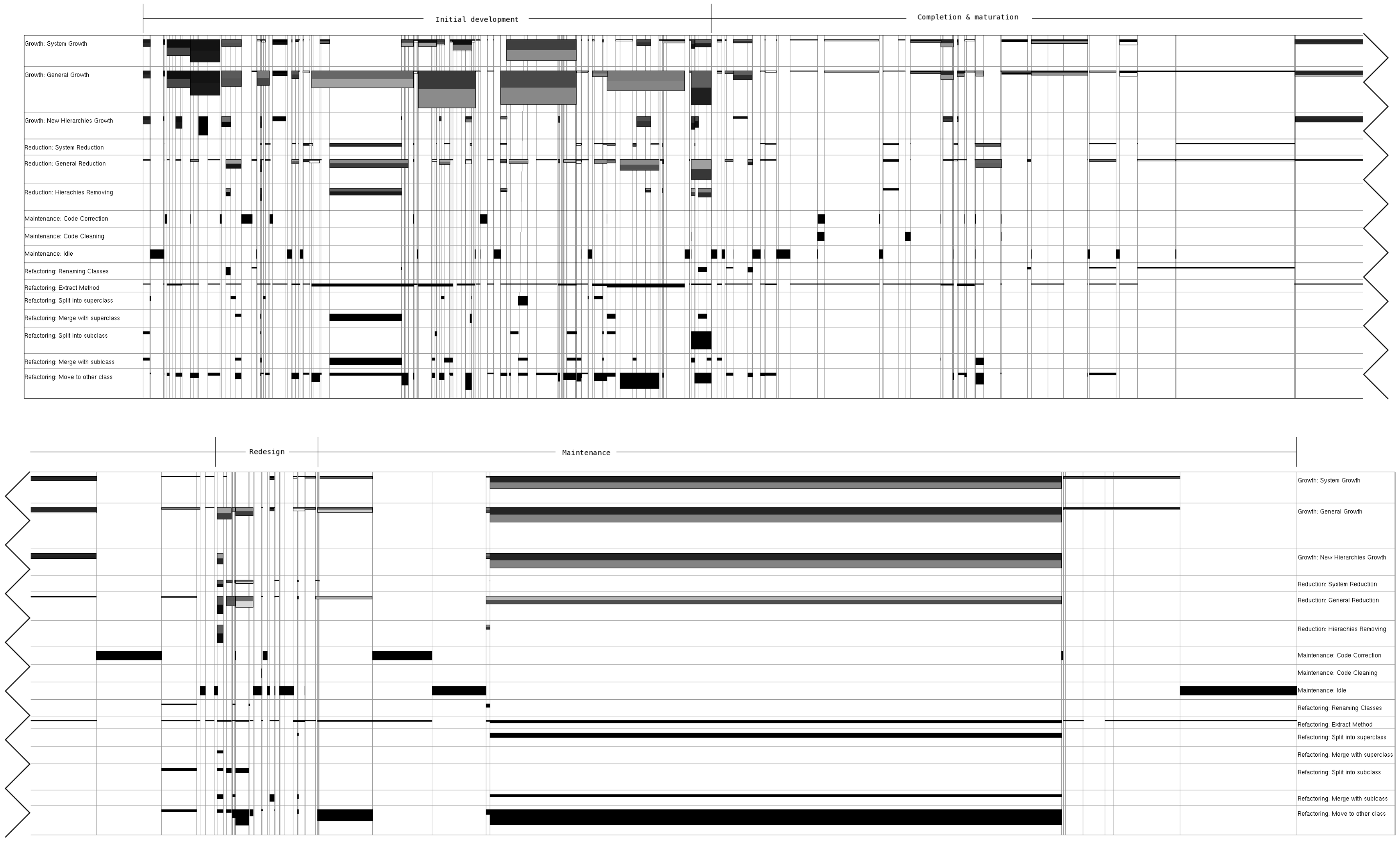


Figure 6.5: The Visualization of the detected phases in SmallWiki. The lower part is the continuation of the upper part

is covered with several phases. It is however most likely that SmallWiki was left unchanged over most of this time span and that the changes have been made only at the end of it.

By looking closer at the visualization, we visually make out four stages: the initial development stage, the completion and maturation stage, a redesign stage, and a maintenance stage. The next sections present how we detected these stages and characterize them.

Initial Development Stage

The upper part of part Figure 6.5 lets us distinguish two distinct stages by the density of the versions which is depicted by the distances between the vertical lines (the distance between two vertical lines is proportional to the time span between the two corresponding versions): In the first stage the versions are closer together than in the latter. Furthermore, there are generally much more phases detected in the first part that in the next part.

Comparing these two stages, we make the following observations:

- **Strong growth solely in the left part -** Both areas are mostly covered by general and system growth phases, but in the left area, the growth phases have a much higher amplitude which indicates a stronger growth. New hierarchies growth phases, which as previously mentioned capture big changes, are almost solely on the left side. Thus, we can conclude that the growth is primarily concentrated in the left area.

- **Refactoring Phases restricted to the left part -** Apart from the phases of the more general refactoring phase descriptions extract method, move to other class, and renaming classes phases, refactoring phases are almost restricted to the left part of the visualization. They thus seem to appear together with growth. Assuming that the detection of refactorings is correct, we conclude that the activity of refactoring has been an essential part of the former stage, but not of the latter.

- **Reduction Phases primarily in the early stage -** Like growth phases, we find reduction phases in both time intervals. However, the occurrence of the general reduction phase and the hierarchies growth phase differs in the two areas: There are much more of them and they have higher amplitudes. This lets us conclude that the removal of code primarily occurred in the left area and thus belongs mostly to growth.

- **Development and fixation of the architecture in the former time span -** The formation of new hierarchies is captured by three different phase descriptions: Primarily by the new hierarchies growth, but also by the split into superclass and the split into subclass phase description. The first one captures the addition of complete new hierarchies, *i.e.,* base classes and leaf classes at the same time, while the others capture the change from a single class into a complete hierarchy. On the left side we find much more and higher new hierarchy growth phases. Phases from the two other phase descriptions are exclusively on the right side. Thus, we can conclude that new hierarchies are (almost) exclusively introduced in the first time span.

Further investigations on the inheritance hierarchies can be detected by studying multiple phases in parallel. For example, the renaming of complete hierarchies would show up as removing and adding complete hierarchies at the same time and thus would show up in a hierarchies removal and a concurrent new hierarchies growth phase. Additionally, there might be a concurrent classes renaming phase if the amount of added classes does not differ much from the amount of the removed classes. We can detect further development activities on hierarchies, but they are restricted to the left area. How we can use our approach recover the evolution of concrete hierarchies is shown later. For now, we conclude that the core hierarchies have been developed and up to a certain degree stabilized in the first stage.

Interpreting this first stage in the evolution of SmallWiki, we assume that in this first stage, a first running version of SmallWiki was developed. This first version might still lack features and have flaws but it already possesses the architecture. We therefore call this stage the initial development stage of SmallWiki. There are different possible ways to verify our assumptions. One possibility would be to uses the detected phases to analyze the exact changes to get a detailed understanding of the applied changes. This technique is demonstrated later in Section 6.2.3 when we show how phases can be used to extract the formation of the architecture from the history. For the verification of our assumptions, we however chose another way: we simply confirmed with the main developer. We found out that there was never an official release, but that users “suddenly” just started to use SmallWiki professionally. First installations have been made starting from the end of the detected initial development phase which is a confirmation for our assumption. Also verified is that the formation and especially the stabilization of the architecture (respectively of the core of SmallWiki) took place in this stage. Two parts however have been completely re-implemented in a later stage.

Summarizing, we can say that in this initial development stage a first versions of SmallWiki was developed that started to be used professionally. In the next stage we thus expect the completion of SmallWiki, *i.e.*, the addition of new features or improvement of existing features, and the improvement of flaws. This stage is described in the next section.

Completion and Maturation Stage

This stage is practically covered with general and system growth phases with low amplitudes. This means that there was continually code added, but never much. There was also continually little amounts of code removed (visible in the "small" general reduction phases). The removal of code has almost exclusively been made while concurrently code has been added since parallel to general reduction phases there are almost always general growth phases. Sometimes the removal was stronger which results in a system reduction phase and sometimes the addition was stronger what shows up in a system growth phase. The addition of code however appears also without a concurrent general reduction phase. It however does not seem to affect the inheritance hierarchies since there are almost no new hierarchies growth, extract hierarchy, split into subclass or split into superclass phases. We conclude that in this stage, existing functionality was improved and completed. We thus call this stage completion and maturation stage.

The described behavior changes again in the lower part of the visualization. Thus, another stage begins which is described in the next section.

Redesign Stage

After the completion stage, we find again a stage where the versions are closer together. Also, in contrast to the completion stage, there are again considerably more and more different refactoring and reduction phases. This stage is mostly covered by general growth, but not by system growth phases. The system is thus in this stage not growing. Instead, the reduction phases are overwhelming. Striking are especially the high general reduction phase and the one version lasting hierarchy removal phase. In parallel to this phase, there are also a new hierarchies growth phase, a system reduction phase and several refactoring phases. By looking at the system reduction phase or the change of the number of classes, we would conclude that the system was reduced. But by considering the detected phases, we see that other changes than removing code have been made. Coming along with the reduction phases, there is one new hierarchy growth and several refactoring phases with partly high amplitudes. This lets us assume that this stage captures the redesign of a part of SmallWiki. This redesign results in a smaller size of the overall system - but the functionality of the system might even have been enhanced. To verify our claim and to clearly understand the changes, we had to further analyze the considered phases. How this could be done is demonstrated in the next Section. We confirmed again with the main developer of SmallWiki and found out that indeed a major part of SmallWiki was redesigned, and another part was improved which confirms our assumptions.

Maintenance stage

It the last stage shown in Figure 6.5, the versions are generally far apart, which means that a lot of time passed between two versions. Outstanding is mainly one time span where there has no version been made for a long time, *i.e.*, for more than 4 months. This time span is covered by many phases with different phase descriptions in parallel which all cover the same versions. Some of these phases, especially the growth phases, have high amplitudes. This lets us conclude that some major changes have been made in this phase. However, most likely in most of this time span, SmallWiki has been left unchanged and the changes have been made only at the end of this stage.

The rest of the maintenance stage is mainly covered by maintenance phases, *i.e.,* by idle and code correction phases. These phases capture either only minimal changes or no changes at all. As the completion and maturation stage, this stage shows compared to the initial development and the redesign stage only a small amount of phases with a high density. Summarizing, we can say that the maintenance stage covers a big amount of the analyzed time span in which only a small amount of versions have been made and in which in only one time span relevant changes have been made.

6.2.3 Recovering the formation of the architecture

Up to now we analyzed the evolution of SmallWiki by interpreting measurements and the visualization of the phases. We compared the shape of the phases and looked for the occurrence of concurrent phases. This gives us a coarse understanding of the evolution. However, phases can also be used as pointers to analyze changes in detail. The applied phase descriptions mostly use detection strategies as high level metrics to detect the occurrence of specific changes. Outgoing from a single phase or certain concurrent phases, we can now apply the underlying detection strategies on the versions the phase encapsulates. Like that, we obtain a set of suspect entities, *e.g.,* classes that we can manually examine. Since the used detection strategies mainly detect entities based on the way they changed, this technique helps us recover the changes in the system. Note that with this technique, we first step from the entire evolution (more than 300 versions) to single versions and from there to single design changes. To do this in an efficient manner, we need the support by a software tool. The tool we implemented and used is described in Appendix A.

To demonstrate how phases can be used to gain an understanding of discrete changes, we focus on the evolution of the architecture, which is mainly captured in inheritance hierarchies. Thus, we focus on phases that indicate changes in superclass and possibly also leaf classes, *i.e.,* we mostly focus on new hierarchy growth, hierarchy removal and extract hierarchy phases. Additionally, we take the split into superclass and split into subclass phases into consideration. In the following, we present a number of different shifts from one version to the next and describe then how the architecture was changed and how we obtained the changes. The analyzed version shifts are all part of the initial development stage which is depicted in detail in Figure 6.6.

- **Version 1.5 - 1.6.** There is a small new hierarchies growth phase which indicates that the amount of added hierarchies is low. Furthermore, this phase has a certainty value of 1.0 and a classification number of 0.2 which says that the change to version 1.6 consisted of the addition of a few added methods,

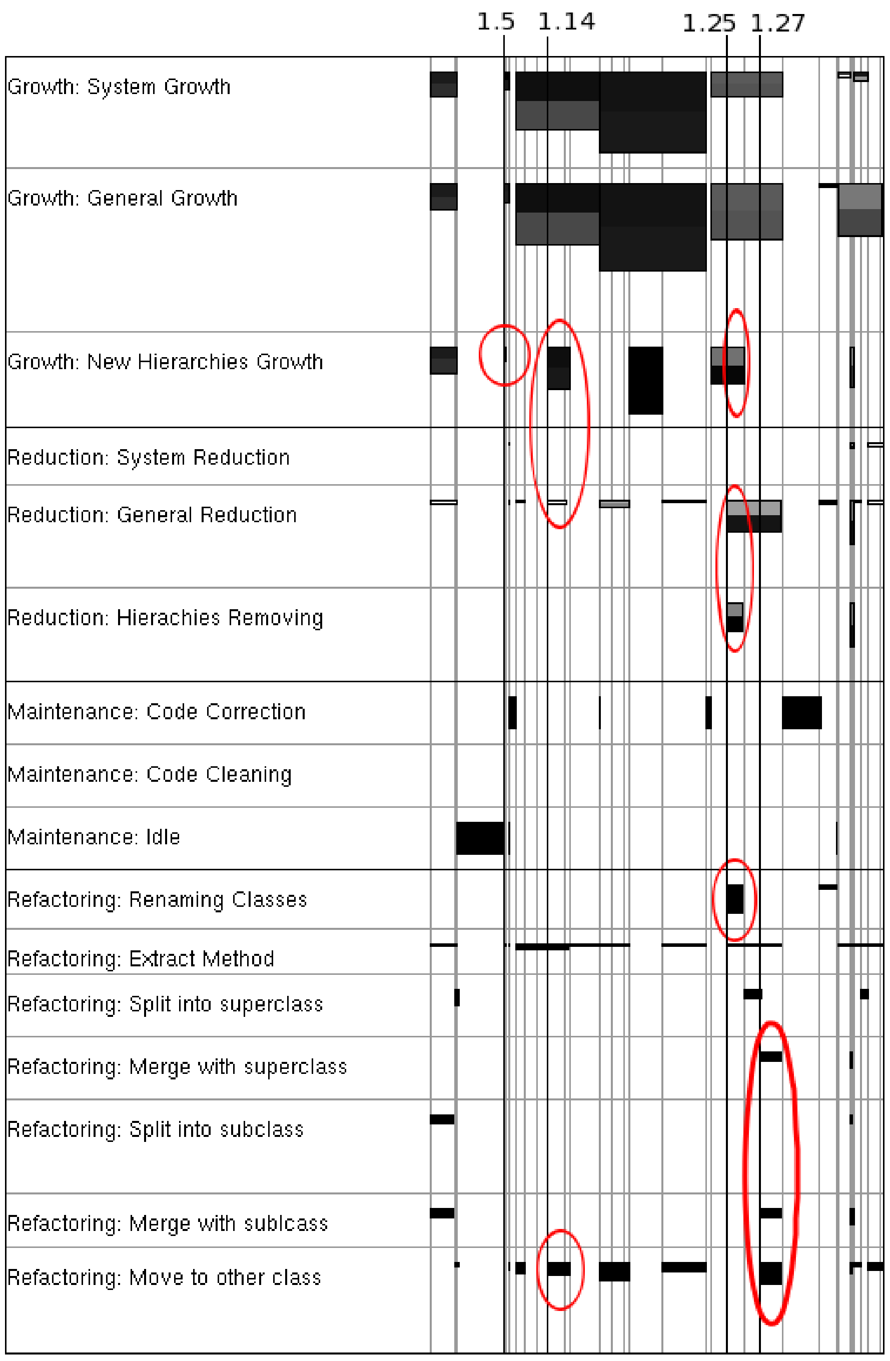


Figure 6.6: The Initial Development Stage of SmallWiki in Detail

no removal of classes or methods at all but mostly of the addition of new classes. To know which hierarchy was introduced, we apply the detection strategies *NewSuperclass* and *NewLeafClass* on the version 1.6. We see that the abstract class *Structure* and its leaf classes *Book*, *Page* and *Shelf* were added.

- **Version 1.14 - 1.15.** Between these versions, there is again a new hierarchies growth phase which this time has a higher amplitude. Concurrently, there is a general reduction and a move to other class phase. We thus expect that functionality has been moved from one or more classes into an newly created inheritance hierarchy. Applying the detection strategies *NewSuperclass* and *NewLeafClass* we find out that the base class *Action* and its direct and indirect subclasses are newly implemented (totally 12 classes). Three of the direct subclasses are named accordingly to the subclasses of the class *Structure*: the class *BookAction, PageAction* and *ShelfAction*. Applying the detection strategy *MoveToOtherClassSource* or *ReducedClass* on the version 2.15 we get out that functionality has been moved away from the subclasses of the class *Structure* (*Book*, *Page*, and *Shelf*). Where the functionality was moved to is harder to determine by applying detection strategies. However, comparing the names of the removed methods with the method names of the added classes, we find out that the classes of the hierarchy *Action* represent operations that can be applied on (sub-) classes of the *Structure* hierarchy.

- **Version 1.25 - 1.26.** We are again looking at a new hierarchy growth phase. This time there are hierarchies removal, classes renaming and a general reduction phase in parallel. We thus presume that one or more complete hierarchies were renamed - what in our model shows up as removing and addition of the same amount of classes. The concurrent classes renaming phase backs our presumption up. By applying the detection strategies *RemovedSuperclass, NewSuperclass, RemovedLeafClass* and *NewLeafClass* we confirm our expectation: The classes *PageVisitor* and all its subclasses were renamed into *WikiVisitor* (accordingly the subclasses).

  The names of those classes give us an idea about their responsibility; they seem to conform to the *Visitor* design pattern [GHVJ93] and thus represent an operation to be performed on the elements of an object structure. Their change of name lets us assume that their responsibility changed from traversing an object structure that consists of instances of the class *Page* into traversing *WikiItems* (which seem to be more abstract than *Pages*). Applying the *NewSuperclass* detection strategy on version 1.26 we find out that new superclass *WikiItem* has been introduced, which is an abstraction of the existing hierarchies *PageComponent*, *PageLeaf* and *Structure*. Additionally, this complete hierarchy was enhanced with multiple leaf classes. This verifies our previous assumption.

- **Version 1.27 - 1.28.** In the version shift from version 1.27 to 1.28, the system has been reduced since this version shift is encapsulated in a system reduction phase. Concurrently, there are a merge with subclass and a merge with superclass phase. The merge with superclass phase captures the move of functionality from a removed superclass into one of its subclasss while the merge with subclass captures the oppositional change, *i.e.,* the move of functionality from a removed subclass into its superclass. Applying the detection strategy *MergeWithSubclass* on version 1.29, we find out that the class *Preformatted* is no longer a subclass of the class *Text* and that the class *Text* has grown. With the detection strategy *MergeWithSuperclass*, we discover that the merge with superclass refactoring captures the same change from another direction: it reveals that the class *Preformatted* has changed from being a subclass of the class *Text* into a subclass of the class *PageComposit*. Since the class *Preformatted* was detected by the detection strategy *MergeWithSuperclass*, it also has has been extended.

We demonstrated how the detection of phases can be used as a tool to get a detailed understanding. Generally, we studied phases that appear concurrently and made an assumption about the change based on the semantics of the underlying phase description. Our assumption then had to be verified by applying single detection strategies on single versions. Applied in this way, the detection of phases is an aid to filter out and understand relevant changes. What changes are relevant is dependent on the addressed problem.

6.3 SmallWiki Coarse Grained

In this section, we again analyze the same time span of the evolution of SmallWiki. But this time, we only consider every fifth versions. More precisely, we analyze every version of which the number is divisible through 5 plus the first and the last one, *i.e.,* version 2.1, 2.5, 2.10, 2.15, 2.20, ... , 1.305, 1.310, and 1.313. In this case study, we forgo an analysis of the computed measurements. Instead we only present the visualization of the detected phases and compare it to the visualization of the fine grained case study. We however present two different ways of detecting the phases. In Chapter 3 we mentioned that there are different ways of forming phases out of the versions a phase description detected. Up to now, we always formed the maximal phases, *i.e.,* phases that encapsulate the maximal amount of consecutive versions. On the other extreme, we can form phases that all have the length of one. Thus, if a phase description detected two consecutive versions, we build two phases. In this section, we present both mentioned ways, first the maximal phases and then the minimal.

Visualizing Maximal Phases

Figure Figure 6.7 shows the coarse grained visualization of the history of SmallWiki. Based on that, we can make the following observations:

One Instead of 65 Extract Method Phase. The visualization shows one single extract method phase that encapsulates the entire history. In the previous case study, we found 65 single extract method phases and accordingly time spans in-between that were not covered with extract method phases. The most probable reason for this is that the extract method refactoring has been applied at least once in five versions and thus is detected in every considered version in this case study. In the visualization of the fine grained case study, we can indeed not discover five consecutive versions where there is no extract method phase. This does however not necessarily imply that those refactorings are also detected in this coarse grained case study: It could for example be that the refactoring has been applied on a class from version 1 to version 2 and that from version 3 to version 4, another change has been made to the same class which 'hides' the refactoring. Considering only the version 1 and 5, the application of the refactoring can then not be detected. We can thus assume that the extract method refactoring has generally not been 'hidden' by other changes, that is, the detected classes where the extract method refactoring has been applied are up to a certain degree stable. But we can not exclude is that the phase description is not precise enough and that misclassifications occurred. The extract method refactoring however only detects a version as part of a phase if there is a class that at the same time has more methods than before and some methods have been reduced. False positives are restricted to the situation where both changes occur but are unrelated. Therefore, we conclude that this reason is most likely not the cause for the notably different detection in the two case studies.

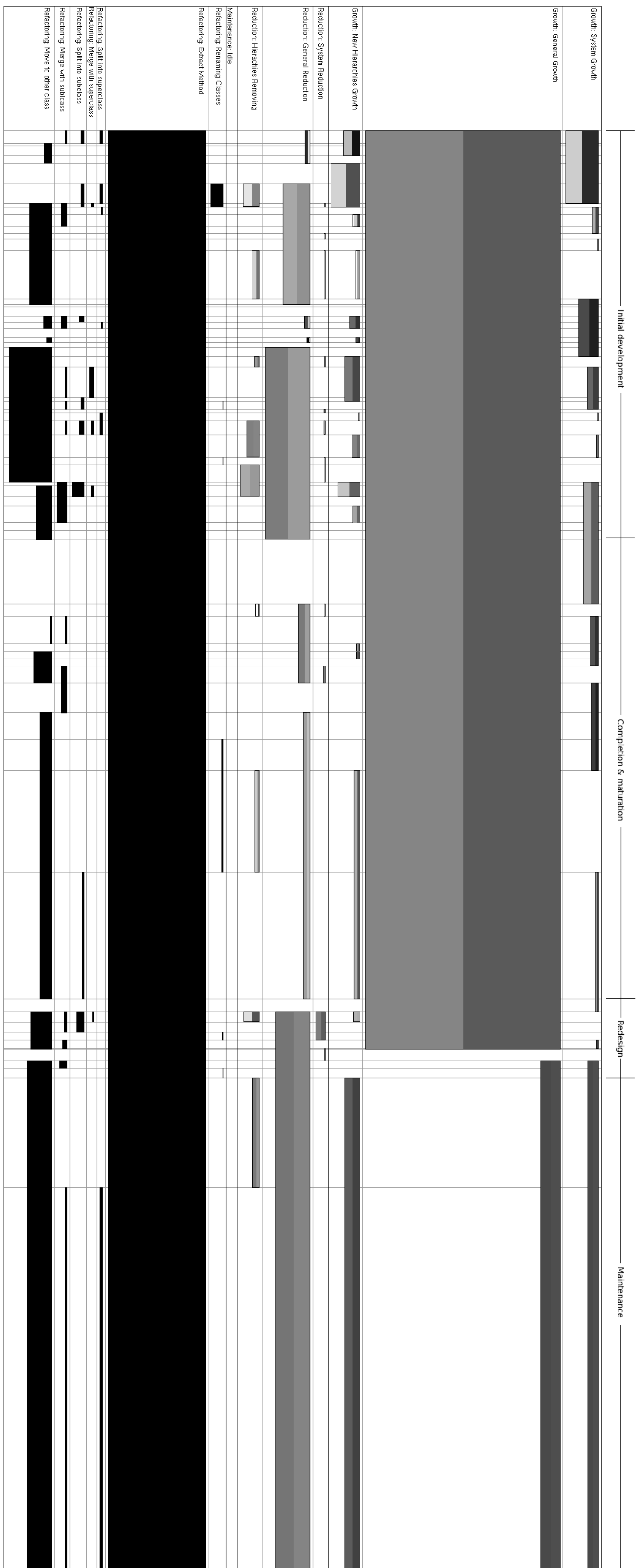


Figure 6.7: SmallWiki coarse grained

Similar differences between the fine grained and the coarse grained case study arise in other phase descriptions, for example in the general growth, the move to other class and the general reduction phase descriptions. The cause for the difference is the same as in the extract method refactoring although the phase descriptions are defined differently: If there are less than five consecutive versions that are not covered with a phase in the fine grained case study, they are covered by a phase with one of the mentioned phase descriptions in the coarse grained. The maintenance phase descriptions however show a different change, as shown in the next paragraph:

No Maintenance Phases Discovered. Except for one single 'idle' phase which is so short that it is not even visible in the visualization, there are no maintenance phases detected. In the fine grained case study in contrast, we found 40 idle, 19 code correction and 5 code cleaning phases. Maintenance phases thus in contrast to growth phases have much lower coverage values compared to the previous case study. For that, we found mainly two reasons:

- The first reason lies in the way the phase descriptions are defined: The maintenance phase descriptions in contrast to most other phase descriptions exclude most kind of changes. For example, they exclude the addition of new classes. At least one of the excluded changes has been applied at least once in five versions. If we analyze only every fifth version, the occurrence of this change breaks the detection of the maintenance phase.
- The changes that the maintenance phase descriptions aim at detecting are phases where only small changes in specific classes have been applied. If in an interval of 5 versions where 'maintenance changes' have been applied to a specific class, also other changes have been made on the same class, the maintenance changes could not be detected anymore.

Evolutionary stages are hardly visible. In the previous case study, we could discern several different stages in the analyze history, *i.e.,* the initial development, the completion and maturation stage, a redesign stage, and a maintenance stage. Especially the initial development stage differed clearly from its successor.

One of the main differences that made us distinguish the stages was the density of the versions, *i.e.,* how much time passed between the versions. This difference is still visible, but it is not so clear anymore. Other main differences were the occurrence of the refactoring and the reduction phases and the amplitudes of the growth phases. The amplitudes of the growth phases hardly reveals the stages since we found very long phases which do not show differences of the amount of added code in the encapsulated versions. The initial development and the succeeding maturation stage both are covered with one single general growth phase which stands out with a very high amplitude. This phase however does not show any difference between the two stages. It just shows that code has mainly been added in the first half of the analyzed time span. The occurrence of the refactoring and the reduction phases show the biggest differences between the detected stages: As in the fine grained case study, they are mainly detected in the initial development stage and in the redesign stage.

Summarizing, we can say the density of the versions and the occurrence of refactoring and reduction still shows differences between the stages, but describing them is hard in the coarse grained case study.

Conclusion

We found that the detected phases offer less information if we only consider every fifth version. The phases of most applied phase descriptions are long and provide information about the entire encapsulated time span. Maintenance phases however disappeared entirely. Furthermore, even the detection and description of the detected stages got difficult with the reduction to every fifth version.

To get more information out of the considered versions, we can however build minimal instead of maximal phases, *i.e.*, make phases that all encapsulate only one single version. Like that we see for example a different amplitude in every single version.

6.3.1 Visualizing Minimal Phases

In this section, we compare the visualization of minimal and maximal phases. We restrict ourselves to a part of the entire history of SmallWiki, *i.e.*, on that part that was covered by the first general growth phase in the previous case study (version 1.1 to 1.285). Furthermore we restrict ourselves to the growth and the reduction phase descriptions. Figure 6.8 shows the visualization of the maximal and the minimal growth and reduction phase descriptions. We now compare both visualizations and study the effect of the way phases are built.

We see that the big general growth phase in the upper figure is split up into many single general growth phases. The sum of the amplitudes of all of these phases equals to amplitude of the general growth phase in the upper figure. In contrast to the upper visualization, we discover time spans with a bigger amount of added code and time spans where hardly any code has been added. We see more detailed information not only in the amplitude measurement but also in the certainty and the classification number measurements. That is, we can discover phases with higher or lower certainty values or classification numbers.

The differences between maximal and minimal phases can exemplary be seen in the general reduction phases between version 1.260 and 1.265. In the upper visualization, we see one single phase that shows information about the entire encapsulated time span. It has one amplitude which shows the total amount of removed code, one certainty value which compares the amount of added and removed code over the total length of the phase and one classification number which shows an average property of the total time span. In the lower visualization we see 4 con-

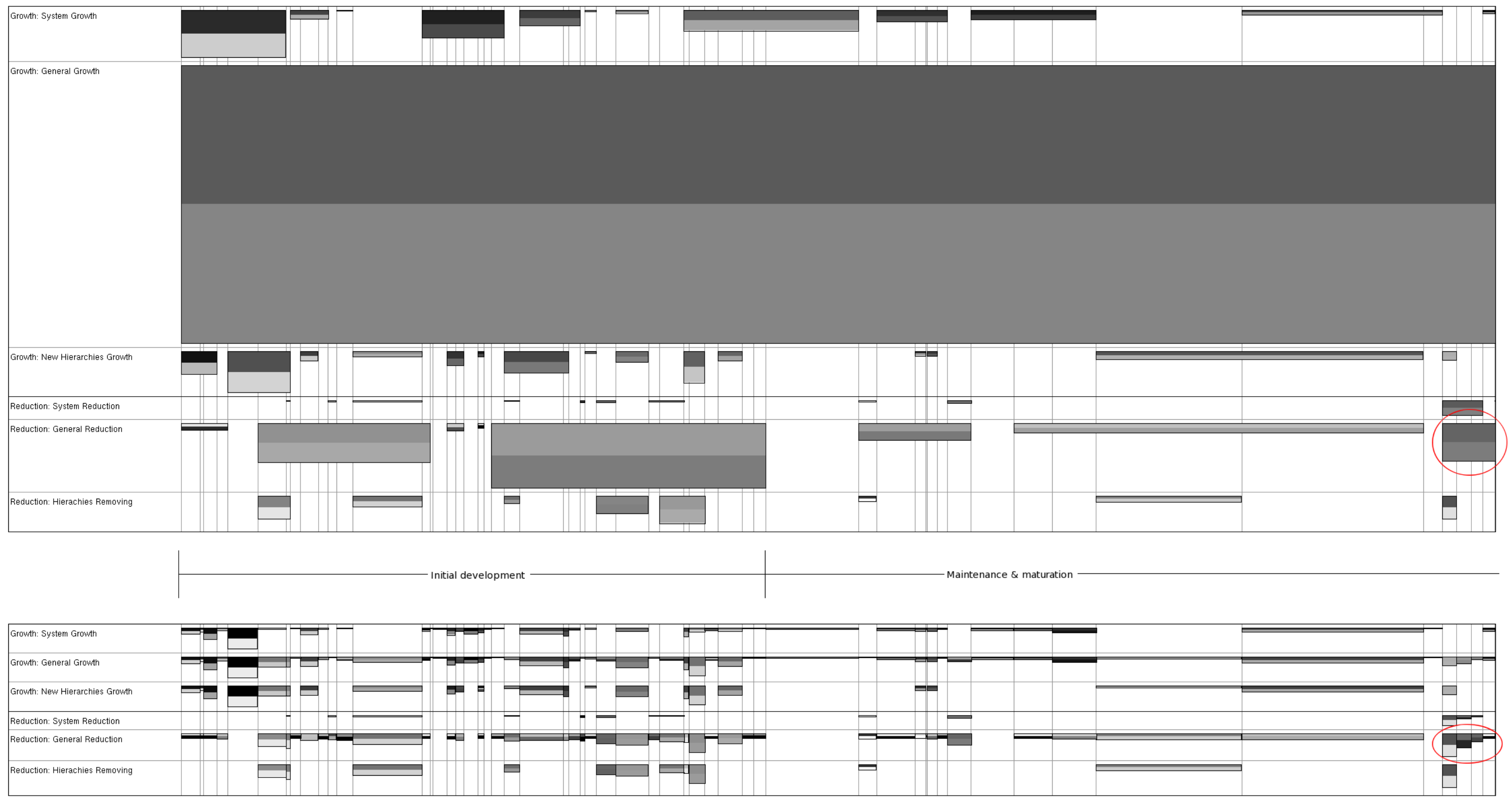


Figure 6.8: SmallWiki coarse grained building maximal (on top) and minimal phases (below).

secutive phases, each having a different amplitude, different certainty value and a different classification number. This indicates that each phases encapsulates a different shape of change. For example, we discover that most functionality has been removed in the first version shift and that each of theses phases has a lower amplitude than its predecessor phase. We find the same in the concurrent general growth phases. The amplitude of the concurrent general growth phase is however in four phases smaller so that a system reduction phases is detected. In the last version shift however, the amplitude of the general growth phases is bigger so that a system growth phase is detected.

By looking at all growth phases we can discover the end of the initial development stage in the evolution. In the growth phases, especially the system growth and the new hierarchies growth phases, we see that most of the growth has been made in the initial development phase. Also the reduction phases which we primarily found in the initial development stage, have higher amplitudes than in the succeeding state.

Conclusion

In the previous case study, we found that the maximal phases together with the consideration of only every fifth version resulted in a not detailed enough information content. By varying the construction of phases from version a phase description detected, we gained a more detailed visualization of the evolution of SmallWiki. Summarizing, we can say that the building of minimal phases seemed to be more appropriate considering only every fifth version. Taking every single version into consideration, we however found it more appropriate to build maximal phases since the summarizing of versions lets us regard the evolution on a higher level.

Chapter 7

Detecting Phases in Class Histories

In this chapter, we demonstrate the application of phase descriptions on the class level, *i.e.,* we detect phases in histories of classes. For that, we apply the catalog of phases descriptions we defined in Chapter 5 on three classes of Jun and on two classes of SmallWiki. The case studies Jun and SmallWiki are introduced in the previous chapter. Generally, we do not present the complete history of the classes but furthermore selected a part of the history. We chose the classes and the parts of their histories so that we ended up with five evolutions that have different characteristics.

7.1 The Evolution of the Class JunTopologicalElement

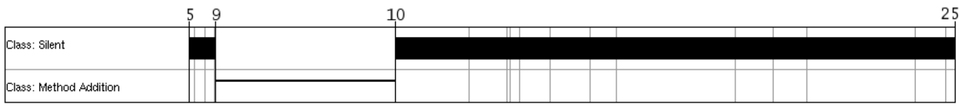


Figure 7.1: The Evolution of the class JunTopologicalElement

The evolution from version 5 to version 25 of the class JunTopologicalElement is depicted in Figure 7.1. The entire evolution is covered by phases which means that the class has been part of every single version. Except for one version shift, the history is covered by *idle* phases. This indicates that the class *JunTopologicalElement* has been changed once, from version 9 to 10. This change is captured by the *method addition* phase. By interacting with the this phase, we discover that the amplitude of this class is 1 which means in this phase, one single method has been added. But there have however no messages been added to this class since there is no concurrent *statements addition* phase. This lets us assume that some code has been extracted from one method into a newly created one and that no functionality has been added to this class. By reading the source code of the versions 9 and 10 we however discover that the code in the added method has been commented and thus does not contain any statements. We therefore conclude that this class has been extended with an “empty” method in version 10 and remained unchanged in the rest of the analyzed time span.

7.2 The Evolution of the Class JunOpenGL3dPolylineLoop

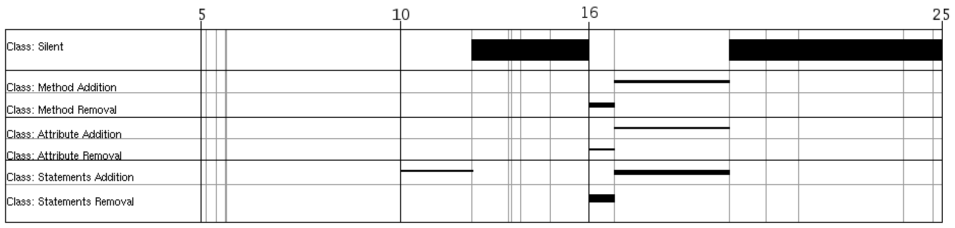


Figure 7.2: The Evolution of the class JunOpenGL3dPolylineLoop

Figure 7.2 depicts the evolution of the class JunOpenGL3dPolylineLoop from version 5 to 25. The class was introduced in version 10 since the time span between version 5 an ten is not covered by any phases. In the following version shift, one or more methods have been extended but no new ones have been added since there is a *statements addition* phase but no *method addition* phase. This *statements addition* phase has an amplitude of 1 which means that one single statement has been added to one method of the class. In this version shift, no methods or attributes have been added since there is no *method addition* or *attribute addition* phase. The following silent phase indicates that the class has not been changed from version 11 to 16. From version 16 to 17, there is a *method removal* phase with an amplitude of 4, an *attribute removal* phase with an amplitude of 1, and a *statements removal* phase in parallel. This indicates that 4 methods and one attribute have been removed. In the following version shift, the class grew again since there is a *method addition*, a *attribute addition* and a *statement addition* phase in parallel. The amplitudes of those phases reveal that in total, one attribute and two methods have been added. After this version shift, the class has not been changed anymore.

7.3 The Evolution of the Short-Living Class JunBrowserEnhance

Figure 7.3 depicts the class JunBrowserEnhance which was introduced in version 49, remained unchanged til version 50 and then was removed again in version 51.

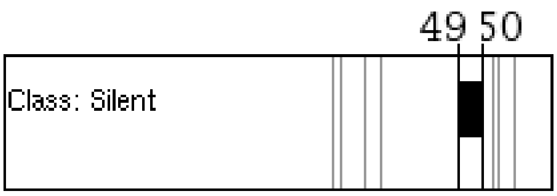


Figure 7.3: The Evolution of the class JunBrowserEnhance

7.4 The Evolution of the Unstable Class Structure

Figure 7.4 shows SmallWiki's class *Structure* from version 1.66 to 1.174. Most of the depicted time span is covered by silent phases. But there are many addition and removal phases distributed on the entire visualization. This indicates, that this class is constantly enlarged and belittled in the depicted time span. For example, we find several attribute addition and attribute removal phases. This indicates that the amount of attributes in this class is not stable. The same is true for the number of methods and the number of statements. Thus, this class undergoes constant changes which means that it is unstable. Note that we are considering more than 100 versions of this class, *i.e.*, this class is unstable in more than 100 versions.

Generally, there are more of the phases that capture the addition of code than of those phases that capture the removal. The addition phases also have higher amplitudes which lets us conclude that the class *Structure* was in average growing in the depicted time span.

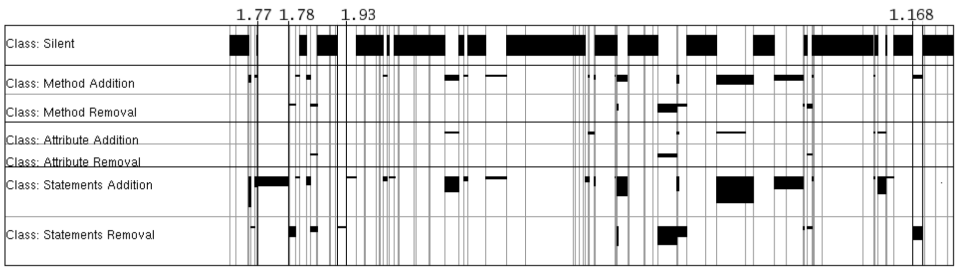


Figure 7.4: The Evolution of the class Structure

In the time span from version 1.77 to 1.78, there is only a statement addition phase and no other concurrent phase. This means that no methods or attributes have been added or removed, but that existing methods have been extended. This could be a sign for the correction of bugs: The additional statements in methods fulfill missing requirements in the methods they have been added to.

An oppositional change is depicted in the statements removal phase from version 1.92 to version 1.93. It also has no other concurrent phases. Thus, no methods or attributes have been added or removed but statements have been deleted in existing methods. The phase however only considers the total amount of statements, *i.e.*, the total amount of statements has been reduced. This could be a sign that some methods have been implemented in a more elegant way. Since the statements removal phase description only considers the overall amount of statements of a class, it could however be that some methods have been extended while others have been reduced.

In the time span from version 1.168 to 1.169, there are a method addition and a concurrent statement removal phase. This indicates that the total amount of methods grew, but the total amount of statements in this class was reduced. This could be a sign that common code in several methods has been factored out into new methods so that the total amount of statements has been reduced. This change refers to the extract method refactoring [FBB$^+$99].

7.5 Class VisitorRendererHtml

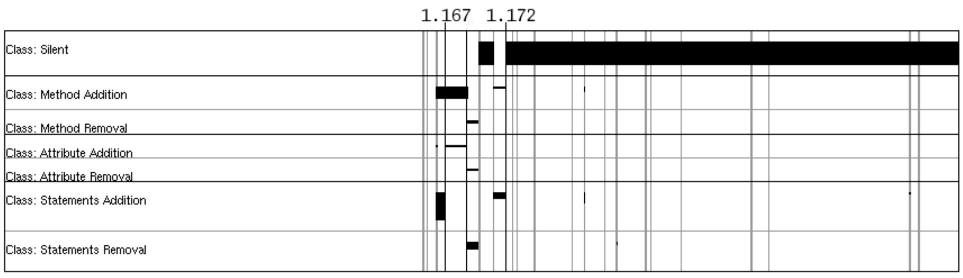


Figure 7.5: The Evolution of the class VisitorRendererHtml

Figure 7.5 depicts the introduction of SmallWiki's class *VisitorRendererHtml* from version 1.160 to 1.200. In the first depicted versions, the class did no exists since there are no phases detected. In the beginning, this class shows an unstable behavior like the class *Structure*. But then, apart from version 1.172 on, this class is stable. Thus, the class *WikiVisitorRenderer* has been introduced, then changed in five version shifts until it reached its final state.

The shift from version 1.167 to version 1.168 shows a method addition and an attribute addition phase in parallel. In this time span, there is however no concurrent statement addition phase. Thus, methods have been added but the total amount of statements remained the same.

7.6 Conclusion

In this chapter we analyzed the evolution of five different classes with 7 phase descriptions. The phase descriptions itself were, except the idle phase description based on a single metrics only. They indicated us when and to what extent this metric changed in a class. For example a method addition phase with an amplitude of seven indicates that seven methods have been added in the encapsulated time span. The concurrent occurance of the detected phases lets us understand the changes on a more exact level and lets us make assumptions about the reasons of those changes. For example, a method addition and a concurrent statements removal phases let us assume that the extract method refactoring had been applied to the analyzed class. In our analysis, we analyzed the concurrent occurance of phases visually. For that, we could however also use composition operators to combine two or more phase descriptions.

Chapter 8

Conclusion

8.1 Summary

In this work, we presented an approach to understand the evolution of software by detecting phases. We introduced our approach step by step. First we introduced its prerequisites starting with FAMIX which is a meta-model for a language independent representation of object-oriented source code at a single point in time. Then we presented the HISMO history meta-model which provides means to represent a set of FAMIX models of different points in time as a history, and thus enables analyzing evolution. Then we introduced software metrics which we divided into structural and evolutionary software metrics. Based on software metrics, we presented detection strategies which are quantifiable expressions to detect specific design fragments. After introducing these prerequisites, we presented our approach of detecting phases with phase descriptions. A phase is a set of consecutive versions that all comply with a phase description which is an expression to detect phases. We introduced measurements on phases, namely the length, duration and the density, and measurements on phase descriptions, *i.e.*, on the set of phases a phase description detects in a history. We also provided attributes of phase descriptions. These are measurements that have a common meaning across all phase descriptions but the computation is defined based on the specific phase description. Then, we presented a way to visualize phases and like that study the concurrent occurrence of phases with different phase descriptions.

We then applied the approach of detecting phases on the system and on the class level. For that, we first presented two catalogs of phase descriptions, one that is applicable on the system level, and a simpler one applicable on the class level. We presented the visualization of a part of the evolution of the system Jun and then analyzed the entire evolution of SmallWiki. We showed how the defined measurements can be used to gain information about the entire time span. Then, we discovered several stages in SmallWiki's evolution by reading the visualization of the detected phases. And finally, we demonstrated how phases can be used as a tool to get a detailed understanding of the changes by using phases to recover the formation of SmallWiki's architecture. Therefore we inspected the phases with phase descriptions that indicate changes in the inheritance hierarchies. Then, we applied our approach also on the class level, that is, we used the second catalog of phase descriptions to discover phases in the history of five different classes from Jun and SmallWiki.

8.2 Future Work

- **Enhancement of the catalogs of phase descriptions** - In this work, we presented a set of phase descriptions that proved to be useful for our works. Most of these phase descriptions are however simple and are based on heuristics or assumptions. For example, system growth and the system reduction phase descriptions are based on one single formula that defines how the size of a system changes. Also the refactoring phase descriptions use simple detection heuristics, *e.g.,* the renaming classes phase description that only considers the amount of removed and added classes but disregards if those classes are identical. By improving the phase descriptions, we could get a higher degree of correct detections and also get a better understanding of the analyzed evolution. Improving the phase descriptions includes improving the definition of the certainty and the amplitude of the phase descriptions. Furthermore, we could define a classification number on more phase descriptions and also define it as a multi-dimensional vector instead of a single number.

- **Use phases to indicate the introduction or improvement of flaws** - Together with detection strategies, phase descriptions could be used to detect certain design flaws and with that also time spans in the evolution where flaws have been introduced. Furthermore, with phases and the presented visualization, we could detect the introduction of design flaws and visualize with phases how they evolved. For example, we could detect the introduction of a god class and discover in the visualization if the flaw has been corrected, if the corresponding structure has been unstable, or if it stays stable to the current version.

- **Apply phase descriptions on more levels** - In this work, we presented two catalogs of phase descriptions, one applicable on the system level and one on the class level. Phases could also be used on other levels, for example to study the history of subsystems, single methods, attributes, etc. For that, we had to define new catalogs of phase descriptions on the appropriate level. For example, phases could be used to concurrently study the evolution of two subsystems of a system. Like that, we might discover subsystems that evolve similarly or in an oppositional way.

- **Relate to quality related issues** - In this work, we always focused on using phases to understand the evolution of software. We are however convinced that phases can support the assessment of quality of a system using its history. For example, we might detect phases that indicate that each functional enhancement of a system caused major redesigns, which would be a bad sign for the maintainability of a system.

- **Use phases to find patterns in software evolution -** Comparing the evolution of different systems, we could detect patters of how software evolves. Such patterns could be found on various levels, for example on comparing the measurements based on phase descriptions (as presented in the SmallWiki case study), or by comparing the visualization. For example, we might discover that in some software, first single classes have been introduced that later on were subclasses and grew to core hierarchies. Also we might relate detected patters to the work flow of the developers: A software that has been developed with extreme programming most likely shows a different evolution than a software that has been developed with a strict waterfall model [Som00]. That is, in the extreme programming project, we might discover patterns that are not present in the later project and vice versa.

Appendix A

Tool Support

In this chapter, we describe the software *Quala* which we developed and found useful in the context of detecting phases. We focus on describing the functionality of it and forgo a description of the implementation.

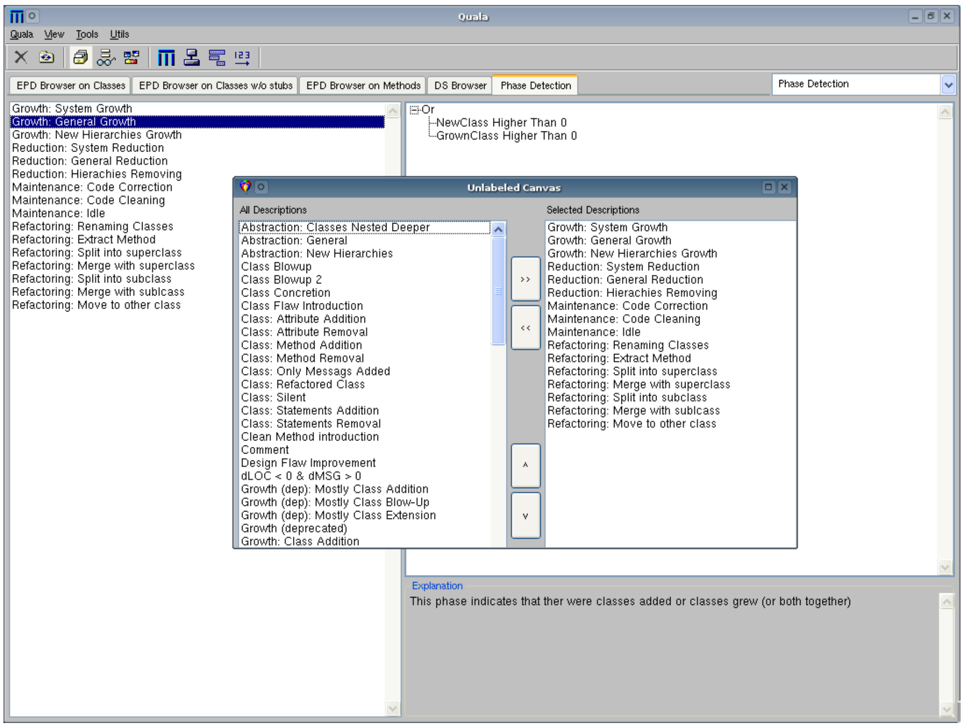
Quala includes the mechanism of the detection of phases with phase descriptions and a set of tools to inspect detected phases. It provides a main window (depicted in Figure A.1) that groups the single tools of *Quala* in tabs. *Quala* is based on *Moose* [DLT00] and Van [1].

To work with phases, the user has to first import a set of FAMIX models, then create a history out of a selection of the imported models, choose a set of active phase descriptions and then detect phases with these active phase descriptions. The importing of the FAMIX models and the creation of a history from a selection of the imported models are implemented in *Moose* respectively in *Van*. The active phase descriptions are shown in the "Phase Detection" tab of the main window of *Quala*. In this tab, the user can select a phase description, view its detection expression and detect phases with it. The phases a phase description detected are cached. They thus do not have to be re-detected if the user chooses to inspect them a second time. This caching is important since detecting phases in a large history might take several hours. The cached phases can however be deleted by choosing `Utils >> Remove Cached Phases` from the menu bar.

To visualize phases, the user can simply click on or which visualize the phases either with identical distance between the single version respectively with distances proportional to the time differences between the versions. *Quala* then detects phases with all active phase descriptions if they are not cached and opens the visualization. The visualization is implemented in *CodeCrawler* which is a language-independent software visualization tool [Lan00, LD04]. A screenshot of CodeCrawler visualizing phases is shown in Figure A.2.

The user can choose the phase descriptions to apply in the appropriate dialog which is shown in Figure A.1. This dialog shows in the left list all defined phase descriptions. The user can select phase descriptions in this list and put them in the

[1]see http://www.iam.unibe.ch/ scg/Research/Van/index.html for more information



```
Growth: System Growth
Growth: General Growth
Growth: New Hierarchies Growth
Reduction: System Reduction
Reduction: General Reduction
Reduction: Hierachies Removing
Maintenance: Code Correction
Maintenance: Code Cleaning
Maintenance: Idle
Refactoring: Renaming Classes
Refactoring: Extract Method
Refactoring: Split into superclass
Refactoring: Merge with superclass
Refactoring: Split into subclass
Refactoring: Merge with sublcass
Refactoring: Move to other class
```

```
Or
 ├NewClass Higher Than 0
 └GrownClass Higher Than 0
```

All Descriptions:

```
Abstraction: Classes Nested Deeper
Abstraction: General
Abstraction: New Hierarchies
Class Blowup
Class Blowup 2
Class Concretion
Class Flaw Introduction
Class: Attribute Addition
Class: Attribute Removal
Class: Method Addition
Class: Method Removal
Class: Only Messags Added
Class: Refactored Class
Class: Silent
Class: Statements Addition
Class: Statements Removal
Clean Method introduction
Comment
Design Flaw Improvement
dLOC < 0 & dMSG > 0
Growth (dep): Mostly Class Addition
Growth (dep): Mostly Class Blow-Up
Growth (dep): Mostly Class Extension
Growth (deprecated)
Growth: Class Addition
```

Selected Descriptions:

```
Growth: System Growth
Growth: General Growth
Growth: New Hierarchies Growth
Reduction: System Reduction
Reduction: General Reduction
Reduction: Hierachies Removing
Maintenance: Code Correction
Maintenance: Code Cleaning
Maintenance: Idle
Refactoring: Renaming Classes
Refactoring: Extract Method
Refactoring: Split into superclass
Refactoring: Merge with superclass
Refactoring: Split into subclass
Refactoring: Merge with sublcass
Refactoring: Move to other class
```

Explanation: This phase indicates that ther were classes added or classes grew (or both together)

Figure A.1: The main window of *Quala* with the opened phase description selection dialog.

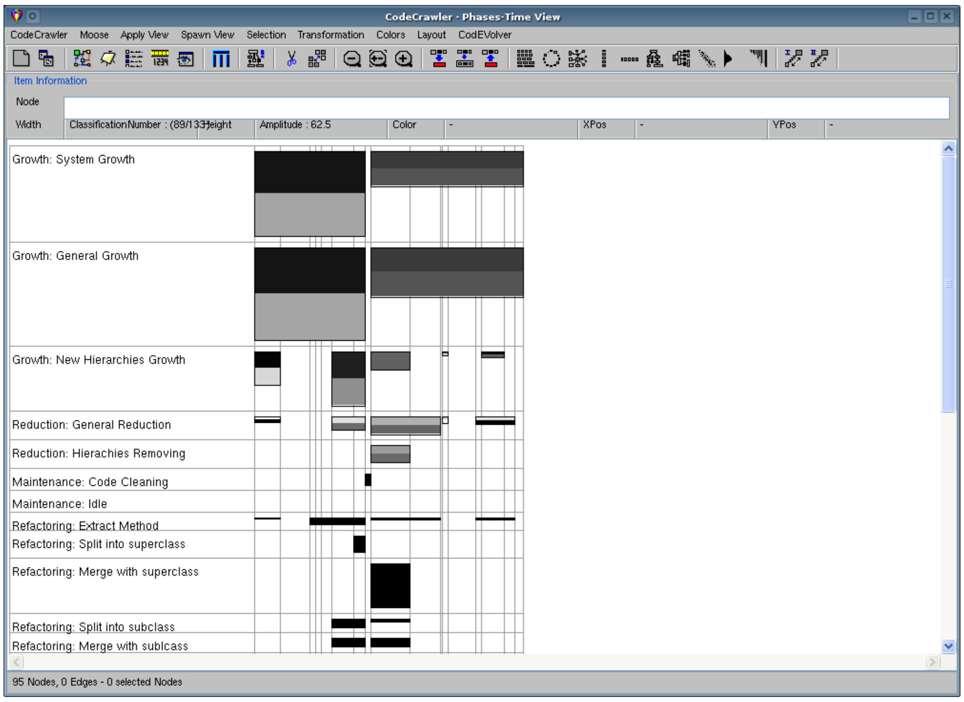

CodeCrawler - Phases-Time View
CodeCrawler Moose Apply View Spawn View Selection Transformation Colors Layout CodEVolver
Item Information
Node
Width ClassificationNumber : (89/133) Height Amplitude : 62.5 Color - XPos - YPos -
Growth: System Growth
Growth: General Growth
Growth: New Hierarchies Growth
Reduction: General Reduction
Reduction: Hierachies Removing
Maintenance: Code Cleaning
Maintenance: Idle
Refactoring: Extract Method
Refactoring: Split into superclass
Refactoring: Merge with superclass
Refactoring: Split into subclass
Refactoring: Merge with sublcass
95 Nodes, 0 Edges - 0 selected Nodes

Figure A.2: CodeCrawler Visualizing Phases

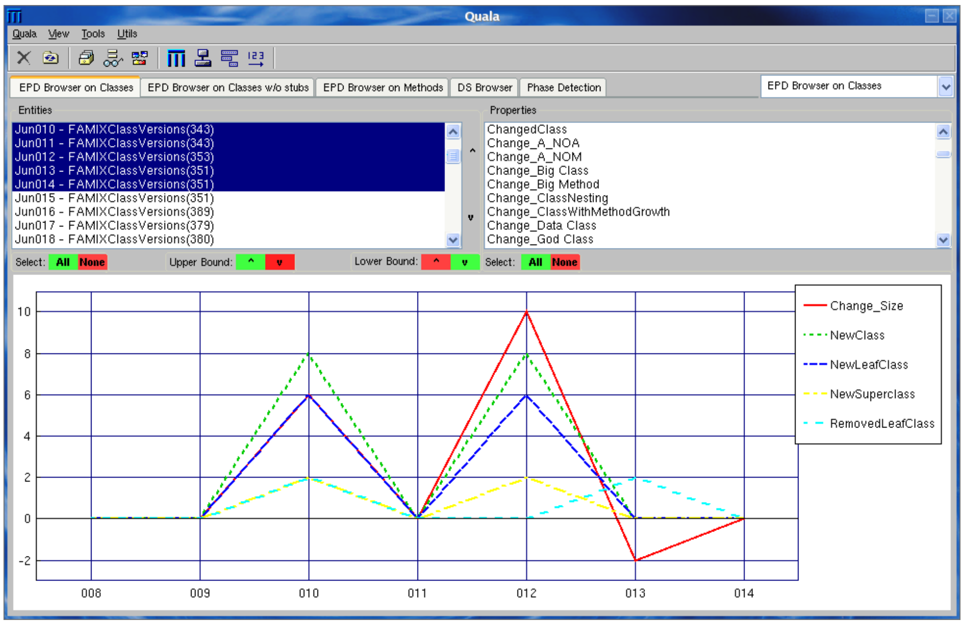
right list which contains those phase descriptions that are currently active. Furthermore, the user can change the order of the active phase descriptions. This order is relevant for the visualizations and for the “Phase Description” tab of the main window. This dialog also offers the possibility to combine phase descriptions with composition operators. The user can do so by selecting a set of phase descriptions in the right list and choose a composition operator in the right-click menu.

Besides its core of detecting and visualizing phases, *Quala* contains a set of tools to study histories, *i.e.*, entire histories or single phases. These tools are applicable on any kind of a history, for example on a system history or on a class history. They are presented in the next sections.

A.1 Version Property Viewer

The *version property viewer* is a tool to graphically display the graph of one or more measurements over a set of versions. A screenshot of this tool is presented in Figure A.3. It basically consists of two selectable lists and a diagram part. The first selectable list contains a set of versions while the other list contains the set of measurements defined on the kind of versions that the tool is applied on. The selected measurements of the selected versions are then graphically displayed in the diagram part. This part gets automatically updated if the user changes the selection.

Additionally, the *version property viewer* provides a set of buttons to change the selection in the two lists. For both lists, it provides buttons to select all or no elements of the list. On the versions lists, there are furthermore buttons to move the upper or the lower bound of the selection or the entire selection. If the user moves the entire selection down, the lower and the upper bound of the selection are moved down. After applying this change to the screenshot in Figure A.3, the versions Jun011 to Jun015 would thus be selected.



Entities

Jun010 - FAMIXClassVersions(343)
Jun011 - FAMIXClassVersions(343)
Jun012 - FAMIXClassVersions(353)
Jun013 - FAMIXClassVersions(351)
Jun014 - FAMIXClassVersions(351)
Jun015 - FAMIXClassVersions(351)
Jun016 - FAMIXClassVersions(389)
Jun017 - FAMIXClassVersions(379)
Jun018 - FAMIXClassVersions(380)

Properties

ChangedClass
Change_A_NOA
Change_A_NOM
Change_Big Class
Change_Big Method
Change_ClassNesting
Change_ClassWithMethodGrowth
Change_Data Class
Change_God Class

Figure A.3: Version Property Viewer

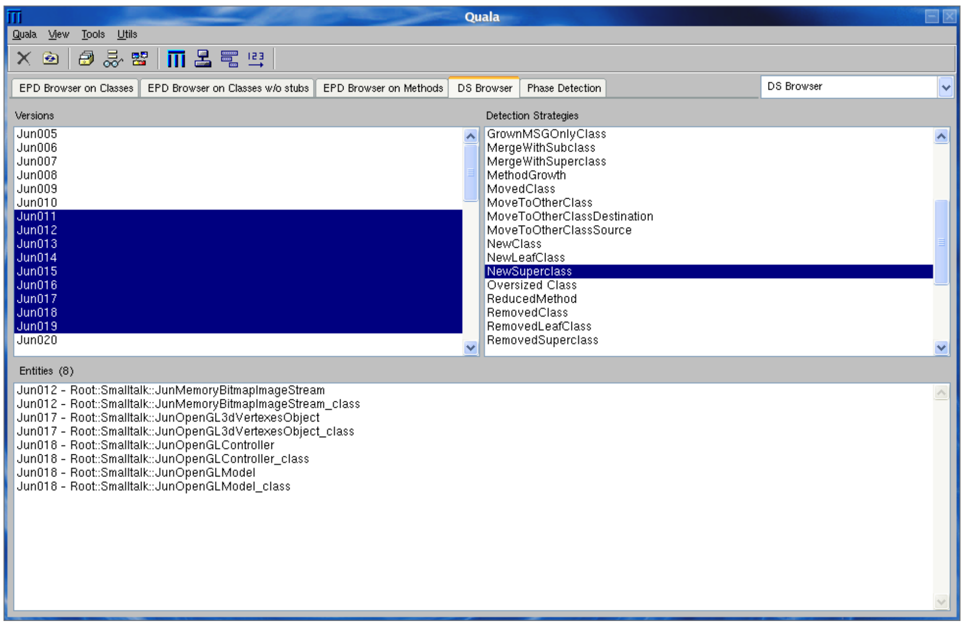
In the context of phases, this tool is helpful to understand the evolution of measurements inside a phase. For example, if we detected a general growth phase with the length of 5 and a high amplitude, we know that in each of the 5 encapsulated phases code has been added, but we do not know how much code in which version has been added. Applying the *version property viewer* on all encapsulated versions of the detected phase shows for example the graph of the number of added and removed classes and thus offers the desired information. We implemented the opening of the *version property viewer* on the versions encapsulated in a phase so that automatically the graph of those measurements that are relevant for the phase are shown. Which measurements are relevant is declared in the definition of the phase description of the phase.

Note that we also use detection strategies as measurements. The situation depicted in Figure A.3 for example shows the graph of the number of new leaf classes. The number of new leaf classes in a version is the size of the set that results by applying the detection strategy *NewLeafClass* on this version. In our implementation, the detection strategy as a measurement is available as soon as the detection strategy is defined.

A.2 Detection Strategy Version Browser

The *detection strategy version browser* is a tool to apply detection strategies on a set of versions. Like the *version property viewer* it provides two selectable lists: One to select a set of versions and one to select one or more detection strategies. The *detection strategy version browser* applies the selected detection strategies on every selected version. In the list below, it then either displays the set of those entities that are

- detected in any of the selected versions, or
- detected in all of the selected versions, or
- not detected in the first but in a succeeding versions of the selection.



```
Jun005
Jun006
Jun007
Jun008
Jun009
Jun010
Jun011
Jun012
Jun013
Jun014
Jun015
Jun016
Jun017
Jun018
Jun019
Jun020
```

```
GrownMSGOnlyClass
MergeWithSubclass
MergeWithSuperclass
MethodGrowth
MovedClass
MoveToOtherClass
MoveToOtherClassDestination
MoveToOtherClassSource
NewClass
NewLeafClass
NewSuperclass
Oversized Class
ReducedMethod
RemovedClass
RemovedLeafClass
RemovedSuperclass
```

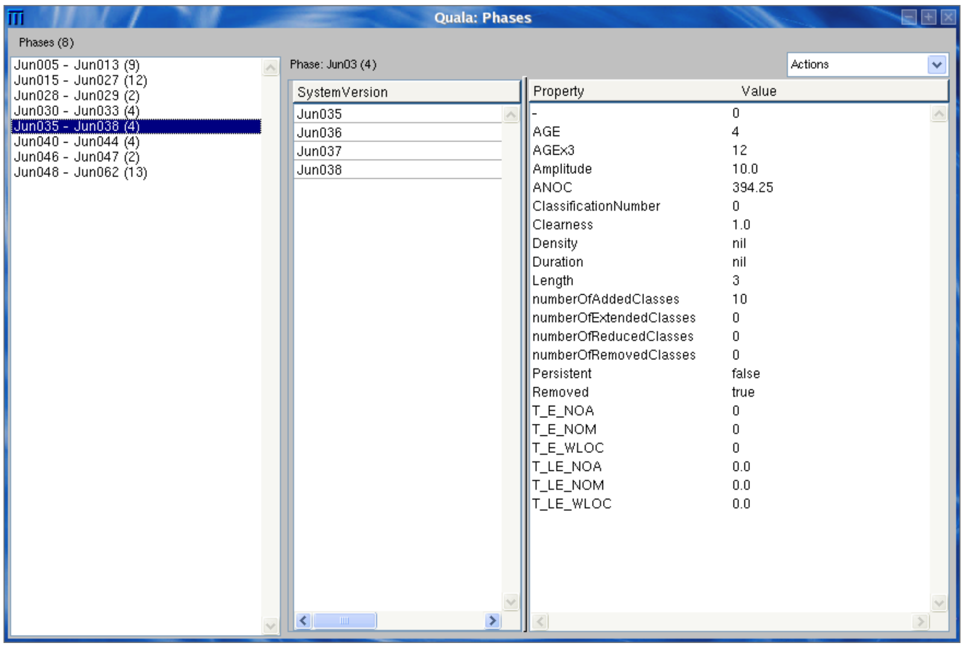
```
Jun012 - Root::Smalltalk::JunMemoryBitmapImageStream
Jun012 - Root::Smalltalk::JunMemoryBitmapImageStream_class
Jun017 - Root::Smalltalk::JunOpenGL3dVertexesObject
Jun017 - Root::Smalltalk::JunOpenGL3dVertexesObject_class
Jun018 - Root::Smalltalk::JunOpenGLController
Jun018 - Root::Smalltalk::JunOpenGLController_class
Jun018 - Root::Smalltalk::JunOpenGLModel
Jun018 - Root::Smalltalk::JunOpenGLModel_class
```

Figure A.4: Detection Strategy Version Browser

In the context of phases, this tool is helpful to further inspect phases of which the phase description is based on one or more detection strategies. For example, the new hierarchies growth phase description is based on the detection strategy *NewSuperclass*. In a further inspection of a detected new hierarchies growth phase, we might want to know which superclasses have been added to the system. This information can be gained by applying the “detection strategy version browser” on all versions that are encapsulated in the phase. In our implementation, the tool automatically selects those detection strategies that are relevant for a phase. In the example, the tool automatically selects the detection strategy *NewSuperclass*. Which detection strategies are relevant is declared in the phase description of a phase.

A.3 Phase Inspector

The phase inspector is a tool to numerically inspect a set of phases, normally the set of phases detected by one phase description. On the left side, it provides a list of phases of which the user can select one. Each line in this list represents one phase textually by indicating the first and the last version and the length of a phase in brackets. On the right side, the selected phase is then displayed with two tables. The left table has a column that contains a list with the versions a phase encapsulates. To this table, the user can add measurement columns of which the measurement values of each version are then displayed. On the right side of this table, there is another table which displays the defined measurement values of the entire phase, *e.g.*, the duration of phase.



Jun005 - Jun013 (9)
Jun015 - Jun027 (12)
Jun028 - Jun029 (2)
Jun030 - Jun033 (4)
Jun035 - Jun038 (4)
Jun040 - Jun044 (4)
Jun046 - Jun047 (2)
Jun048 - Jun062 (13)

SystemVersion
Jun035
Jun036
Jun037
Jun038

Property	Value
-	0
AGE	4
AGEx3	12
Amplitude	10.0
ANOC	394.25
ClassificationNumber	0
Clearness	1.0
Density	nil
Duration	nil
Length	3
numberOfAddedClasses	10
numberOfExtendedClasses	0
numberOfReducedClasses	0
numberOfRemovedClasses	0
Persistent	false
Removed	true
T_E_NOA	0
T_E_NOM	0
T_E_WLOC	0
T_LE_NOA	0.0
T_LE_NOM	0.0
T_LE_WLOC	0.0

Figure A.5: The Phases Inspector

Appendix B

Detection Strategies

In this Chapter we present the list of detection strategies [Mar02] used to define phase descriptions. For each detection strategy, we first present a paragraph that contains a description. The detection strategies are defined as mathematical functions in the paragraph definition. As defined by Marinescu, the input is part of the detection strategy. It is described in the input paragraph of each definition. In this work, we however also apply detection strategies on various inputs. For example, we use the *ReducedMethod* detection strategy to detect reduced methods in single classes and even in all methods included in a system version.

In some detection strategies, we use as their input a set of added or removed classes. This set is in our implementation computed by selecting those class versions in a system versions that do not have a successor respectively a predecessor version. The detection strategies themselves do not enforce that the input classes are newly added to the system or removed but instead presume that.

B.1 Extended Class

Description. This detection strategy detects all classes S' in a certain system version S where at least one method or one attribute has been added. New classes are not detected.

Input: A set of class versions S

Definition

$$ExtendedClass(S) := S' \left| \begin{array}{l} S' \subseteq S,\, \forall C \in S' \\ (\delta NOM(C) > 0) \vee (\delta NOA(C) > 0) \end{array} \right.$$

B.2 Reduced Class

Description. This detection strategy detects all classes S' in a certain system version S where at least one method or one attribute has been removed. New classes are not detected.

Input: A set of class versions S

Definition

$$ReducedClass(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (\delta NOM(C) < 0) \vee (\delta NOA(C) < 0) \end{array} \right.$$

B.3 MSGOnly-Extended Class

Description. This detection strategy detects classes all classes S' in a certain system version S that have the same number of methods and the same number of attributes but a higher number of messages sent or contain a higher number of statements.

Input: A set of class versions S

Definition

$$MSGOnlyExtendedClass(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (\delta NOM(C) = 0) \wedge (\delta NOA(C) = 0) \wedge \\ (\delta MSG(C) > 0) \wedge (\delta NOS(C) > 0) \end{array} \right.$$

B.4 MSGOnly-Reduced Class

Description. This detection strategy detects classes all classes S' in a certain system version S that have the same number of methods ant the same number of attributes but a reduced number of messages sent contain a reduced number of statements.

Input: A set of class versions S

Definition

$$MSGOnlyReducedClass(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (\delta NOM(C) = 0) \wedge (\delta NOA(C) = 0) \wedge \\ (\delta MSG(C) < 0) \wedge (\delta NOS(C) < 0) \end{array} \right.$$

B.5 New Superclass

Description. This detection strategy detects all newly added superclasses S' in a certain system version S, *i.e.,* added classes that have one or more subclasses. The subclasses might be added at same time or already exist.

Input := A set of added class versions S

Definition

$$Superclass(S) := S' \left| \begin{array}{l} S' \subseteq S,\ \forall C \in S' \\ NOC(C) > 0 \end{array} \right.$$

B.6 New Leaf Class

Description. This detection strategy detects all newly added leaf classees S' in a certain system version S, *i.e.*, added classes that have no subclasses.

Input := A set of added class versions S

Definition

$$Superclass(S) := S' \left| \begin{array}{l} S' \subseteq S,\ \forall C \in S' \\ NOC(C) = 0 \end{array} \right.$$

B.7 New Class

Description This detection strategy detects all newly added classes in a system version. Since the input already contains newly added classes only, this detection strategy simply returns the input. This detection strategy provides means to use the number of new classes as a metric.

A class version is considered as new if in the predecessor version there is no class version with the same name.

Input := A set of added classes in a version.

Definition

$$NewClass(C) := True$$

B.8 Removed Class

Description This detection strategy detects all removed classes in a system version. Since the input already contains removed classes only, this detection strategy simply returns the input. This detection strategy provides means to use the number of removed classes as a metric.

A class version is considered as removed if there is no class with the same name in the successor version.

Input := A set of removed classes in a system version.

Definition

$$RemovedClass(C) := True$$

B.9 Split into Superclass

Description. This detection strategy detects classes S' that have a higher class hierarchy nesting level and a lower number of methods, attributes or class variables. It aims at detecting situations where functionality of a class has been pushed down into a newly created superclass.

Input := A set of class versions S

Definition

$$SplitIntoSuperclass(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (\delta HNL(C) > 0) \wedge ((\delta NOM(C) < 0) \vee \\ (\delta NOA(C) < 0) \vee (\delta NCV(C) < 0)) \end{array} \right.$$

B.10 Merge with superclass

Description. This pd detects classes that have a lower nesting level in the hierarchy but more methods, attributs or class variables. It aims at detecting situations where a superclass has been removed and part of it's functionality has been moved into one or more of its subclases.

Input := A set of class versions S

Definition

$$MergeWithSuperclass(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (\delta HNL(C) < 0) \wedge ((\delta NOM(C) > 0) \vee \\ (\delta NOA(C) > 0) \vee (\delta NCV(C) > 0)) \end{array} \right.$$

B.11 Split into Subclass

Description. This detection strategy detects class versions that have a higher amount of subclasses and a lower amount of methods, attributes or class variables. It aims at detecting situations where functionality was moved from one class into one or more newly created subclasses.

Input := A set of class versions S

Definition

$$SplitIntoSuperclass(S) := S' \left| \begin{array}{l} S' \subseteq S,\, \forall C \in S' \\ (\delta NOC(C) > 0) \land ((\delta NOM(C) < 0) \lor \\ (\delta NOA(C) < 0) \lor (\delta NCV(C) < 0)) \end{array} \right.$$

B.12 Merge with Subclass

Description. This detection strategy detects the set of class versions S' that have a lower count of subclasses but a higher amount of methods, attributes or class variables.

Input := A set of class versions S

Definition

$$MergeWithSubclass(S) := S' \left| \begin{array}{l} S' \subseteq S,\, \forall C \in S' \\ (\delta NOC(C) < 0) \land ((\delta NOM(C) > 0) \lor \\ (\delta NOA(C) > 0) \lor (\delta NCV(C) > 0)) \end{array} \right.$$

B.13 Move to Other Class Source

Description. This detection strategy aims at detecting class versions where functionality has been moved away from, *i.e.*, where functionality has been removed. It detects class versions that do not change their hierarchy nesting level or their number of subclasses (children) but have less methods, attributes or class variables.

Input := A set of class versions S

Definition

$$MoveToOtherClassSource(S) := S' \left| \begin{array}{l} S' \subseteq S,\, \forall C \in S' \\ ((\delta NOM(C) < 0) \lor (\delta NOA(C) < 0) \lor \\ (\delta NCV(C) < 0)) \land (\delta HNL(C) = 0) \land \\ (\delta NOC(C) = 0) \end{array} \right.$$

B.14 Move to Other Class Destination

Description. This detection strategy aims at detecting class versions where functionality has been moved to, *i.e.*, where functionality has been added. It detects class versions that do not change their hierarchy nesting level or their number of subclasses (children) but have a higher amount of methods, attributes or class variables.

Input := A set of class versions S

Definition

$$MoveToOtherClassDst(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ ((\delta NOM(C) > 0) \vee (\delta NOA(C) > 0) \vee \\ (\delta NCV(C) > 0)) \wedge (\delta HNL(C) = 0) \wedge \\ (\delta NOC(C) = 0) \end{array} \right.$$

B.15 Extract Method

Description. This detection strategy aims at detecting class versions where the refactoring *split method* has been applied, *i.e.,* where one or more methods have been split up into multiple methods. It detects class versions that have a higher amount of methods and a lower amount of messages sent (summed up over all methods) and where one or more methods have been reduced. Reduced methods are detected by applying the *ReducedMethod* detection strategy (Appendix B.17) on all method versions of a class version.

Input := A set of class versions S

Definition

$$ExtractMethod(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ (|ReducedMethod(C)| > 0) \wedge (\delta NOM(C) > 0) \wedge \\ (\delta MSG(C) < 0) \end{array} \right.$$

B.16 Extended Method

Description. This detection strategy detects all method version in a set of method versions that have a higher count of messages sent and a higher count of statements compared to the previous version.

Input := A set of method versions S; for example all method versions of one class version or all method versions in an entire system version.

Definition

$$ExtendedMethod(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall M \in S' \\ (\delta MSG(M) > 0) \wedge \delta NOS(M) > 0) \end{array} \right.$$

B.17 Reduced Method

Description. This detection strategy detects all method version in a set of method versions that have a lower count of messages sent and a lower count of statements compared to the previous version.

Input := A set of method versions S; for example all method versions of one class version or all method versions in an entire system version.

Definition

$$ExtendedMethod(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall M \in S' \\ (\delta MSG(M) < 0) \wedge \delta NOS(M) < 0) \end{array} \right.$$

B.18 Removed Leaf Class

Description. This detection strategy detects all classes S' that have been removed and that have been leaf clases, *i.e.*, had no subclasses.

Input: A set of class versions S that have been removed

Definition

$$RemovedLeafClass(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ NOC(C) = 0 \end{array} \right.$$

B.19 Removed Superclass

Description. This detection strategy detects all classes S' that have been removed and that have been superclasses, *i.e.*, had one or more subclasses.

Input: A set of class versions S that have been removed

Definition

$$RemovedLeafClass(S) := S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ NOC(C) > 0 \end{array} \right.$$

B.20 Shrunk and Subclases Class

Description. This detection strategy aims at detecting situations where functionality has been moved away from a class into a newly created subclass. It detects all class versions S' that have less attributes or methods and a higher number of subclasses.

Input: A set of class versions S

Definition

$$RemovedLeafClass(S) := S' \left| \begin{array}{l} S' \subseteq S,\ \forall C \in S' \\ (\delta NOA(C) > 0 \vee \delta NOM(C) > 0) \wedge \\ \delta NOC(C) > 0 \end{array} \right.$$

Appendix C

Applied Metrics

In the following three tables we present the structural metrics we applied in this work. Table C.1 contains the metrics on the system level, Table C.2 the ones on class level and Table C.3 contains the applied method level metrics.

Acronym	**Name**	**Description**
NOCL	Number of classes	The total number of classes in the system
NOM	Number of methods	The sum of the number of methods of every class in a system version

Table C.1: Applied System Metrics

Acronym	**Name**	**Description**
NOM	Number of methods	The number of methods defined in a class
NOA	Number of attributes	The number of instance variables defined in a class
NCV	Number of class variables	The number of class instance variables defined in a class
NOS	Number of statements	The sum of the number of statements of every method of a class
MSG	Number of messages sent	The number of messages sent in all methods of a class
NOC	Number of children	The number of direct subclasses of a class
HNL	Hierarchy nesting level	The nesting level of a class in it's inheritance tree, *i.e.*, the number of superclasses

Table C.2: Applied Class Metrics

Acronym	**Name**	**Description**
MSG	Number of messages sent	The number of messages sent in a method
NOS	Number of statements	The number of statements in a method

Table C.3: Applied Method Metrics

List of Figures

List of Tables

Bibliography

[AHK$^+$01] 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.

[BM99] 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.

[CJH] Stephen Cook, He Ji, and Rachel Harrison. Software evolution and software evolvability.

[DDN00] 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.

[DGF04] 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)*, 2004. to appear.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[EGK$^+$01] 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 Trans. Softw. Eng.*, 27(1):1–12, 2001.

[FBB$^+$99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[GDL04] 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 (International Conference on Software Maintenance)*, 2004.

[GHVJ93] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[GJKT97] 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.

[GL04] Tudor Gîrba and Michele Lanza. Using visualization to understand the evolution of class hierarchies, 2004.

[GM03] 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.

[GR95] Adele Goldberg and Kenneth S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison Wesley, Reading, Mass., 1995.

[GSV02] 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.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 131. IEEE Computer Society, 2000.

[Jaz02] Mehdi Jazayeri. On architectural stability and evolution. In *Reliable Software Technlogies-Ada-Europe 2002*, pages 13–23. Springer Verlag, 2002.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JGR99] Mehdi Jazayeri, Harald Gall, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM '99 Proceedings (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society, 1999.

[JH99] McDermid J. and Bennet K. H. Software engineering research in the uk: a critical apprisal. In *IEEE Proceesings - Software, vol. 146, no. 4*, pages 179 – 186, 1999.

[Jon98] T. Capers Jones. *Estimating Software Costs*. McGraw-Hill, Inc., 1998.

[Lan00] Michele Lanza. Codecrawler, 2000. http://www.iam.unibe.ch/~lanza/CodeCrawler/codecrawler.html.

[Lan01] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[Lan03] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[LB85] Manny M. Lehman and Les Belady. *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[LD03] 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.

[LD04] Michele Lanza and Stéphane Ducasse. Codecrawler an extensible and language independent 2d and 3d software visualization tool. In *Reengineering Environments*. tba, 2004. to appear.

[Leh96] Manny M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[LPR$^+$97] M.M. Lehman, D. E. Perry, J. F. Ramil, W. M. Turski, and P. D. Wernick. Metrics and laws of software evolution - the nineties view. In *Metrics '97, IEEE*, pages 20 – 32, 1997.

[LPR98] M. M. Lehman, Dewayne E. Perry, and Juan F. Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM 1998)*, pages 208–217, 1998.

[Mar01] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of TOOLS*, pages 173–182, 2001.

[Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. Ph.D. thesis, Department of Computer Science, ”Politehnica” University of Timişoara, 2002.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Raţ03] 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.

[RDGM04] 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.

[Ren03] Lukas Renggli. Smallwiki: Collaborative content management. Informatikprojekt, University of Bern, 2003.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[Som00] Ian Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[TDD00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. FAMIX and XMI. In *Proceedings of WCRE 2000 workshop on Exchange Formats*, November 2000.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000 (International Conference on Software Evolution)*, pages 157–167. IEEE, 2000.