DISSERTATION

# sPACE
# Software Project Assessment in the Course of Evolution

ausgeführt
zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Harald C. Gall
Institut für Informatik
Universität Zürich
und
o.Univ.Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
Institut für Informationssysteme
Technische Universität Wien

eingereicht
an der Technischen Universität Wien
Fakultät für Informatik

von

**Dipl.-Ing. Jacek Ratzinger**
*ratzinger@infosys.tuwien.ac.at*

Matrikelnummer: 9725085
Birneckergasse 48, 1210 Wien, Österreich

Wien, October 2007

# KURZFASSUNG

Das Softwaregeschäft und dessen Technologien entwickeln sich rasant weiter und die Anforderungen an Unternehmen und Projekte steigen. Die Größe als auch die Komplexität von Entwicklungsprojekten haben ein Niveau erreicht, bei dem es schwer ist, zuverlässige Aussagen über die Qualität zu machen. Personen, die für das Steuern von Softwareprojekten verantwortlich sind, müssen wissen, wie sich das Produkt entwickelt, um auf Projektebene reagieren zu können. Informationen über Softwareevolution ermöglichen die Eigenschaften des resultierenden Softwaresystems einzuschätzen, wie zum Beispiel die erwartete Anzahl an Fehlern, wodurch viele projektrelevante Entscheidungen unterstützt werden (z.B. welche Aktionen als nächstes umgesetzt werden sollten, damit ein besseres Produkt erzeugt werden kann).

Diese Dissertation zielt auf die Entwicklung von Werkzeugen und Methoden zur Bewertung von Softwareprojekten ab. Die Entwicklung eines Sofwaresystems kann durch eine Sequenz von Ingenieursaktivitäten beschrieben werden, welche auf einem hohen Abstraktionsniveau zu Prozessmodellen der Softwareentwicklung zusammengefasst werden. In solchen Aktivitätssequenzen sind die einzelnen Ereignisse nicht isoliert, sondern mit einander verknüpft. Für die Bewertung der gegenseitigen Abhängigkeiten in Projekten können wir Informationssysteme verwenden, welche die Entwicklungsaufgaben unterstützen. Als Quellen für die Information über die Evolution verwenden wir daher Konfigurationsmanagementsysteme (z.B. CVS und Subversion) und Ticketing-Systeme (z.B. Jira und Bugzilla). Da die gewonnenen Datenmengen sehr groß sind, ist eine manuelle Analyse beinahe unmöglich. Daher wenden wir verschiedene Data-Mining Methoden zur Extraktion relevanter Fakten an. Association-Mining ermöglicht die Identifikation von Verknüpfungen zwischen Softwareelementen sowie die Beschreibung der Architektur von einer Evolutionsperspektive. Zukünftige Aktivitäten and Produkteigenschaften können mit Hilfe von Methoden für Regression und Klassifikation vorhergesagt werden.

Wir haben bestehende Ansätze durch Data-Mining von Werteserien erweitert, welche auf Evolutionsattributen basieren und den Verlauf der Entwicklung über die Zeit beschreiben. Die Erkennung von sequenziellen Mustern ist essenziell, da sie zur Verbesserung der Genauigkeit von Vorhersagemodellen genutzt werden kann. Als Grundlage für Series-Mining zur Vorhersage der Anzahl von Fehlern rekonstruieren wir die Aktivitätstypen des Softwareevolutionsprozesses. Das Verhältnis zwischen unterschiedlichen Typen von Aktivitäten bietet sehr gute Ergebnisse mit einer Korrelation von mehr als 0,9 zwischen der vorhergesagten und tatsächlichen Anzahl von Fehlern. Diese Werte basieren auf Vorhersagemodellen, welche Aktivitäten einbeziehen, die durch Ausdrücke wie ”refactor” oder ”comment” gekennzeichnet werden. Daher erreichen unsere Vorhersagemodelle eine sehr hohe Genauigkeit, was durch die Verwertung der sequentiellen Eigenschaften von Softwareevolution erreicht wird. Für die Bewertung von Softwareprojekten definieren wir eine große Anzahl an Evolutionsmetriken und beschreiben Techniken wie diese Metriken genutzt werden können, um zukünftige Ereignisse zu antizipieren. Unsere Vorhersagemodelle fokussieren auf kurze Zeitintervalle und ermöglichen es Refactorings und Fehler der nächsten zwei Monate vorherzusagen, wobei wir Evolutionsmetriken der vorherigen zwei bis drei Monaten verwenden.

# ABSTRACT

The software business and its technology continuously advance and the requirements on manufacturers of software systems increase. The size and complexity of development projects have reached a level, where it is difficult to make reliable statements about quality. People responsible for steering software projects need to know how the application evolves to react on project level. Information on software evolution can be used to assess the characteristics of the resulting system, which provides support for many project related decisions (e.g. which activities should be realized next to achieve a better product in terms of quality).

This thesis aims at the development of tools and techniques for the assessment of software projects. The development of a system can be described through a sequence of engineering events. On a high level, these events are grouped to process models for software development. In such sequences events are rarely isolated but related to each other. For the assessment of the interdependencies in products and projects we can utilize information systems of today that support development projects. Our sources of evolution information are configuration management systems (e.g. CVS and Subversion) and issue tracking systems (e.g. Jira and Bugzilla). The amount of data from such tools is very large and the manual investigation is almost impossible. Therefore, we apply different data mining methods for the extraction of relevant facts. Association mining enables us to identify coupling between software entities and to describe the architecture from an evolution perspective. Future activities and product attributes can be anticipated with the help of regression and classification methods. We evaluate our approach based on a field study of commercial and open source projects.

We extend the basic data mining approaches with the mining of value series, which are based on evolution attributes describing the course of development over time. The recognition of sequential patterns is essential, because it can be exploited to improve the accuracy of our prediction models. For the input to this series mining algorithm we reconstruct the event types of software evolution processes. The relation between different types offers very good results with a high correlation of more than 0.9. These values are reached with prediction models in which events are involved that are described through the terms ”refactor” or ”comment”. Thus, our predictions have a very high accuracy, which is due to the exploitation of the sequential nature of software evolution. For the assessment of software projects we define a large number of evolution metrics and techniques to apply them for the anticipation of future events. Our prediction models focus on short time frames and allow us to predict refactorings and defects in the next two months based on evolution metrics from just the previous two or three months.

# ACKNOWLEDGMENTS

It is a pleasure for me to thank the many people who made this thesis possible.

First, I am greatly indebted to my advisor Prof. Harald Gall who gave me the possibility to pursue my research ideas. I want to thank him for fruitful discussions and for sharing his great knowledge with me.

Additionally, I owe sincere gratitude to Prof. Mehdi Jazayeri who is an active advocate of diversification. His great work in leading the Distributed Systems Group at the Vienna University of Technology allows many different researchers growing in their research directions. This gave me the opportunity to do this PhD in the creative work environment that he established.

Special thanks go to my current and former colleagues at Tiani Medgraph, GWI, and Agfa HealthCare. I am indebted to Martin Tiani who gave me the opportunity to start research in an industrial environment. Additionally, I want to thank Christoph Becker, Rainer Wegenkittl, and Peter Steiger, who made it possible to work in parallel in an interesting industrial environment and in the world of science.

I want to express my gratitude to many other researchers for the inspiring cooperation and the proof reading of the first drafts on many ideas: Serge Demeyer, Peter Vorburger, Martin Pinzger, Beat Fluri, Jayalath Ekanayake, Abraham Bernstein, Patrick Knab, Emanuel Giger, and many others.

I am glad that Martin Schlager, a dear friend, was crazy enough to start the PhD studies together with me despite our jobs in the software engineering industry, which became a funny race to see who will keep on track and finish the PhD.

Many thanks to my parents who greatly supported me over the years and steadily encouraged me to keep on track with my studies. Without them all this would have never been possible.

I especially thank my wife Cornelia for all her support. Without her help this thesis would not have been possible. She was understanding when I was working long hours on this research project that took away a substantial fraction of our scarce time. On the other hand, she motivated me to continue with the thesis when I was not focused. Also her practical support was very valuable in proofreading my publications and in sharing her knowledge on the supervision of undergraduate students.
Cornelia, thank you very much for our great time together!

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

To be successful in the software industry it is important to provide solutions on time and within budget, which requires to continuously improve the development methodology. For this goal a thorough understanding of the underlying technical and project related issues is necessary. This thesis called sPACE is about sustainable pace in software projects. It describes techniques for the assessment of software projects and provides support for several roles in a software project.

Software engineers are focused on the technical aspects of the project and in particular on development of the software product. We utilize data on software evolution to provide an architectural inspection of the product based on its development history. Additionally, we create prediction models for future events in the project. The structure of the models and the involved attributes help software engineers to understand the influences on the resulting product and in particular its quality measured in terms of defect densities.

Project managers have to focus on keeping software projects on track and to deliver the product on time an with the desired characteristics. To fulfill their role they need a more coarse-grained view on the project and the product in comparison to software engineers. Our prediction models support project managers in several ways. First, they provide information about the expectations (predictions) for the near future and allow taking the right decisions. Second, the models also point out the interrelations of the underlying aspects in software projects, which helps to guide the involved people in the right way.

One of the management related tasks is the right use of the (scarce) resources. This aspect is important for multiple roles in software projects and in particular for the testing team. Due to the fact that this group does not focus on ”productive” work, but support in the quality assurance of different aspects of projects and products, the resource limitations are often very tight for the testing people. sPACE supports to focus on elements that are expected to miss the desired characteristics. For example the parts of the product with the highest probability for defects could be tested spending more time and investing this limited resource.

## 1.1 MOTIVATION

Software maintenance as a phase of the software life cycle is a very resource consuming task. An effective way to overcome or avoid the negative effects of software aging is by placing *change* in the center of the software development process. Without explicit and immediate support for change and evolution, software systems become more and more complex, fragile, and unreliable. This negative spiral is rapidly shortening due to hardware and business innovations, changes in legislation and continuing internationalization. We must therefore advance beyond the current software engineering practice and provide fundamental support for software change and evolution.

When software systems evolve, we need to measure the quality of the systems to support evolvability. Therefore, certain characteristics of software are critical for the success of software projects like the remaining number of defects. During evolution, software is constantly changed and adapted to new requirements. This includes bug fixes during the entire evolution period. According to the staged life cycle [100], bug fixes are the last kind of software changes that take place after the transition from the evolution phase to the servicing phase.

To estimate the quality of a software system for project purposes, we need to measure the data about the evolution itself. This data provides information about the change events that took place. All events together build up an evolution process that could be described by standards of software processes such as the Rational Unified Process (RUP [61]). The attributes of events in the software evolution process are essential to evaluate the software development to manage the project properly. For example the frequency of refactoring (see Section 2.6) and testing cycles provide insights into the quality of the software. Further attributes such as the number of detected defects during the testing cycles are important measures for software projects following a software process [29].

## 1.2 RESEARCH STATEMENT

This thesis *investigates software evolution patterns to effectively assess software projects*. Estimation models for large object-oriented software systems are built by exploiting the huge amounts of information that resides in version control and bug tracking systems, which has been largely ignored by software engineering so far.

Software vendors are interested in the characteristics of software systems before they are installed on customer sites. A wide range of prediction models has been proposed (see Section 2.7), in which often size and complexity metrics are used, such as McCabe [79] or Halstead [57]. These models are intended to predict the number of defects in a system or to estimate the reliability of the system in terms of time to failure.

However, size and complexity metrics are often not sufficient for the prediction of defects and other software quality characteristics [34]. The predictive power is not accurate enough and the interpretations of the results are inconsistent. As software systems can be measured in many

dimensions, taking a single metric seems to be inappropriate. Other types of measures for software quality seem more promising. For instance, prediction models involving careful collection of data about defects discovered during early inspection and testing phases provide impressive accuracy.

Companies in Japan [22,71] as well as the NASA Space shuttle team [66] claim to achieve results on defect prediction within 95 percent confidence limits. The idea of these quality estimation studies relies on the phases of the software development process. With predefined phases at which data is collected, statistical extrapolation techniques are used to predict the defect rate of the next phase.

The goal of this thesis is to provide means for software project assessment and controlled software evolution based on evolution data at various levels. Specifically, it aims to answer the following questions:

- Q1: *How to set up metrics from sources such as modification reports and process/project management tools?* We are particularly aiming at investigating this integration of data sources for evolution analysis and software project assessment.
- Q2: *How do software evolution metrics relate to external software product attributes?* We analyze the distribution and correlation of selected metrics and study their effects on software quality attributes such as the number of defects.
- Q3: *How can data mining be efficiently utilized in software projects to improve software evolution?* We inspect events and phases of software projects to assess the quality of the resulting products and services and to provide prediction models in software project assessment.
- Q4: *How can models on evolution processes be effectively adapted for the assessment of software projects?* We evaluate the influence of process events during software evolution on software quality and predictability of quality attributes.
- Q5: *How to set up effective models that result in prediction with high accuracy?* We focus on this question in several experiments and investigate the influence of different metrics covering multiple aspects of software evolution.

We propose to tackle these questions by exploiting and understanding the huge amounts of information which reside in release history data. We focus on large-scale software systems both from the open source and the industrial area. We evaluate sPACE with the help of a field study consisting of three different software projects. Two of them are representatives of the open source community. The first one is ArgoUML, which is a widely used UML engineering tool for the design of systems. As second project we selected the Spring framework, which is an application server well known in the context of enterprise Java programs. ArgoUML and Spring framework are large projects both developed in the Java programming language and consist of about 5.000 and 10.000 classes [1], respectively. The third project that complements the field study is a large

[1] in the entire document classes are a synonym for object-oriented classes

software system of a health care company for radiology applications. It is a Picture Archiving and Communication System (PACS) that includes a viewing workstation for the visualization of radiological material and an archive. The radiological material is acquired from different modalities such as magnetic resonance, computer tomography, or ultrasound scanning. This system is written in Java and currently contains more than 8.600 classes with 735.000 lines of code. In Java classes are almost equivalent to files, thus we use files as basic instances in our approach.

## 1.3 CONTRIBUTIONS

The objective of this thesis is to define effective models for the assessment of software engineering projects and especially to estimate the quality. The contributions of this thesis comprise:

- *Definition of multiple evolution measures.* The definition of a large set of evolution features measures different aspects of software prjectes for their assessment. Based on the data from software repositories we extract evolution attributes grouped in several categories such as size, team relationships, process orientation, complexity of existing solution, difficulty of problem, relational aspects, time constraints, and testing.

- *Effective predictions on short time frames.* We show that our prediction approaches utilize only two or three months of development time for the prediction of subsequent two months. Thus, project managers can estimate the best release date based on the lowest number of predicted defects.

- *Different types of events in software processes can be predicted.* In addition to defects, our approach allows us predicting other events such as refactoring, which is one type of perfective maintenance in modern software projects.

- *Locations that should be refactored from a quality perspective can be detected based on evolution data.* Refactoring as a state-of-the-art technique in the field of software engineering should be thoroughly applied in a software project to maximize the effectiveness for future project trends. We identify locations needing refactoring utilizing evolution analysis and inspect the system for a period after the applied refactoring, which provides insight whether the refactoring show the right effect.

- *Prediction models with very high accuracy taking time series and process information into account.* Evolution metrics change over time. This fact should be taken into account for improvements of prediction models. With genetic programming we automatically select transformations and functions on value series to extract meta-features for our prediction models.

Field studies illustrate the applicability of our approach to several projects from multiple domains with different business models (open source and commercial).

## 1.4 THESIS OUTLINE

The remainder of the thesis is structured as follows:

**Chapter 2** introduces different research areas to which sPACE is related to. Especially previous work on software evolution has a deep impact on the current work. Evolution activities and refactoring, as one of their prominent representatives, are used in this thesis in several ways: As input to assessment, as target element that has to be identified, and as techniques to improve evolvability. Our assessment of software projects focus on different aspects of software quality such as evolvability of software systems and reliability measured through the number of defects. For the assessment we use metrics and data mining techniques, which are discussed in this chapter as well as prediction approaches in the context of software evolution.

**Chapter 3** settles the foundation of this research work. In this chapter, we describe the extraction of evolution information from different data sources and the processing of the data to define multiple measures of evolution attributes. This provides an answer of Q1 of Section 1.2 on the set-up of metrics from modification reports and process/project management tools.

**Chapter 4** gives a motivating example how evolution data can be applied to project assessment. The building block of this first step in sPACE is co-change coupling, which is detected with the help of a technique that is similar to association mining. The co-change coupling is then explored through visualization and navigation facilities that we support through a tool called EvoLens. The reference to association mining directs us towards data mining, which is extensively used in the following chapters.

**Chapter 5** shows how to use metrics and data mining for the identification of design shortcomings, which are software entities that demand refactoring in the future. According to Fowler [38] refactoring is a technique to improve the design of existing code. Thus, if a class in an object-oriented system needs refactoring, it indicates that the design has deficiencies. In contrast to Chapter 4, Chapter 5 does not require the interaction of the user on the visualization, but detects the locations of design weaknesses automatically with the help of data mining.

**Chapter 6** focuses on an aspect of customer perceived quality that is approached with the identification of defect-prone classes. We analyze the predictability of different categories of defects such as the ones before a software release in contrast to the ones after it and compare the accuracy of the different prediction models. Additionally, this chapter provides an insight into the reason of defect proneness trough an evaluation of the evolution measures that contribute best to the prediction models.

**Chapter 7** introduces a general approach for the improvement of predictions in the field of software engineering. Software evolution is a continuous process where different types of activities are applied on software entities to satisfy customer requirements. As a result,

we have to incorporate the information about the sequential nature of evolution into our models. Based on value series we develop prediction models utilizing linear regression and genetic programming. The results show that by focusing on series mining we obtain models with very high correlations.

**Chapter 8** describes the final step of sPACE where information about the software evolution process is used as an input to our prediction approach. Therefore, we recover the building blocks (i.e. events) of evolution processes and show that this information together with the series mining of Chapter 7 is sufficient to establish prediction models with high correlation and low error values. An analysis of the events that enable the creation of the best prediction models provides an insight into the evolution process and the relationship between evolution events and software defects.

**Chapter 9** discusses the results of this thesis and relates them to the research questions raised in Section 1.2. Additionally, it describes the benefits of sPACE in the context of software projects and shows how the different roles can use the techniques described in this thesis to improve their way of working.

**Chapter 10** concludes the thesis and summarizes the major contributions of this work. It describes the lessons learned in sPACE and gives an outlook on potential future work.

# Chapter 2

# Related Research Areas

In this chapter, we give an overview of the related work of other researchers that concern this thesis. The basics for sPACE are given by research in the field of software evolution. The software evolution phenomenon was especially recognized within the research done by Lehman and Belady [73]. Software evolution includes the phases of initial development and continuous maintenance. We obtain data about the software evolution from versioning systems and issue tracking systems and use several processing steps to extract information for the assessment of the software itself and the development processes and projects. An insight into the research done on software evolution is given in Section 2.1.

The information about software evolution is utilized in this thesis for the assessment of software projects and the developed software products. We particularly aim to investigate software quality, which is approximated through the number of defects. However, defects are just one of the measurable effects of quality factors such as understandability, changeability, testability, etc. Several quality models and the related factors and attributes are presented in Section 2.2. In addition to the number of defects, sPACE focuses on evolvability as a quality aspect. With our architectural analysis of co-change couplings in Chapter 4 we support the understanding of the forces that influence evolvability. Also the prediction models of the subsequent chapters and their internal structure provide insight into the interrelations of the factors in software projects that influence the resulting product and its quality.

To be able to assess software projects we must measure their attributes and condense them into facts. This leads to the next related research area of metrics that is elaborated in Section 2.3. According to Daskalantonakis [24] software metrics can be defined as ”a method of quantitatively determining the extent to which a software process, product, or project possesses a certain attribute. This includes not only the formula used for determining a metric value, but also the chart used for presenting metric value as well as the guidelines for using and interpreting the chart and metrics in the context of specific projects.” In this thesis we measure evolution attributes and use them as input to our investigations and the creation of assessment models.

The data base containing all the evolution metrics that are defined in this thesis is very large. This makes the assessment for human beings a tedious and difficult task. As a result we use data mining and related techniques in sPACE to support fact extraction from the large amount of data.

In Section 2.4 we provide an overview of research in the field of data mining, which is a technique for discovering recurring patterns in data. In this thesis we investigate software evolution, which is a continuous process and demands appropriate techniques for its analysis. As classical data mining techniques focus on a fixed state and build models to reveal the dependencies between the input values, we use sequential data mining techniques in the field of software engineering as one of the first researchers. The approaches to sequential data mining are introduced in Section 2.4.1.

All tasks within software development projects can be described on a high level as software processes. Several processes have been proposed for software development such as the Rational Unified Process (RUP) [61] or Extreme Programming (XP) [7]. But what about the actual course in software projects? This information is an important input for project assessment and therefore we reconstruct process events from software evolution data as an input to the data mining techniques. The research area of software processes is further elaborated in Section 2.5.

Software processes contain different types of events such as adaptive, corrective, and perfective activities. Refactoring is one of the perfective activities as it is a technique to improve the structure of existing software without changing the external behavior [38]. This type of software evolution is particularly interesting in the context of software quality as it is often used to improve understandability and changeability of a system. Thus, we expect that information about refactorings in software projects provide valuable input for the prediction techniques in sPACE. We describe research related to refactoring in Section 2.6.

The ability to predict future events and results is strongly requested in software projects as several different people can profit, which is described in Chapter 9. Prediction models based on data mining are one of the cornerstones of sPACE. The existing approaches to predictions in software engineering are described in Section 2.7.

## 2.1 Software Evolution

In [43] evolvability is defined as ”the capability of software products to be evolved to continue to serve its customer in a cost-effective way.” Many steps have to be fulfilled to respond to a change request of a user: First, the current system has to be analyzed and it has to be identified how the request can be satisfied. This analysis leads to the location which parts of the software have to be changed. Second, the implementation of the necessary changes may demand further adaptation of other parts. Thus, the software architecture has to be revised after the modifications. Third, the changed product is tested and the correctness of the functionality is ensured. Therefore, the evolvability of a software system is determined by:

- Analyzability
  Is the ability to of the system to be clear and to enable the analysis by software engineers to implement necessary changes.
- Changeability
  The system is transformable from one stable state to another.

- Stability
  The architecture of the system must be flexible enough to allow necessary changes to be made, whereas the existing functionality should not be harmed.

- Testability
  Functional and non-functional properties of the system have to be guaranteed to serve the customers needs.

- Compliance to standards
  If a product is developed according to standards, the evolution of this product is simplified.

Lehman and Belady's *Laws of Software Evolution* [73] establish that as systems evolve, they become more complex, and consequently more resources are needed to preserve their structure. They also establish that *successful* systems / E-type systems (i.e. embedded in a real-world environment) *must change*, or they become progressively less useful in that environment. Lehman, Perry and Ramil explored the implication of evolution metrics on software maintenance [74] [75]. They used the number of modules to describe the size of a version and defined evolutionary measurements which take into account differences between consecutive versions. By focusing on the dynamic behavior of a relatively large and mature (12 years old) system, they made a number of observations about the size and complexity growth of the system. These investigations led to the laws of software evolution:

- Continuing change
- Increasing entropy (or: complexity)
- The fundamental law of software evolution
- Conservation of organization stability (invariant work rate)
- Conservation of familiarity
- Continuing growth
- Declining quality
- Feedback system

The software life cycle and its stages were subject to many discussions. One of the models defining the different steps software passes is the staged model for the software life cycle of Rajlich and Bennett [100]. They state that the software life cycle should be changed to place evolution at its center.

There are several approaches that analyze the influence of changes in an evolving software system: Burd and Munro analyzed the influence of changes on the maintainability of software systems by defining a set of measurements to quantify the dominance relations which are used to depict the complexity of the calls [15]. Gold and Mohan defined a framework to understand the

conceptual changes in an evolving system [53]. Based on measuring the detected concepts, they could differentiate between different maintenance activities. In terms of change effects, impact analysis approaches (e.g. [5]) attempt to determine, given a point in the source code involved in a modification task, all other possible points in the code that are transitively dependent upon this seed point. Many of these approaches are based on static slicing (e.g. [45]) or dynamic slicing (e.g. [2]).
The development process has large impact on program complexity and affects software related events such as release dates. Negative effects on software can be detected by examining logs of the source code repository [58]. Mathematical concepts from information theory can guide the investigation of software evolution. Textual descriptions of changes often indicate the type of the performed change. Mockus et al. [83] discovered that a strong relationship exists between the type and size of a change and the time required to carry it out. Gefen and Schneberger [48] explored two distribution patterns of software maintenance modifications (constant and decreasing) to determine if the distribution is homogeneous. They noted that generally the rate of maintenance modifications decreases over time, but not if viewed in individual phases. They distinguish three different phases: stabilization, improvement, and expansion.
Rajlich and Bennet [100] identified several stages in their ”staged life cycle model” of software systems:
*Initial development:* During this phase of the life cycle the first version of the software is developed. The finished product is shipped to the user. This phase is crucial to the remaining lifetime of the system, as appropriate software architecture of the application domain has to be established.
*Evolution:* When a system is exploited in a real world environment, the ongoing changes of the environment have to be reflected in the software to handle new requirements. The established needs of the customer have to be satisfied, through (possibly) major extensions of the functionality and adaptations.
*Servicing:* Based on the symptoms of software aging, changes to the system are entailed with growing expense and effort. At a certain point in time it is not possible to evolve the system any more, because each extension introduces more errors and thereby losing the ability to serve customer needs. Thus, only minor defect repairs and simple functional changes are applied in the servicing phase.
*Phase-out:* When changes are not supported by a transparent architecture, servicing cannot be fulfilled anymore. The company tries to gain benefits from the unchanged software as long as possible. The users have to work around existing pitfalls.
*Closedown:* Finally, the product is withdrawn from the users and possibly replaced by a new software system.
For most software projects the initial development and the evolution phase are responsible for almost the entire effort measured in financial expenses and consumed time. The longer a system exists the more important becomes the evolution phase of the software life cycle. Many business-critical applications are in place for many years or even decades. These systems are often very large.
In addition to source code repositories, several other information sources such as mail messages and defect reports can by explored to get a better understanding how a software product has

evolved since its conception [49]. In [52] four different kinds of studies for software evolution are presented and compared. The studies consider long-running observations of growth and evolution as well as fine-grained issues such as code cloning and software architecture. Software evolution analysis is a very computational intensive task, where Bevan et al. have implemented a system called Kenyon for the efficient fact extraction from data sources such as software repositories and bug tracking systems and storage of the evolution information for further analysis [8].

Besides this work, some approaches also consider the change history of the system under study: Zimmermann et al. place their analysis at the level of entities in a meta-model [124]. Their focus was to provide a mechanism to warn developers that: "Programmers who changed these functions also changed ...". Association analysis was also used in the work of Ying et al. [122] for change predictions. After the detection of undesired evolution patterns the software should be restructured to support evolvability. Several re-engineering patterns have been discovered by Demeyer et al. [28]. The range of software evolution analysis was extended by Bird et al. as they developed a technique to identify the submission and acceptance of software patches in open source projects [10].

In terms of re-engineering activities, Demeyer et al. [28] propose practical guidelines to identify where to start a re-engineering effort: working on the most buggy part first or focusing on the client's most important requirements. This approache is based on information that is outside the code whereas the above mentioned approaches work with the code base.

Additionally, several factors play an important role and affect how systems evolve. In particular, three properties of evolving systems should be mentioned here: *domain of the system, experience of stakeholders*, and *development process covering methodology, technology and organization*. An aspect of the research work done by Cook et al. [21] is the including of internal metrics for complexity, cohesion or coupling to obtain an enriched impression of the quality of a software system. Based on such measurements, the evolvability of a system can be determined and support for re-engineering activities can be provided. Also the risk of a change to break an already existing feature can be assessed by analyzing software changes [84].

Clustering techniques may be used to improve modularity and support evolution. A measure for similarity or dissimilarity between two objects has to be identified to subdivide objects into clusters. Chung-Horng Lung [77] presents examples for utilizing clustering techniques for the improvement of software architecture. He suggests incorporating reverse engineering tools to identify the dependencies among classes.

## 2.2 SOFTWARE QUALITY

Strongly related with the evolvability of software systems is its internal quality. Several factors may play an important role in the evolution of a system. They affect how systems evolve and emphasize the requirements for well-engineered software systems. One of the first quality models was established by McCall et al. [80] that is a category based hierarchical model. The first level is defined by the three perspectives *product revision* as the ability to undergo changes, *product transition* as the ability for adaptability to new environments for the software, and *product*

*operations* describing the external product attributes. Each of these three categories assembles several quality factors:

1. Product revision
    - Maintainability: effort to locate and fix a fault
    - Flexibility: ease of making changes
    - Testability: ease of testing
2. Product transition
    - Portability: effort to transfer software to another environment
    - Reusability: ease of reusing software for a different purpose
    - Interoperability: effort to link the software with another system
3. Product operations
    - Correctness: how software conforms to specifications
    - Reliability: ability not to fail
    - Efficiency: in the context of processor time and storage
    - Integrity: protection from unauthorized access
    - Usability: ease to access functionality

Almost at the same time another quality model was developed by Boehm et al. [12] that exhibits also a hierarchical structure. They focus more on the aspect of measurement and provide attributes and metrics. The first level of their model is comprised of three characteristics: *As-is utility* describing how easy and efficient the system can be used, *maintainability* describing the effort needed for understanding and modifying the system, *portability* describing how the system can be extended to other environments. Figure 2.1 depicts the entire model.

Dromey [32] discards the hierarchical quality model and relates three factors with each other: *Components* are the software entities, *quality attributes*, and *component properties* that influence quality attributes. He defines the following quality carrying properties:

1. Correctness properties
    - Computable
    - Complete
    - Assigned
    - Precise
    - Initialized
    - Progressive

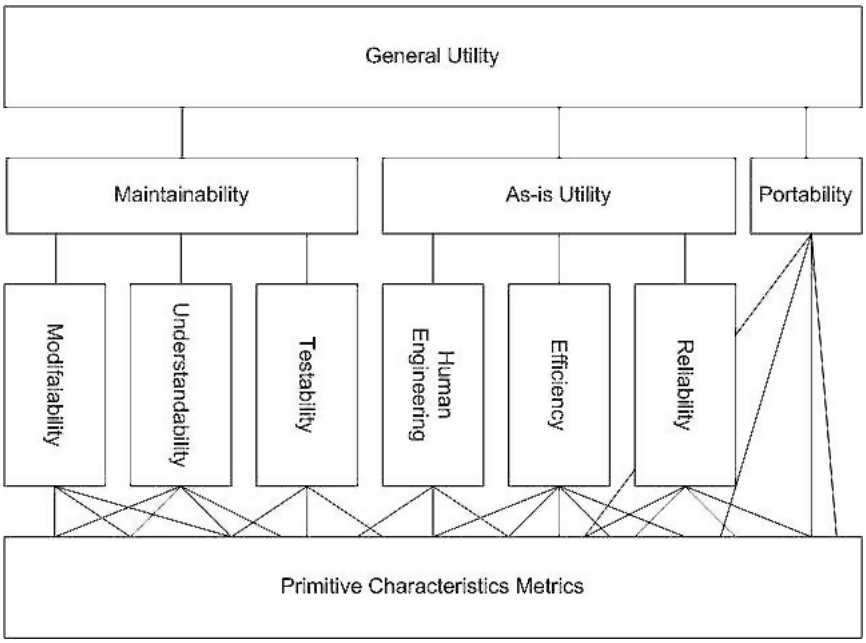



Figure 2.1: Quality model of Boehm et al. [12]

- Variant
- Consistent

2. Structural properties
    - Structured
    - Resolved
    - Homogeneous
    - Effective
    - Non-redundant
    - Direct
    - Adjustable
    - Range-independent
    - Utilized
3. Modularity properties
    - Parameterized
    - Loosely coupled
    - Encapsulated
    - Cohesive

   - Generic
   - Abstract

4. Descriptive properties
   - Specified
   - Documented
   - Self-descriptive

A quality model summarizing all the efforts in a common standard was defined by the International Standards Organization (ISO) as standard ISO 9126 [91]. It contains the following quality factors:

- Functionality
- Reliability
- Usability
- Efficiency
- Changeability
- Portability

Bansiya et al. [6] uses the same categories as ISO 9126 and defines metrics covering the quality carrying properties of Dromey. These metrics can represent the quality categories as numerical values. A master thesis of Jetter [62] focusing on software quality attributes describes more details of quality models.

Software quality assurance accounts for approximately 50% of the development time [51]. Therefore, it is important to improve defect-detection techniques as well as reduce their costs. Bug finding tools can find defects in software source code using an automated static analysis. This automation may be able to reduce the time spent for other testing and review activities. For this we need to have a clear understanding of how the defects found by bug finding tools relate to the defects found by other techniques. There are various techniques to identify critical code pieces. The most common one is to define typical bug patterns that are derived from experience and published common pitfalls in a certain programming language. Wagner et al. [115] analyzed several industrial and development projects with the help of bug finding tools as well as with other types of defect-detection techniques. The main finding is that the bug finding tools predominantly discover different defects than can be found through testing but nevertheless the types that can be detected are analyzed more thoroughly. They additionally have a significant ratio of false positives. Hence, bug finding tools can not substitute dynamic tests or reviews, as they discover significantly more and different defect types.

## 2.3 METRICS

Software metrics provide scalability for analysis approaches and are therefore a widely used technique to establish a broad understanding of software evolution and the development process. Lanza and Marinescu [72] give an overview how metrics can be applied in software engineering, where they deduce many single and combined threshold values from multiple industrial projects. Numerical values are easily comparable, which helps to evaluate a broad range of systems. In the context of software evolution a number of studies investigated the long-term stability of software systems and the ability to cope with new requirements. Different types of measurements are applied to get insights into evolvability.

A number of metrics have been suggested to evaluate diverse kinds of software engineering activity in recent years. Chidamber and Kemerer's metrics [18] are often used to detect faulty classes. Tsantalis et al. [112] try to quantify the change probability of each class in future releases with the help of these metrics. Forward and backward logistic regression was performed with the result, that only some proposed measures such as class size were relevant in the analyzed case studies. Their work identifies classes that have a high probability of change and at the same time can affect a large number of classes.

Different types of measurements are applied to get insights into maintainability. Software metrics are used to establish a broad understanding of the software system and the development process. Numerical values are easily comparable, which helps to evaluate a broad range of systems. On the contrary, the human brain is capable to distinguish multiple aspects of a complex problem based on a graphical visualization. Hence, an interesting approach is to combine the advantages of visualization with the scalability of metrics. A graph can be enriched with metric information concerning the attributes classes. Nodes representing object-oriented entities may be used to provide additional information based on the node size, the node position, and the node color. Demeyer et al. [27] incorporated simple metrics collected from source code into graphical layouts. Different graphs were combined with different metrics, which provided an initial understanding of the studied software and assisted in the unveiling of design anomalies.

In relation to the successes that can be achieved with the help of software metrics, an evaluation is necessary which metrics are useful for which types of problems. Demeyer et al. [26] evaluated a number of existing size and inheritance metrics on three releases of a medium sized object-oriented framework. From framework documentation they deduced that the transition from the first release to the second release was mostly restructuring. On the contrary the functionality was extended from the second to the third release. This restructuring and extension could be confirmed with the help of measurements. During the restructuring phase inheritance hierarchies were reordered that could be identified by investigating the structure of class hierarchies. The number of defined methods was used as another indicator for restructuring activities.

Other metrics such as cohesion and coupling could be incorporated to round up the analysis approaches. Kabaili et al. [63] investigated whether cohesion metrics could also be used as changeability indicators and concluded that this is not the case; at least not for the common cohesion metrics LCC (loose class cohesion [9]) and LCOM (lack of cohesion in methods [76]). Therefore, more suitable cohesion metrics have to be developed. As a result coupling metrics have to be regarded to receive a better understanding of a software system.

Stevens et al. [109], who first introduced coupling in the context of structured development techniques, define coupling as "the measure of the strength of association established by a connection from one module to another". It is supposed that, the stronger the coupling between modules the more difficult these modules are to understand and correct. This results in a more complex software system [14].

One work concerning coupling metrics identified coupling between objects (CBO). CBO is defined in [17] as follows: "CBO for a class is a count of the number of non-inheritance related couples with other classes." An object of a class is coupled to another, if methods of one class use methods or attributes of the other one. In [18], a revised definition is proposed: "CBO for a class is a count of the number of other classes to which it is coupled — this includes coupling due to inheritance."

Many coupling measures are based on source code and the exposed attributes. One of the research approaches base on source code were introduced by Eder et al. [33]. There three different types of coupling were identified:

- Interaction relationships between methods: This type of relationship is caused by message passing.
- Component relationships: Each object has a unique identifier (the object identity). An object may reference another object using the identifier of object. This introduces a component relationship between the two classes.
- Inheritance relationships between classes: Two classes $c$, and $c'$ are inheritance coupled, if one class is an ancestor of the other.

With this classification one can derive different dimensions of coupling which are classified according to different strengths.

Hitz and Montazeri [60] defined two different kinds of coupling, which are determined by the state of an object or an object's implementation. The state of an object is determined by its attributes at a given moment at runtime. The state of an object's implementation is identified by the class interface and body at a given time in the development cycle. From these definitions, they derive two "levels" of coupling:

- Class level coupling (CLC): CLC represents the coupling resulting from state dependencies between two classes in a system during the development life cycle. This consideration is important for maintenance and change dependencies. Changes in one class may lead to changes in other classes which use it.
- Object level coupling (OLC): OLC represents the coupling resulting from state dependencies between two objects during the runtime of a system. The authors claim that OLC is relevant for runtime oriented activities such as testing and debugging.

Instead of couplings that are manifested in the source code we focus on co-change coupling in terms of change patterns [41, 43]. If programs change together across module or subsystem

boundaries, the decomposition structure of the application should be reconsidered and possibly restructured or re-engineered. Our approach attempts to measure coupling based on analysis of multiple releases of a system. This approach is based on observed change behavior of modules in a system and may be categorized as retrospective. Our measures may be used not only as coupling measures to guide restructuring efforts but also to validate the effectiveness of predictive and code-level coupling measures [41, 42, 44].
Coupling and cohesion measures are a way to measure structural cohesiveness of a design. The main purpose is to evaluate how maintainable a design and resulting implementation are, and to guide improvement efforts. The basic idea is that the more dependencies exist among modules the less maintainable the system is because a change in one module will necessitate changes in many dependent modules.

## 2.4 DATA MINING

Metrics provide the benefit that they are usually numeric values and therefore easily comparable. Additionally, large systems can be characterized with a small number of metrics, if the measures of the attributes can be condensed so much. However, when measurements are established in software projects, a large number of metrics is often gathered. This huge data amount renders manual analysis more difficult or makes it even impossible. To cope with this situation we utilize data mining techniques to extract recurring patterns in the data. These techniques enable to answer concrete questions (e.g. how many defects are expected for the near future). Moreover, some resulting models from the data mining (e.g. decision trees) enable the interpretation of the interrelations between the input attributes. Automation is important in the context of the usability of data mining in software projects. To improve this situation Gonnet [54] has integrated the Weka data mining tool [119] with the Eclipse development environment.
Most studies related to decision trees in software engineering use size and complexity metrics of the software system itself to predict reliability during the usage of software [96]. The applied algorithms have sound statistical foundations and the predictions provide reasonable accuracy. The quality of prediction models based on internal software metrics are often improved to obtain better results or simpler prediction models. One way to improve prediction models based on classification trees is the application of Akaike Information Criterion, based on maximum likelihood estimation and least number of complexity metrics [110]. In contrast, we propose using metrics related to the software evolution (e.g. number of authors or number of commit messages) to predict events in the software life cycle.
Software quality is often predicted based on two aspects. First, in the time domain the reliability of software is measured as the probability of failure-free operation for a specific time period under a given environment. Second, in the input domain the reliability of a software system is the probability of failure-free operation for specific input states. Tian [111] integrates these two approaches with tree-based models, where the relationship between the estimated reliability and its input state or timing predictors is available in a tree structure. These models have the advantage that both numerical and non-numerical attributes can be used. The attributes for his integrated reliability analysis are gained from runs of the program.

Classification trees generate partition trees based on a training data set describing known experiences of interest (e.g. characteristics of past software development). The technique of classification trees can handle both continuous and discrete variables and interdependencies between variables are taken into account. Additionally, the tree structure is intuitive and can be easily interpreted. Briand et al. [13] even try to improve the predictive capabilities by combining the expressiveness of classification trees with the rigor of a statistical basis. Their approach called OSR generates a set of patterns relevant to the predicted object. The algorithm groups similar objects and extracts an optimal pattern vector, estimated based on the entropy H.

Khoshgoftaar et al. [68] use software metrics as input to classification trees to predict fault-prone modules. One release provides the training dataset and the subsequent release is used for evaluation purpose. They claim that the resulting model achieved useful accuracy in spite of the very small proportion of fault-prone modules in the system. With the help of statistical tests subsets of modules can be detected with uncertain classifications allowing enhancement strategies to resolve uncertainties. They used historical data for evaluation purposes, but the prediction model is based on conventional software metrics.

### 2.4.1 Sequential Data Mining

Quality models in software engineering rely on condensed metrics but do not consider the course of time. In our research we focus especially on this aspect. According to Geurts [50] the time series classification problem can be defined as follows: Given a universe of objects. Each object is described by a certain number of temporal attributes and classified into one particular class. The goal of the machine learning algorithm is to find a function $f(o)$ that is as close as possible to the true classification $c(o)$. This function should only depend on attribute values.

Kadous solves the problem of multivariate time series classification with the help of parameterized event primitives. The extracted events are clustered to create prototypical events. They are used as the basis for creating more accurate and comprehensible classifiers than hidden Markov Models or recurrent neuronal networks. [64] Manganaris developed a system for supervised classification of univariate signals using piecewise polynomial modeling combined with a scale-space analysis technique (i.e. a technique that allows the system to cope with the problem that patterns occur at different temporal scales) and applies them to space shuttle data as well as to an artificial dataset. [78]

## 2.5 Software Processes

Data mining techniques are applied for several tasks in the software engineering domain. One field for the application of data mining is software defect prediction. Another important field is process mining where patterns of processes are analyzed to investigate if the behavior is as expected. One of the first addressing the problem of discovering process models from event logs were Cook and Wolf [20]. They analyzed three different algorithms that automatically discover models capable to explain all the events in log traces. Even before Garg et al. [46] presented a

meta-process for reuse, process discovery, and evolution. They investigated process history to analyze in the context of process improvement how the process should be changed for the next cycle.
According to [120] process data capture is the activity of obtaining information about an existing software process and process analysis is the manipulation of that information for purposes of problem identification. Wolf and Rosenblum developed a model of the software process and defined a taxonomy for events, which are identifiable, instantaneous milestones in a process. Tools were used to automatically capture event information from the log files of a build tool. This method enables the recording of event data through independent, direct observations. In a study carried out at AT&T the captured build process was analyzed through queries on the event data about the relationships among events, such as dependencies and time intervals to gain a deeper understanding for a following process improvement effort.
In our approach we use the event data captured from software evolution to generate time series, which describe the relationship of different event types. In research focusing on discovering process models the representation used to describe process models are often some kind of networks. Aalst et al. [113] use Petri nets when extracting process models from workflow logs, which contain information about the workflow process as it is actually being executed. The presented $\alpha$-algorithm is able to recover workflow processes.
An extension for the idea of process recovery is introduced by Greco et al. [56] by searching for similarities in different workflow traces to reduce the complexity of the resulting model. They propose a novel framework for the discovering of process models, where they explicitly account for the identification of different variants for the process. A hierarchical clustering technique is used for grouping similar execution traces, where each group is modeled with with a different schema. This increases soundness while preserving maximum completeness. The approach starts with with a preliminary non-disjunctive model that can be obtained with any process mining algorithm from literature and iteratively and incrementally refines the process model to get a monotonic improvement in soundness.
All software systems are produced according to some process model. For many critical systems a defined process model (e.g. Rational Unified Process, V-Model XT, etc.) is a necessary ingredient to build a system of defined quality. The process view is currently very common in industry, which can be observed through the growing importance of quality standards such as ISO 9000 or CMMI. Quality standards enforce a defined process model or at leased the necessity for a definition of the process model to follow. The process model has to be set up compliant with several characteristics laid down by the quality standard. For example the ISO 9000 demands a production process with a planning phase where the system requirements have to be defined, and a development phase where the results have to be verified and validated.
However, projects even under the guidance of quality standards deserve a certain level of control. Process models define how a system ”should be” constructed, but how the system ”is” constructed has a large impact on the success of each project. As a result the project manager and the senior management are interested in the capabilities the projects carried out. For project management it is interesting to know which phases were passed through during the software development or which events took place. This information is necessary in order to take steering actions.

Software process and life cycle models are elaborated in detail in [105], where a software life cycle model is defined as either a descriptive or prescriptive characterization of how software is or should be developed. In contrast, software process models are described to represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution.

## 2.6 REFACTORING

An important aspect of software evolution is the existence of preventive maintenance activities to avoid future problems. These activities are of major interest for our research as a strong relationship between preventive maintenance such as refactoring and software quality is expected. As it is described in [38], refactoring is designed to improve the structure of existing software. It is a technique utilizing behavior-preserving transformations to restructure object-oriented applications. It helps to place code that changes together into one separate place like a class or a particular hierarchy of classes. Refactoring as a technique to reorganize object-oriented constructs has been investigated for quite some years now. The work of Opdyke [90] is one of the first in the area of academic research. It gave inspiration to additional work. Most of the followed studies aimed to automate the application of refactoring on source code (e.g. [19]).

Simon et al. [108] postulate that refactoring should be regarded as a key issue to increase internal software quality. Their approach demonstrates how to use cohesion metrics based on generic distance measures to identify which software elements are closely related and should therefore be assembled in object-oriented entities. Source code inspections are another field where code smells have to be evaluated. jCOSMO was developed to automatically detect code smells such as the ones defined by Fowler [38] and to visualize their distribution in the system [114].

A number of publications show the usefulness of refactoring for software development. Capiluppi et al. [16] found that understandability was increased by refactorings in several projects. Kataoka et al. [65] try to measure the maintainability enhancement effect of program refactorings based on coupling metrics. In a larger experiment the developer's subjective evaluations match the results of the author's approach to determine the effectiveness of refactorings. Najjar et al. [88] investigate refactoring of constructors, because constructors do not ideally communicate the developer's intention, and secondary produce duplicated code. The study investigated several Java systems and found that the code of two systems could be economized. A survey paper, which extensively discusses research in the field of software refactoring was written by Mens and Tourwé [81]. It discusses refactoring activities and numerous techniques. Demeyer et al. [25] validate several metrics for identifying if refactorings took place for a given file. In contrast, we perform a prediction task based on evolution data.

Antoniol et al. [4] analyze the evolution of object-oriented source code at the class level. The focus is on a limited number of refactoring events, which are identified based on a vector space model. The application of the proposed approach to an open source domain name service produced a list of class refactoring operations. Advani et al. [1] analyzed a range of open source systems, regarding whether a refactoring occurred and if so which were the most common. They

found that simple refactorings, like basic operations on fields and methods occurred more frequently, than more interweaved refactorings, such as those, requiring manipulation of the inheritance hierarchy.

In a previous work we investigated the design of a software system based on evolution data, extracted from source code management systems [102]. This information is used to detect architectural shortcomings in the design of the analyzed software and how refactoring can support the evolvability of software systems.

## 2.7 Prediction in Software Engineering

As described in Section 2.3, software metrics provide quantitative information that can be used to measure software projects — an important software quality indicator. Complexity and design quality as well as some other properties of the final software can be predicted by analyzing the data. One question that remains to be answered is, how internal software metrics relate to external software attributes such as software quality, which is most useful but difficult to measure objectively. Our work aims to provide support to answer this question. Other researchers have also addressed this topic from different viewpoints. For example Yu et al. [123] present a case study, where first the distributions and mutual correlations of selected metrics are analyzed and then the effects on software fault-proneness are studied. Two techniques are used: Regression analysis and discriminant analysis. One important result obtained in this study is that coupling within and especially across inheritance hierarchies shows significantly different effects on fault-proneness. A multivariate analysis shows that it is fairly possible in predicting the actual number of faults for each class and more effective in just identifying whether a class contains faults.

Many organizations want to predict the number of defects in software systems, before they are used, to predict the likely quality and maintenance effort. The testing of a program is a costly part of software development. Defect-detection techniques are the mostly used method to assess quality in software. Therefore, we need to understand the relations between costs regarding those techniques. Wagner and Seifert [116] proposed a cost model for defect-detection techniques that uses a reliability model to analyze the future failure behavior of the software and hence to predict the external costs. The other costs are determined based on expert estimations and direct measurements.

An interesting approach was presented by Weiß et al. [118]. They predict the necessary effort to fix a bug based on the comparison to other similar bugs. As input for their predictions they use an issue tracking system (i.e. Jira), which is also used in our approach for the data on defects in our study. In their work, the issue title, which is a one-line summary, and the detailed description of the issue are used to identify similar bug records with the help of the nearest neighbor approach (kNN). In our study we use the EM algorithm for the search of text similarities.

In a previous study the application of machine learning (inductive) technique is tested to the software maintenance process. Shirabad et al. [107] presents an example of an artificial intelligence method that can be used in future maintenance activities. They call it the inductive or machine learning method which belongs to the category of classification learning. This system does not

require a large body of knowledge as it learns from past experience. An induction algorithm is applied to a set of pre-classified training examples of the concept we want to learn e.g. a relation between fields. The output of the induction algorithm is referred to as a classifier, which is also known as model or concept. Each training example is a known or solved case. Once a model or classifier is learned from a set of training examples it can be used to predict the class or outcome of unclassified or unsolved examples. The authors explain how to find out if other files are affected by the changes applied from a software user. The best results are obtained by combination of text-based features with syntactic structures which improves the results and are even more appropriate for real life deployment. The large size and complexity of systems, high staff turnover, poor documentation and the long periods of time these systems must be maintained lead to a lack of knowledge in how to proceed the maintenance of software systems.

Nikora and Munson [89] developed a standard for the enumeration of faults based on the structural characteristics of the MDS software system. This program allows one measuring and predicting faults precisely and accurately. Units are measured proportional to the way that the system changes over time. Changes to the system are visible at the module level (procedures and functions) and therefore the level of granularity is measured. Since the measurements of system structure are collected at the module level, another aim is to collect information about faults at the same granularity. A fault, by definition, is a structural imperfection in a software system that may eventually lead to a defect during the operation of the system. There are different reasons for each fault: Some faults are existing because of errors in the specification of requirements. Others are directly attributable to error committed in the design process. Finally, there are faults that are introduced directly into the source during implementation. The standard of Nikora and Munson and the related fault measurement process was applied to a software system's structural evolution during its development. Every change to the software system was measured and every fault was identified and tracked to a specific line of code.

Only a small number of empirical studies using industrial software systems have been performed and published. Ostrand and Weyuker [92] believe that this is because it is difficult to get access to these large systems and needs a lot of money and time to analyze the data. They evaluated a large inventory tracking system at AT&T. In that case study data was collected whenever a fault is identified in the software system. Furthermore faults are detected how they are distributed over different releases. They discover that faults are always heavily concentrated in a relatively small number of releases during the entire life cycle. Additionally, the number of faults in these releases is getting higher as the product matures and high-fault releases tend to remain faulty in later releases. So it would be worthwhile to concentrate fault detection only in a relatively small number of high fault-prone releases, if they can be identified early.

Another approach using software histories for defect prediction presented by Kim et al. [69] does not focus on metrics but tries to identify entities that are in the locality of other bugs (or bug fixes). They exploit temporal and spatial locality and keep the information in a bug cache to predict locations where defect inducing changes took place. The approach is based on the idea that failure inducing changes are related with each other. When bug fixes take place they use this information to identify failure inducing changes. From this information they deduce other elements that have been involved or related with these changes and are therefore also suspicious

to contain defects. As this approach provides impressive results, it focuses on one of the aspects related with defects. Fenton and Neil [34] provide a critical review of literature that describes several software metrics and a wide range of prediction models with the purpose to predict the likely quality and maintenance of a software product. He found out that most of the statistical models are based on size and complexity metrics within the aim to predict the number of defects a system. Others are based on testing a testing process, the ”quality” of the development process, or take a multivariate approach. However, there are a number of fundamental serious theoretical and methodological problems in many studies. As a result, they recommend holistic models for software defect prediction using Bayesian Belief Networks, as alternative approaches to the single-issue models used in the past. Therefore, we use measures covering multiple aspects in our prediction approaches.

By focusing on the types of changes, costs and efforts to evolve, Kemerer and Slaughter [67] suggest that future trends within a particular system are predictable. Within their research they collected, coded, and analyzed more then 25000 change events of 23 commercial software systems over a 20-year period. Two of these systems were compared to show the efficacy of flexible phase mapping and gamma sequence analytic methods originally developed in social psychology to examine group problem solving processes. With this study the authors wanted to identify and understand the phases of these techniques through which a software system travels as it evolves over time.

# Chapter 3

# Measures of Software Evolution

The availability of a broad range of metrics supports an in-depth analysis of software projects. The current chapter focuses on the definition of multiple evolution metrics, where different aspects of software projects are taken into account. We define 63 metrics to cover different topics and according to the guidance of Fenton and Neil [34] we organized our evolution metrics into the following categories: size, process orientation, defect discovery, team, complexity of existing, difficulty of problem, relational aspects, and time constraints.

## 3.1 Preparing Evolution Data

Development projects usually maintain a lot of information concerning the evolution of the system. Project managers have to be able to observe the status of single tasks as well as the progress of the entire project. Developers need information about what is requested from them and need storage systems for their results. Thus, different aspects are covered by different systems, which we have to integrate for our analysis. To mine software development projects we use versioning systems (e.g. CVS) and issue tracking systems (e.g. Jira).

Versioning systems enable the handling of different versions of files in cooperating teams. These tools log every change, which provides the necessary information about the history of a software system. The log-information for our mining approach—pure textual, human readable information—is retrieved via standard command line tools, parsed and stored in a relational database [35].

Issue tracking systems manage data about project issues such as bug reports and feature requests. These systems provide a view of software products on project level and give a chronological overview of the requirements and their implementation. We extract the data from such systems based on their backup facilities, where the entire issue data can be exported into XML files. These files are processed to import the information into our evolution database.

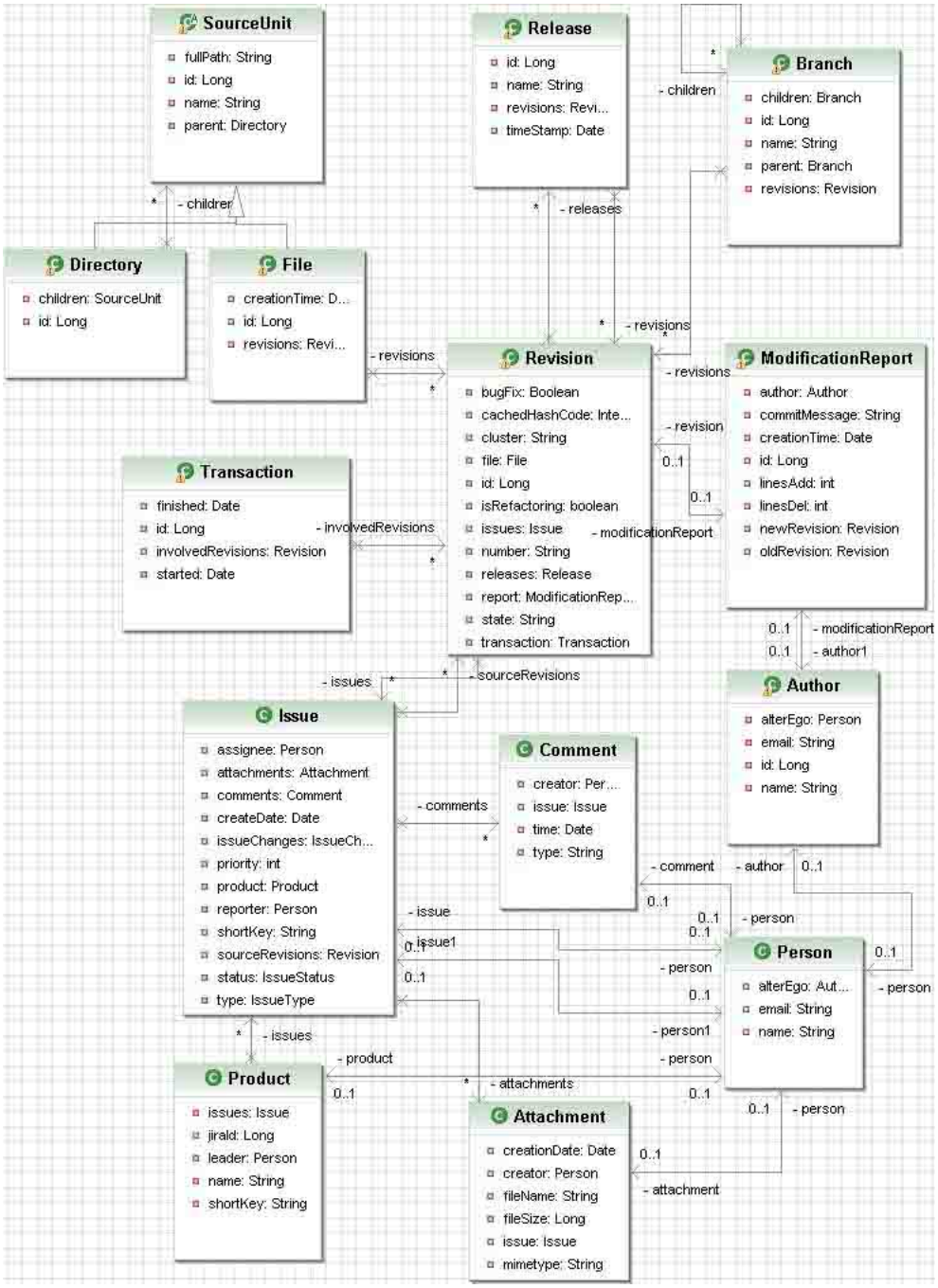



Figure 3.1: Evolution Data Model

### 3.1.1 DATA ELEMENTS AND TRANSACTIONS

Figure 3.1 describes the data model of the evolution database. The upper part shows the entities containing data from versioning systems. Revisions are located in the center and are strongly related to modification reports (MR). Both provide historical data about files such as the change date, the size of the change measured on the basis of lines, and the author of the change. Revisions are related with each other as the code of one revision replaces the code of the previous one. This information is maintained by modification reports in our model. Releases are defined as collections of revisions of all files maintained by the particular versioning system. Additionally to this grouping of revisions, they are related to each other based on commits of developers. When a developer checks in changes based on several files at once, the versioning system only stores the dates of the new revisions for each particular file, but does not maintain the transaction information that the files where commited together. As a result, we have to reconstruct commit transactions (also called change sets) identifying the files that were changed together in a post processing phase.

Transactions $T_n$ are sets of files that were checked-in into the versioning system by a single author with equal commit messages within a short time-frame—typically a few minutes. To capture entire transactions possibly lasting several minutes we use a dynamic time adaptation approach. Each transaction is initially set to last for 60 seconds. The revisions of different files with equal author and commit message within the time window of the transaction is added to this particular transaction. The window is expanded on each detected revision to last for 60 seconds after the time of the last revision related to this transaction. Transactions are then used in the evaluation of couplings between software entities. We refer to a co-change coupling as: *Two entities (e.g. files) are co-change coupled, if modifications to the implementation affected both entities over a significant number of commit transactions*. The strength of the co-change coupling between two entities $a$, $b$ can be determined easily by counting all common transactions of $a$ and $b$, i.e., $C = \{\langle a, b\rangle | a, b \in T_n\}$ is the set of co-change coupling and $|C|$ is the strength of the coupling of these two software entities. [43]

We compute the change couplings as follows:

1. **Extract all software entities of interest**: The couplings are generated based on classes. If the user selects modules as a point of interest, all classes within this module and all its descendant submodules have to be considered for the determination of the couplings:

   $$E = \{e | e \in M, d(M)\}$$

   $M$ is set of selected entities (i.e. modules and classes) and
   $d(M)$ are all descendants of the selected modules

2. **Detect transactions within the give time period**: Based on a given time window $T$ for which the coupling of the selected entities should be computed, we have to identify the commit transactions $ct$ that happened in the given time period (i.e. the time $t(ct)$ of the commit transaction is contained in the given time window $T$).

   $$CT = \{ct | t(ct) \in T\}$$

$t(ct)$ is the time of the commit transaction $ct$ and
$T$ is the given time period

3. **Get all revisions of the discovered commit transactions**: To calculate the co-change coupling for a group of entities (i.e. classes) we need to detect the revisions $r$ of related classes that have been changed together with the entities of interest (i.e. $E$) within the defined time frame:

$$R = \{r | ct(r) \in CT, (e(ct(r)) \cap E) = \emptyset\}$$

$ct(r)$ is the commit transaction of revisions $r$ and
$e(ct(r))$ is the set of classes related by the commit transaction

4. **Extract coupling for classes of interest**: We are looking for pairs of classes that represent coupled software entities, where at least one class is an element of the selected modules or classes:

$$C = \{c =< e_1, e_2 > | (e_1 = e(r_1) \ \vee \ e_2 = e(r_2)) \in E, ct(r_1) = ct(r_2), \ r_1, r_2 \in R\}$$

$e(r)$ is the class of the revision $r$.
$|C|$ is defined as the strength of the coupling.

### 3.1.2 Relating Data Sources

The data items from versioning systems are related with issue tracking data in two dimensions. The first one relates revisions of file histories with issue objects based on the commit message of revisions. Usually when developers commit changes to a versioning system they have to specify which issue is addressed by these changes. For our analysis the issue key is extracted from the commit message and the database is searched for existing issues with the given key. If a matching issue is found the date of the issue is compared to the date of the revision. The relation between the revision and the issue are only accepted if the issue has been created before the date of the revision, because developers can only solve issues that are known at the time of development. For example in Jira a key of an issue may be ”PJVI-1482”. When a string with such a pattern (i.e. PJVI-[1-9]*) is found in a commit message of a file, the appropriate issue is searched in the database and the two entities are related with each other.

Additionally, versioning and issue tracking systems are related with each other based on the accounts (i.e. IDs) of people working on the project. The accounts of CVS and Jira are related with each other when the account names have a similarity of more than 3 characters. With the mapping between the accounts it is possible to divide the issues in different groups: Issues that are created by developers and issues that are created by customers and customer related staff such as service personnel and project managers.

| Name | Description |
|---|---|
| linesAdded | number of added lines<br>normalized on the total lines of code (LOC) |
| linesModified | number of modified lines<br>normalized on the total lines of code (LOC) |
| linesDeleted | number of deleted lines<br>normalized on the total lines of code (LOC) |
| linesType | indicating relation of added to modified lines<br>negative if more lines added otherwise positive |
| largeChanges | number of changes with added lines >= 2*average<br>normalized on the total number of changes |
| smallChanges | number of changes with added lines <= average/2<br>normalized on the total number of changes |

Table 3.1: Size related evolution metrics.

## 3.2 Evolution Metrics

From the basic data model we derive 63 evolution metrics, which are gathered on a file basis, where data from all revisions of a file within a given time period (i.e. the training period) is summarized. To build balanced assessment models we create multiple metrics to represent several important aspects of software development such as the complexity of the designed solution, process used for development, interrelation of classes, etc. As previous studies (e.g. [70, 86]) found out that relative values provide better performance in prediction than absolute ones, we decide that all our evolution metrics have to be relative measures. We set up the following categories of metrics for each file containing changes within the given time period:

### 3.2.1 Size

Table 3.1 provides an overview of our size related metrics. This category groups ”classical” metrics such as lines of code, but from an evolution perspective: *linesAdded*, *linesModified*, or *linesDeleted* relative to the total LOC of a file. The last metrics measures a certain aspect of the discipline of the developers, where developers remove code that they regard as unnecessary to keep the system simple. Other metrics of this category are *linesType*, which describes if there are more lines added or lines modified. Additionally, we use *largeChanges* that describes changes, which have more than double of the LOC of the average change size and *smallChanges*, which indicates changes with less than half of the average LOC. We expect that the last one is an important metrics, as other studies have found out that small modules are more defect prone than large ones (e.g. [59, 85]).

| Name | Description |
|---|---|
| authorCount | number of authors working on file normalized on the number of changes |
| authorSwitches | number of hands over between authors normalized on the number of authors |
| peopleCount | number of assignees of the related issue normalized on the number of authors |
| peopleSwitches | number of re-assignments of issue normalized on hands over between authors |

Table 3.2: Team related evolution metrics.

### 3.2.2 TEAM

As we see in Table 3.2 this category contains several people related metrics. The number of authors working on a common piece of software influences the way software is developed. We expect that the more authors are working on the changes the higher is the possibility of rework and mistakes. In this context D'Ambros et al. [23] presented an analysis of developer effort. We define a metric for the *authorCount* regarding the number of authors in terms of CVS relative to the number of changes. Further, the interrelation in the work of developers is interesting. We investigate work rotation between the authors involved in the changes of each file as *authorSwitches*. The number of people assigned to an issue (*peopleCount*) is put in relation to the number of authors contributing to the implementation of this issue.

### 3.2.3 PROCESS ORIENTATION

In this category (see Table 3.3) we assemble metrics that define how disciplined individuals follow software development processes and which effects this has on their development work. On source code changes developers have to include the issue number in their commit message to the versioning system. We define a metric regarding *issueCount* relative to the number of changes. At least the developer should provide some rational for the implemented changes in the commit message. Thus, we use *withNoMessage* measuring the number of changes without any commit comment.

In each project the distribution between different priorities of issues should be balanced. Usually, the number of issues with highest priority is very low. A high value may indicate problems in the project that have effects on quality and re-work amount. Accordingly, we investigate *high-PriorityIssues* and *middlePriorityIssues* relative to the total number of issues. Also the time to close certain classes of issues provides interesting input for project assessment and we use *avg-DaysHighPriorityIssues* and *avgDaysMiddlePriorityIssues* in relation to the average number of days that are necessary to close an issue. To get estimation for the work habits of the developers, we inspect the number of *addingChanges* and *modifyingChanges* per file.

| Name | Description |
|---|---|
| issueCount | number of issues<br>normalized on the number of changes |
| withNoMessage | number of changes without even a comment<br>normalized on the number of changes |
| highPriorityIssues | number of issues with priority high<br>normalized on the number of issues |
| middlePriorityIssues | number of issues with average priority<br>normalized on the number of issues |
| avgDaysHighPriorityIssues | number of days to close high priority<br>normalized on the average days for closure |
| avgDaysMiddlePriorityIssues | number of days to close middle priority<br>normalized on the average days for closure |
| addingChanges | number changes: lines added $>$ changed<br>normalized on the number of changes |
| modifyingChanges | number changes lines added $<$ changed<br>normalized on the number of changes |

Table 3.3: Process oriented evolution metrics.

### 3.2.4 COMPLEXITY OF EXISTING SOLUTION

Table 3.4 summarizes our evolution metrics related to the complexity of the existing solution. We expect that according to the laws of software evolution [73], software continuously becomes more complex. Changes are more difficult to add, as the software is more difficult to understand and the contracts between existing parts have to be retained. As a result, we investigate the *changeCount* of the training period in relation to the number of changes during the entire history of each file. The *changeActivityRate* is defined as the number of changes during the entire lifetime of the file relative to the number of months measuring the entire lifetime. The *linesActivityRate* describes the number of lines of code of the file relative to the age of the file in months.

We approximate the quality of the existing solution by the *bugfixCountBefore*, which describes the number of bug fixes implemented before our training period. Again we use relative values and compute the division of the number of bug fixes before and the number of changes. We expect that the higher the fix rate is before a given period the more difficult it is to establish a better quality later on. The *bugfixCount* is used as well as *bugfixLinesAdded*, *bugfixLinesModified*, and *bugfixLinesDeleted* in relation to the base values such as the number of lines of code added, modified, and deleted for this file. Usually, for bug fixes not much new code should be necessary, as most code is added for new requirements. Therefore, *linesAddPerBugfix*, *linesModifiedPerBugfix*, and *linesDeletedPerBugfix* are potential indicators to reveal the complexity of the existing solution.

| Name | Description |
|---|---|
| changeCount | number of changes in training period<br>normalized on the total number during lifespan of the file |
| changeActivityRate | total number of changes during lifespan<br>normalized on the number of months of the lifespan |
| linesActivityRate | total number of lines of code<br>normalized on the age of the file measured in months |
| bugfixCountBefore | number of bugs before the training period<br>normalized on the number of changes of any type |
| bugfixCount | number of bug fixes<br>normalized on the overall number of changes |
| bugfixLinesAdded | number of lines added within bug fixes<br>normalized on the overall number of added lines |
| bugfixLinesModified | number of lines changed within bug fixes<br>normalized on the overall number of changed lines |
| bugfixLinesDeleted | number of lines deleted within bug fixes<br>normalized on the overall number of deleted lines |
| linesAddPerBugfix | number of lines added within bug fixes<br>normalized on the number of bug fixes |
| linesModifiedPerBugfix | number of lines changed within bug fixes<br>normalized on the number of bug fixes |
| linesDeletedPerBugfix | number of lines deleted within bug fixes<br>normalized on the number of bug fixes |

Table 3.4: Evolution metrics focusing on the complexity of the existing solution.

| Name | Description |
|---|---|
| coChangeNewFiles | number of new files introduced with changes of this file normalized on the number of co-changed files |
| issueAttachments | number of attachments of the related issues normalized on the number of issues |

Table 3.5: Evolution metrics focusing on the difficulty of the problem.

### 3.2.5 DIFFICULTY OF PROBLEM

Table 3.5 shows our metrics related to the difficulty of the existing solution. We expect that new classes are added to object-oriented systems when new features and new requirements have to be satisfied. We use the information whether together with changes to the file of interest other files were newly introduced. To measure how often a file was involved during the development with the introduction of other new files we use *coChangeNewFiles* as an indicator. Co-changed files, which are files that have co-change coupling with the particular file of interest, are identified as described in Section 3.1.1.

The amount of information necessary to describe a requirement is also an important source of information. The metric *issueAttachments* identifies the number of attachments per issue.

### 3.2.6 RELATIONAL ASPECTS

Table 3.6 provides an overview of the evolution metrics for relational aspects. In object-oriented systems the relationship between classes is an important information. We use the co-change coupling between files to estimate their relationship (see section 3.1.1). The first metric of this category is *couplingChanges*, which describes the number of changes/revisions where a group of files were commited in one transaction. We use the number of co-changed files relative to the change count as metric *cochangedFiles*, which counts the number of other files that where changed together with a particular file. Thus, several co-changed files may be related to one *couplingChange*.

Additionally, we quantify co-changed couplings with metrics based on commit transactions similar to the size metrics for individual files: *tLinesAdded*, *tLinesModified*, and *tLinesDeleted* relative to lines of code added, modified, and deleted. The *tLinesType* describes if the transactions contained more lines added or lines modified. *tChangeType* is a coarser grained metric that describes if this file was part of transactions with more adding revisions or more modifying revisions.

For file relations we also use bug fix related metrics: *tLinesAddedPerBugfix* and *tLinesChangedPerBugfix* are two representatives. Additionally, we use *tBugfixLinesAdded*, *tBugfixLinesModified*, and *tBugfixLinesDeleted* relative to the *linesAdded*, *linesModified*, and *linesDeleted*.

| Name | Description |
|---|---|
| couplingChanges | number of changes together with other files<br>normalized on the overall number of changes |
| cochangedFiles | number of co-changed files<br>normalized on the number of changes |
| tLinesAdded | number of added lines in all files of the commit transactions<br>normalized on the number of added lines in the particular file |
| tLinesModified | number of changed lines in all files of the commit transactions<br>normalized on the number of changed lines in the particular file |
| tLinesDeleted | number of deleted lines in all files of the commit transactions<br>normalized on the number of deleted lines in the particular file |
| tLinesType | indicating relation of added to modified lines of the transactions<br>negative if more lines added otherwise positive |
| tChangeType | indicating if this file was part of more adding or changing transact.<br>negative if more adding transactions otherwise positive |
| tLinesAddedPerBugfix | corresponding to linesAddedPerBugfix just for transactions |
| tLinesChangedPerBugfix | corresponding to linesModifiedPerBugfix for transactions |
| tBugfixLinesAdded | corresponding to bugfixLinesAdded regarding transactions |
| tBugfixLinesModified | corresponding to bugfixLinesModified regarding transactions |
| tBugfixLinesDeleted | corresponding to bugfixLinesDeleted regarding transactions |

Table 3.6: Evolution metrics focusing on relational aspects.

| Name | Description |
|---|---|
| avgDaysBetweenChanges | number of days<br>relative to the number of changes |
| avgDaysPerLine | number of days<br>relative to the number of lines added + changed |
| relativePeakMonth | (sequence number of month with most changes)<br>/ (number of months) |
| peakChangeCount | number of changes in the peak month<br>normalized on the overall number of changes |
| changeActivityRate | number of changes<br>relative to the number of months in the period |
| linesActivityRate | number of lines added + changed<br>relative to the number of months |

Table 3.7: Evolution metrics focusing on time constraints.

| Name | Description |
|---|---|
| bugfixesDiscoveredByDeveloper | number of bugs reported by developer (before the training period) normalized on the overall number of bugs (before the training period) |

Table 3.8: Evolution metrics focusing on time constraints.

### 3.2.7 TIME CONSTRAINTS

Table 3.7 shows our evolution metrics related to time constraints. As software processes stress the necessity of certain activities and artifacts, we consider the time constraints to be important for software predictions. The *avgDaysBetweenChanges* metric is defined as the average number of days between revisions. The number of days per line of code added or changed captured as *avgDaysPerLine*.

Peaks and outliers have been shown to give interesting events in software projects [43]. We measure for *relativePeakMonth* the location of the peak month, which contains most revisions within the prediction period. The *peakChangeCount* metric describes the number of changes happening during the peak month normalized on the overall number of changes. The number of changes is measured based on the months in the training period with *changeActivityRate*. For more fine-grained data the lines of code added and changed relative to the number of months is regarded for *linesActivityRate*.

### 3.2.8 DEFECT DISCOVERY

We use data (see Table 3.8) on the discovery of defects as an input to our assessment models, because they allow estimating the remaining defect number. The number of bug fixes initiated by the developers itself provides insight into the quality attentiveness of the team and are covered by metric *bugfixesDiscoveredByDeveloper*.

With the definition of the evolution metrics we are able to predict events in software projects such as defects and refactorings.

## 3.3 RÉSUMÉ

Many assessment approaches in software engineering rely on some kind of measurements. In our case we defined a large number of evolution metrics that provide information about the evolution of the analyzed system. This is the foundation of the assessment models of sPACE and provides the null hypothesis for the following experiments.

H0: *Information about software evolution can be provided in a form that is suitable for software project assessment.* We described how data items from sources such as software repositories (i.e. CVS) and issue tracking systems (i.e. Jira) can be extracted and processed. Additionally, we

show how the data items from the two data sources (i.e. CVS and Jira) can be combined with each other. This is done on two levels:

- The accounts of the two information systems are mapped on each other based on similarities of the account names. This allows one to investigate who actually worked on a file of interest and if this person was also the one that got the issue describing the requirements for the change assigned. If work on an issue is handed over from one developer to the other, is this also reflected in the modifications to the source files? Additionally, we can identify how many bug related issues are reported by developers in contrast to internal testers or external customers. A research on the relationship of individuals working on different IT systems was also done by Bird et al. [11], where they investigated the activities in mailing lists compared with the ones on versioning systems. A human being can have several accounts (e.g. several mailing addresses and several CVS accounts in different projects). In the case of mailing lists one individual may access the system using different accounts and therefore it is necessary to recover the different aliases for one person. For CVS (and also Jira) access and accounts are usually controlled centrally for each project and no alias analysis is necessary.

- The major information items (i.e. revisions in CVS and issues in Jira) are related to each other based on textual references. Therefore, the commit messages from CVS are parsed to identify possible identification strings relating to Jira. Then the Jira issues with the discovered IDs are compared by date with the revisions. Only if the revision is established after the creation of the issue then the relationship between these two information items is accepted. With the help of this link we can get more insight into the reason for the change. For example we can identify the changes that should resolve a reported bug, which provides the basis for defect prediction models.

The major research question of Chapter 1.2 that is addressed by this chapter is Q1: *How to set up metrics from sources such as modification reports and process/project management tools?* After establishing the relationships between different data items, it is possible to focus on evolution attributes covering different aspects of software projects. Based on the categorization of Fenton and Neil [34] we divide our evolution metrics into different categories: size, process orientation, testing, team, complexity of existing, difficulty of problem, relational aspects, and time constraints.

This leads us to another research question Q5: *How to set up effective models that result in prediction with high accuracy?* With the help of the large number of evolution measures and the grouping we can identify which aspects are important for defect prediction, which enables an understanding of the influences of different elements of software projects on software quality.

# Chapter 4

# Design Assessment based on Co-change Coupling

This chapter provides a motivating example on how information about software evolution may be used for the assessment of software projects. In particular the co-change coupling between classes in an object-oriented system are utilized to identify locations within the software system that seem to be difficult to evolve and demand re-engineering activities. For an in-depth analysis of a software and to discover potential shortcomings we utilize EvoLens, a tool we have implemented that enables the visualization of co-change coupling and the navigation through several dimensions such as time and structure. To evaluate our approach we apply this technique on a commercial software system and improve the evolvability of the system with the help of refactoring.
Making evolutionary aspects explicit via visual representations can help the engineer to focus on particular software parts to identify design erosion that has occurred over the past releases. Although many tools exist that provide zooming-in and -out within the hierarchical decomposition of a software system, only our visualization allows an engineer to view a system through a kind of lens view. Our approach called EvoLens is a visualization approach for explorations of evolution data across multiple dimensions. EvoLens is based on temporal lens views a technique similar to fisheye-views [40]. But the graphical representation of this visualization integrates enhanced zooming by navigating through software hierarchies with arbitrary selectable groups of software parts across module or package boundaries. EvoLens allows an engineer to define a focal point for the lens view and navigate along the time dimension by user-defined sliding time windows. The comprehension is supported by using color for metrics of classes.

## 4.1 TOOL SUPPORT

We implemented a tool for the visualization of evolution data such as co-change couplings, which is derived trough a technique similar to association mining. According to Zimmermann et al. [124] "an *association rule r* is a pair $(x_1, x_2)$ of two disjoint entity sets $x_1$ and $x_2$." Our definition of co-change coupling (see Chapter 3) reduces the entity sets $x_1$ and $x_2$ to single entities $e_1$

and $e_2$. A detailed description how co-change coupling is derived can be found in Chapter 3. Our tool, called EvoLens, parses log files of CVS and calculates co-change couplings between classes based on their common change behavior. The couplings are then visualized together with structural information. Our EvoLens tool provides the capability to navigate easily through structure and time. For every selectable software part and every period in the system's history, EvoLens can show the internal and external couplings of the system. Additionally, growth metrics of classes are also visualized with EvoLens to help assessing the necessity of re-engineering.

### 4.1.1 VISUALIZATION

We aim to support the software engineers when they have to learn and understand large legacy software. As legacy systems are often sporadically documented, we incorporate evolution data as another source of information into the reverse engineering process. With the help of EvoLens the developer obtains deeper insight into the evolution and maintenance process of the analyzed software.

#### NESTED GRAPHS

The extracted dependencies between programming entities like classes are depicted by utilizing nested graphs. Nodes, which describe classes, are connected with each other based on the co-change coupling. The thickness of edges between classes represents the strength of the relationship. Relationships between classes in respective parts are defined as *internal coupling* e.g. the relations between classes of a single module and its submodules. The connections between classes within this module and any other part of the software are considered as *external couplings*.

As software elements are grouped to build up nested graphs, the couplings between modules can be analyzed. While examining the system for common change patterns the attention is drawn towards the modularity of applications. Maintainability can be improved when the system is well composed of self-contained components. An ideal situation would allow changing each component independently of others. If changes must be propagated, the smallest possible set of components should be involved. As a result we organize the graph according to the module structure of the application. The resulting graph groups files, which contain classes according to their package membership.

#### LENS VIEW

Classes in software projects are organized in a hierarchical tree structure such as a file in a file system. As we want to visualize couplings between classes that are organized in modules, it would be very complex to display them in a tree representation. Therefore, we selected the nested graph shown in Figure 4.1. For the visualization we use lens-views, which are based on nested graphs with a focal point. We apply a visualization based on focus + context.

The focus is defined by a selection of a number of classes or modules and the lens-views show the co-change coupling for the selected software entities. However, the graph could become

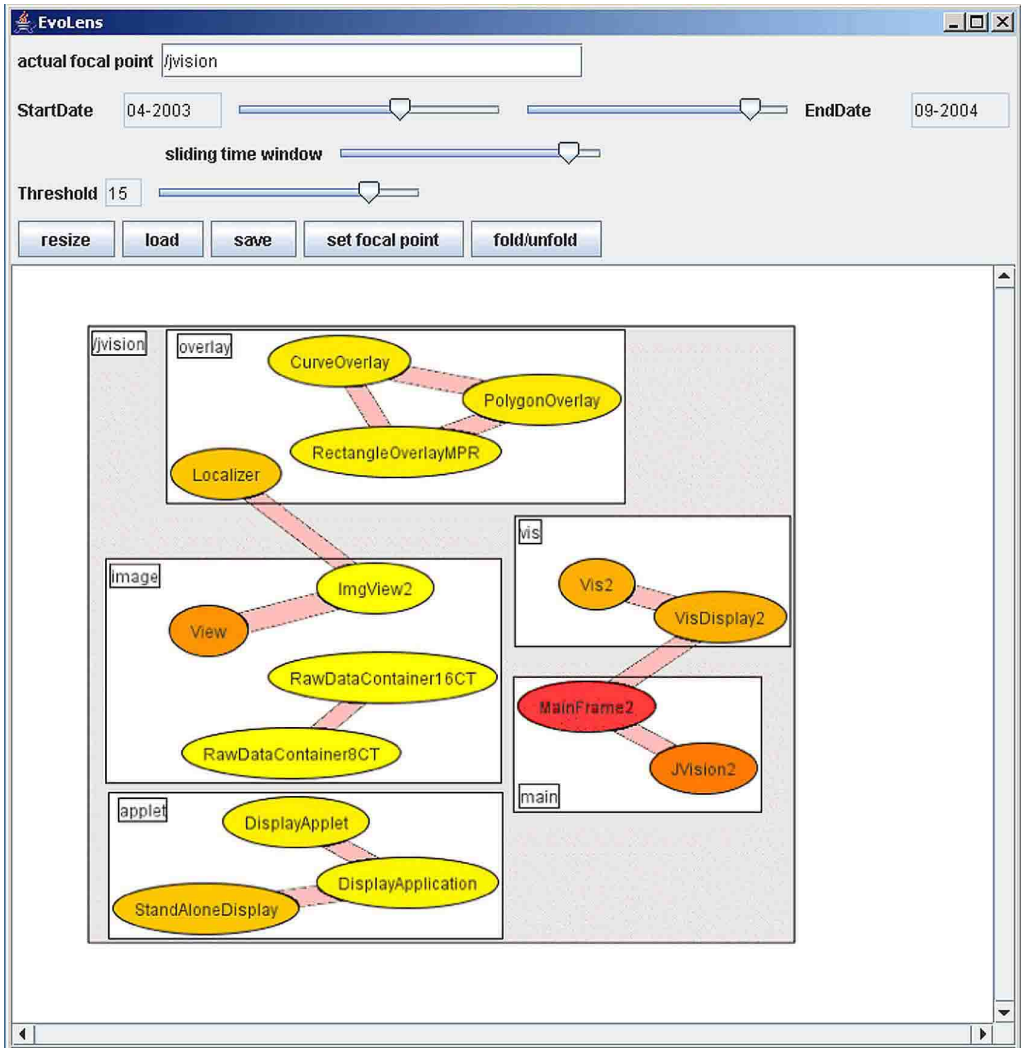



Figure 4.1: The EvoLens tool displaying the evolution of module *jvision*

very large, when the selected entities have a lot of couplings. Therefore, we filter based on the strength of the coupling and display only the ones above a threshold, which can be adjusted by the user of EvoLens. Additionally, we limit the nesting levels by two. Thus only a module and their submodules are displayed through the lens-views. The classes below the second level (i.e. the ones of the sub-submodule or even lower) are projected on the second level of sub-modules, if their coupling is strong enough to be filtered for visualization. Thus, the depicted submodules contain all classes of their own, their sub-submodules and so on. The main focus of the coupling analysis relies on classes. These are the smallest units in the design and architecture of software and the quality of software can be measured on the coupling between these units. Therefore, we avoid too many levels of the hierarchy, which may overload the graph because of the density of evolution data, where some modules contain more than 10 levels of submodules.

We show external couplings of the selected elements to other parts of the software system. These provide the context of our visualization to get a detailed picture of the point of interest. We reduce the context to a small subset of elements to prevent overloading of the visualization. The context based on the visualization of the co-change coupling is limited to the ones that have direct coupling to the selected entities. Transitive couplings are ignored (i.e. if $a$ is a selected entity with coupling to $b$, then $a$, $b$, and their coupling are displayed, but the coupling of $b$ to any other entity that was not selected is not displayed).

In earlier works, researchers have presented views for hierarchical-organized, two-dimensional graphs (e.g. [31]). However, our nesting level and the handling of leaf nodes differ as the nested graph does not depict the entire module hierarchy. The gray box in Figure 4.1 represents the focal point for the nested graph visualization. This focal point is usually represented by a module, which is selected by the user. Within this module the submodules are incorporated as rectangles and classes are represented as ellipses. The connections between the ellipses depict the co-change coupling between the related classes.

### 4.1.2 A Case Study with EvoLens

This section describes the visualizations of EvoLens, and provides some examples. The screen shots are taken from the industrial case study, a Picture Archiving and Communication System (PACS) in the health care domain (see Section 1.2).

#### Color indicating Class Metrics

We use color to indicate evolution metrics of single classes. The color scale from a light to a quite intensive color allows us distinguishing the growth both in color visualization and in gray scale printing. Therefore, we are mapping growth metrics of classes to a color scale from light yellow to dark red. Therefore, we are mapping growth metrics of classes to a color scale from light yellow to dark red (, , ..., ). When a class has a low growth value for the selected time window, it is colored light yellow. The higher the growth value becomes the more intensive red is the represented class colored. The advantage is that growth measures are visualized together with system structure and coupling data. Additionally, numerical values are mapped to colors, so that they can be compared visually instead of reading the values.

The coloring of Figure 4.1 provides hints about what happened during this time.In contrast to *MainFrame2*, which grew much within the first half of the selected time window, *VisDisplay2* has many more lines of code at the end of the observation period than in the middle. The strong growth of classes is an alarming sign, because strong couplings of large classes make it difficult to replace them or to decompose them into smaller pieces. The relationship of *View* and *ImgView2* is also striking. Although *View* grew by more than 300 lines of code, *ImgView2* remained almost constant at 150 lines of code. This information is obtained from the coloring of the classes. These two classes have strong coupling, nevertheless only one grows whereas the other remains equal in size and is continuously modified. Therefore, the continuously changing class desires a clean interface, to improve its relationship to other classes.

### MULTI-DIMENSIONAL VISUALIZATION (STRUCTURE AND TIME)

EvoLens integrates different dimensions of information:

- *Time* on the basis of version history data.
- *Hierarchies* on the basis of software structure
- *Co-change coupling* as representation of common change history

We use a combination of structural information enriched with evolution data to provide better understanding of the development process of large software.

Based on nested graph representations of module structures we visualize evolution data. In EvoLens this coupling information is interweaved with the hierarchy information of the module structure. This module is shown as surrounding rectangle. Edges of different thickness connecting ellipses, which represent classes, give an impression on the common change patterns.

Figure 4.1 shows an example of coupling visualization based on the industrial case study PACS. The focal point of this figure is on the —in Java package— module *jvision*. This module is further divided into submodules. All classes displayed for package *jvision* are included in its submodules. Thus, the user has an impression of how the system is structured, how many modules are related with each other, and if the modules are further divided into smaller units.

In addition to this structural information, the image describes the evolution of the software. It shows which parts were changed at the same time by the programmers of the development team. This provides hints for further maintenance. This coupling describes implications like: ”If class *MainFrame2* has to be changed, then consider also treating other classes like *JVision2* or *VisDisplay2*. This has to be done because they have typically been changed together.”

### FOLDED GROSS STRUCTURE

Often it is important to get a coarse-grained picture of large software systems. Then, not the couplings between classes are of main interest, but the ones between entire modules. In EvoLens such folded views as depicted in Figure 4.2 provide a good general map of the system with

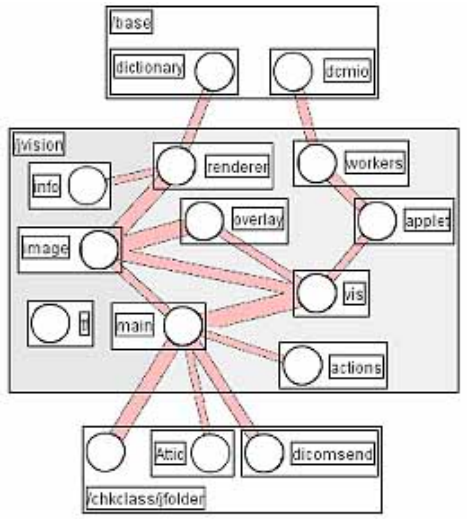



Figure 4.2: Folded Gross Structure of module *jvision* describing module dependencies

*jvision* as focal point. We draw internal couplings and also couplings to other external modules. For external modules, which make up the context in our focus+context visualization, we limit couplings to the ones with the focused module. As this kind of visualization of nested graphs gets often crowded the gross structure only shows the couplings between modules. Classes are not displayed, but instead all classes are denoted through empty circles within modules.

Users may be interested in the individual classes that take part in the overall coupling. In that case they can unfold the graph. Every submodule and every module can be unfolded on its own. For example, it is possible to unfold the focal point to see all the internal couplings but leave the external modules folded. The user may decide on the balance between the clarity of the image and the detail of the class level.

### SELECTIVE COUPLING

Since module boundaries are sometimes too restrictive for in-depth inspections, we decided to incorporate the visualization of individually selected sets of classes. The user can choose some classes during the inspection of the software and let EvoLens show the change relationships for this selected set of modules and classes. [1]

For Figure 4.3 we select four classes: *MainFrame2*, *VisDisplay2*, *ImgView2*, and *Localizer* of Figure 4.1. These classes are the ones that build up the coupling between submodules. In Figure 4.3 all non-selected modules are folded. Figure 4.3 shows that the four selected classes have coupling with some non-selected classes of *jvision*. Furthermore *MainFrame2* has weak coupling with classes of module *chkclass*. With the help of this feature the user can select classes from all over the software system and inspect their coupling.

[1]The selection is accomplished through simple holding the ”ctrl” key and clicking on the desired elements.

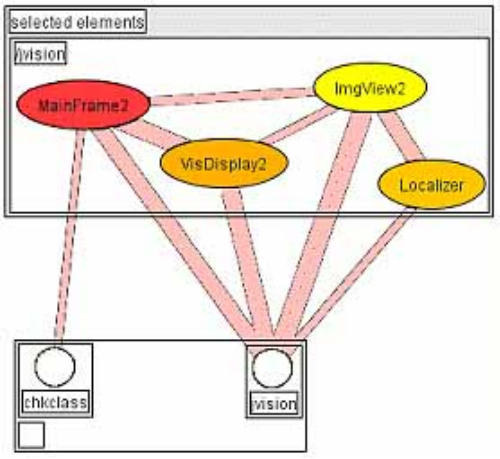



Figure 4.3: Selective Coupling of classes *MainFrame2, VisDisplay2, ImgView2,* and *Localizer*

### ZOOMING THROUGH MODULE HIERARCHY

EvoLens is capable of describing two levels of a class hierarchy within each visualization step. For each module all directly included classes and all submodules on the next level are displayed. Within the submodules, classes are directly included as if there were no further levels down the hierarchy. Classes with strong coupling that are located within the sub-hierarchy of submodules are projected onto the submodule level. EvoLens allows one stepping up and down the hierarchy interactively. The lens view shows one particular module in detail. Within this module the next level of submodules can be directly reached. Thus, users of EvoLens step down to one of the displayed submodules. In the same manner they can step up the module hierarchy. On every step the co-change coupling is interactively displayed.

Figure 4.4 describes the zoom-in into module *jvision/main* visualized in Figure 4.1 with the focal point *jvision*. It shows that many classes of module *main* are involved in the couplings. There are many external couplings too. In Figure 4.4 a new module called *chkclass* is related to the classes of the focused module *main*.

With this lower threshold we detect the external co-change coupling of module *jvision*. The main class itself called *JVision2* is related with external classes. This finding could be justified by the fact that *JVision2* is the start-up class and has therefore to be related with many parts of the entire application. However, when analyzing the software in more detail, we found out that *JVision2* exposes access to many parts of the system through static member variables. Thus, we detected design erosion within the module *jvision*.

### PANNING BETWEEN MODULES

As well as navigating through the hierarchy, the user of EvoLens can navigate to sibling modules and submodules. This horizontal navigation incorporates extended ”panning” into our visualization framework. Thus, EvoLens provides navigation not only vertically but also horizontally.

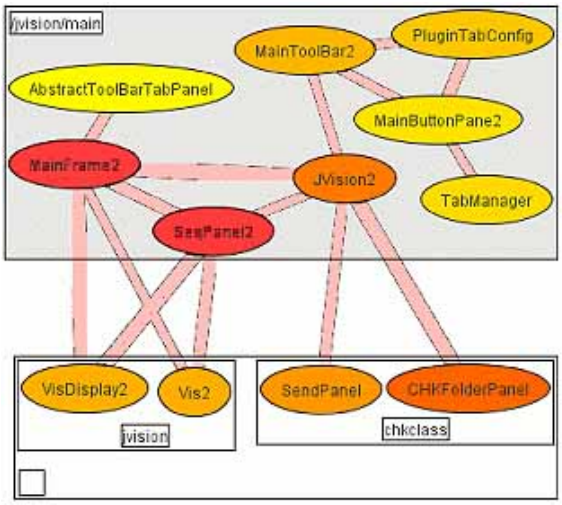



Figure 4.4: Navigation of the module hierarchy by extended Zooming and Panning

Interactive users can move the magnifying glass over a related module. Then this module becomes the new focused one.

### COUPLING THRESHOLD

Users of EvoLens may decide to navigate through the structure and decide which part of the system they are interested in. Within each step only the couplings above a selected threshold are depicted. To find a good balance between clarity and information details, the user can individually adjust the lower bound of the visualized couplings. The threshold defines the minimal strength for couplings that are extracted to be displayed by EvoLens.

Based on the settings of Figure 4.1 we instruct EvoLens to set the threshold of couplings to 10 instead of 15 to see more couplings and the related classes. Figure 4.5 shows the resulting image. With the help of this lower threshold we obtain a finer-grained picture of the couplings within this case study. The classes *ImgView2* and *VisDisplay2* exhibit coupling with several other classes of submodules within module *jvision*. The coupling of *ImgView2* seems reasonable because the related classes are part of the imaging framework. However, *VisDisplay2* should not be changed often together with *SeqPanel2* as the display of images has not much in common with administrative information presented by *SeqPanel2*.

### SLIDING TIME WINDOW

Change couplings are measured in a time window. For example, software engineers may be interested in the coupling of a module through the last six months. With the help of EvoLens they can adjust the desired beginning and end of the time window and interactively retrieve the coupling information. Additionally, a fixed size time window may be slided over the timeline. Therefore, the slider has to be dragged, which interactively adapts the start and end date of the time window.

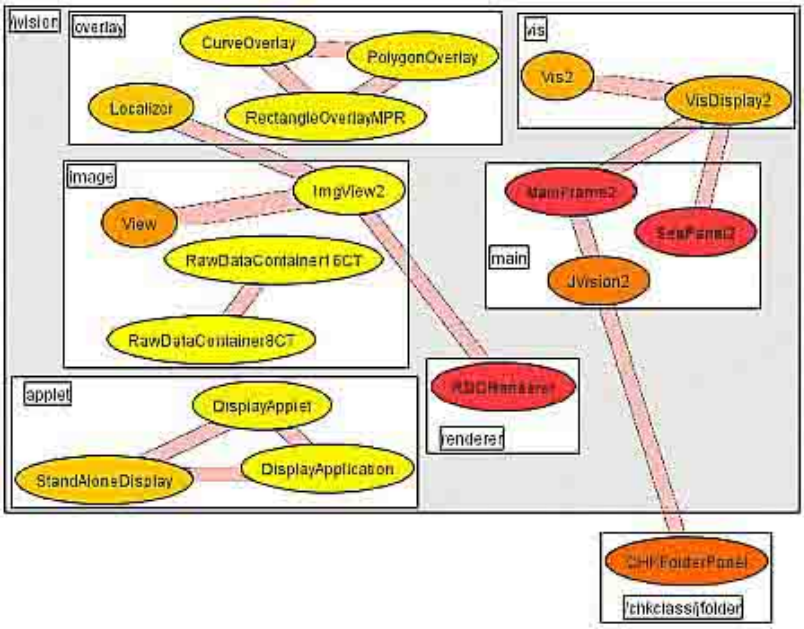



Figure 4.5: Detailed View with a low coupling threshold

Users of EvoLens do not have to switch to another view to navigate in time. They can adjust the size of the time window itself. The graph responses interactively to these adaptations. Figure 4.1 shows the coupling of module *jvision* within the entire eighteen months of the inspection period. In contrast, Figure 4.6 describes the first nine months of this period. So we see how the relationships between the classes evolved. In Figure 4.6 no coupling can be detected between class *MainFrame2* and class *VisDisplay2*. At the end of the eighteen month period these two classes are strongly related.

## 4.2 CHANGE SMELLS

In this section we present an example for the usage of co-change couplings together with refactoring for the improvement of evolvability of the software system. The notion of ”bad smells” was introduced by Fowler [38] and describes a vague suspicion that the software contains design deficiencies that should be refactored. We extend the concept of ”bad smells” to introduce ”change smells”, wich are structural weaknesses that are detectable with the help of software evolution (i.e. the visualization of co-change coupling). Refactoring is a vital technique to improve the design of existing code by changing a software system in such a way that the external behavior of the code is not changed yet the internal structure is improved. Many different refactorings have already been identified (e.g. [38, 90]). Based on our visualization approach of co-change couplings we identify locations of change smell and apply refactoring to improve the evolvability. After a time period of several moths we analyze this part of the software a second time to see whether the initially suggested and implemented refactorings were effective with respect to

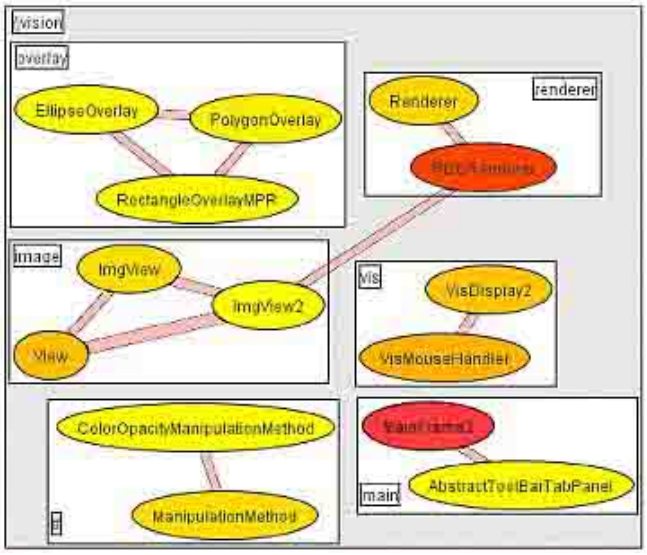



Figure 4.6: Sliding Time Window: First half of the inspection period

co-change coupling.

For demonstration how the evolvability of software can be improved with the help of our visualisation approach we focus on the PACS system described in Section 1.2. For the evaluation of our approach on the improvement of evolvability we investigate the history of the system over a period of 30 months (15 before the refactoring and 15 after). During this time the software grew from approximately 2000 to more than 5000 classes. Within the analysis of the historical data received from CVS we identify a small module (i.e. Java package) with a high changing activity. So we calculate the logical couplings of this module called *jvision/workers*. To get a better understanding of logical couplings for the classes of *jvision/workers* we use a simplified grafical representation (see Figure 4.7).

In this representation classes are depicted as small ellipses. The ellipses are grouped by their membership to modules. Modules themselves are depicted as bounding ellipses surrounding their classes. This structural information is enriched with historical data. From CVS we extract the evolution of classes and calculate co-change coupling between classes. This coupling is depicted in Figure 4.7 through edges connecting the ellipses whereas the thickness of the edges describes the ”strength” of the visualized couplings. The more often a pair of classes is changed at the same time the thicker is the representing edge. This visualization approach has been extended with class based metrics and implemented in EvoLens. The navigation through the couplings based on our visualization approach helps to locate a change smell that we call *Man-in-the-middle* for the class *ImageFetcher*.

**Man-in-the-Middle**: A central class evolves together with many others that are scattered over many modules of the system. Thus, we detect co-change couplings between the central class and the related ones; these related classes often exhibit co-change couplings among each other as well. A *Man-in-the-Middle* smell hinders the evolution of single modules, because of the strong dependencies to other parts of the system. The central class does not necessarily contain much

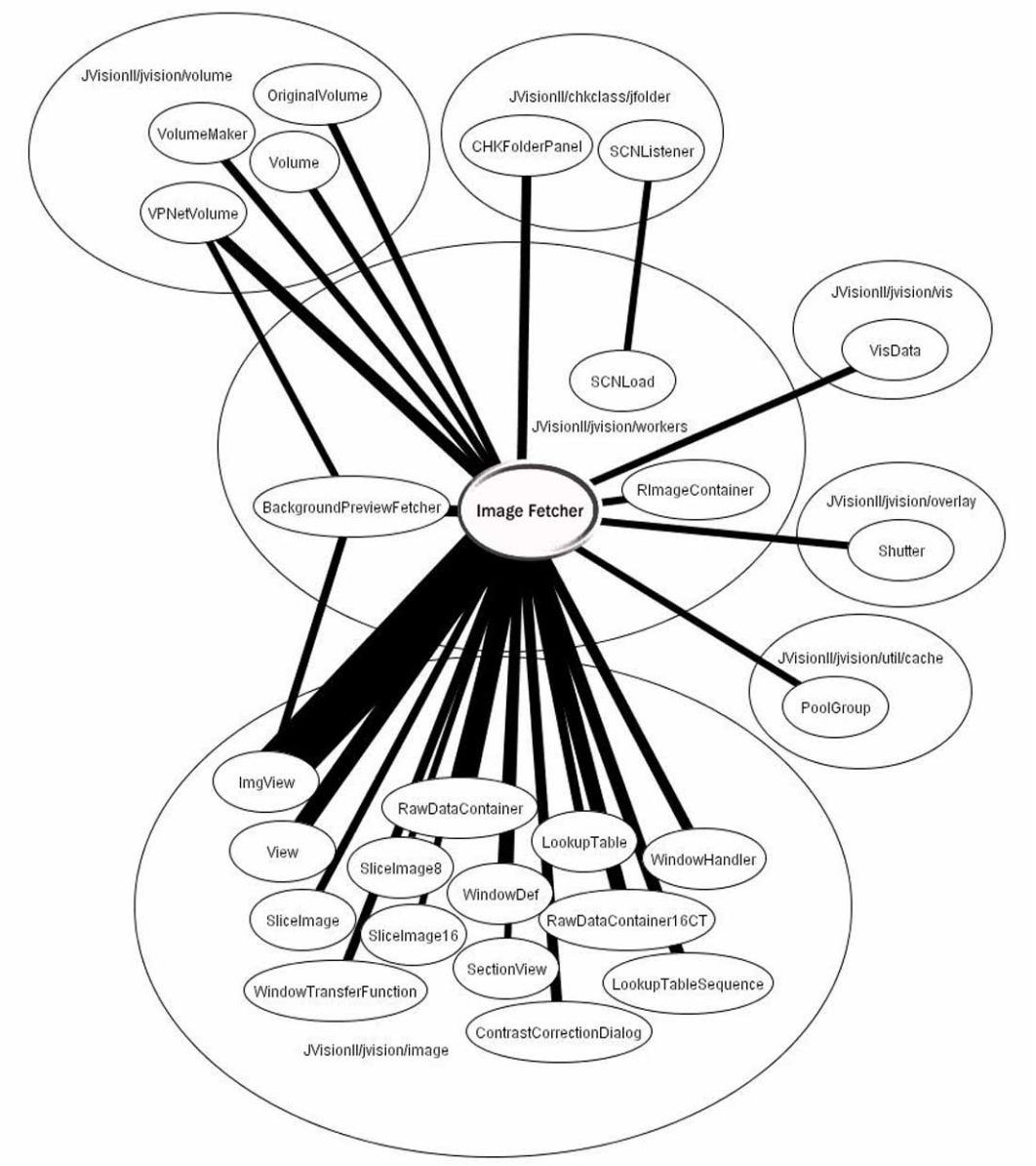



Figure 4.7: Change smell: Man-in-the-Middle

code. Refactorings such as *Move Method* and *Move Field* [38] can repair such a smell. Then the functionality can be pulled to the data and slim interfaces may be introduced.

The class *ImageFetcher* in Figure 4.7 has multiple strong co-change couplings that is often external coupling with other classes. The situation is even worse as it often changes with other classes not from one other part of the system but with several classes of different packages. Thus, when a change has to be made by an engineer, it is scattered over the system. *ImageFetcher* is one of the largest classes of the entire software; it contains almost 2000 lines of code. The methods of this class are of exceptional length: Some of them contain more than 100 lines of code. When trying to reveal the reasons for such ”spaghetti code”, we discover that many methods are similar. Thus, the entire class is internally redundant. The length of the class itself does not automatically lead to the necessity of refactoring, but *ImageFetcher* often changes together with other classes. Thus, each change has to be thoroughly analyzed to implement all necessary changes, which may be scattered over a large part of the system. This has a severe impact on the maintenance effort: When a bug is discovered within one of the methods of this class, many other methods have to be changed in a similar way. Often such changes are missed and have to be fixed later when the bug re-occurs. This results in a high changing activity.

Additionally, this class seems to have divergent changes [38], because it changes together with a lot of classes of other modules. Thus, some methods seem stronger related with classes of a particular module, whereas other methods of *ImageFetcher* have to be changed in conjunction with classes of other modules. When inspecting the source of the *ImageFetcher* we determine that the principle of separation of concerns is violated. This class implements a thread pool, a queue for work items, and logic for loading images altogether. As a result, different classes implementing different functionality are related with *ImageFetcher*.

### 4.2.1 Refactoring to improve Evolvability

We apply several refactorings to reduce the disadvantages of this change smell. Then we continue to observe the evolution of the module *jvision/workers* again for 15 months to see if the evolvability has been improved through evolution guided refactoring.

To minimize code duplication, we first extract code clone parts of methods and reuse the newly formed methods where appropriate. For that, we apply the *Extract Method* refactoring that helps to get reusable items. After these improvements the class contains just 1100 lines of code, because of the removal of duplication.

To further improve the evolvability, we split *ImageFetcher* into new classes encapsulating the different concerns. We move the methods and data for image loading into a separate class called *FetchWorker*. The logics for thread pooling and the handling of the work queue are left together in *ImageFetcher*. After the movements we obtain a surprising result: *FetchWorker* contains just one public method called *loadimage()*. This simple interface results in reduced coupling. Also *ImageFetcher* has a simple interface after the refactorings. It provides methods for starting and stopping the thread pool and for adding orders to load certain images into the work queue.

### 4.2.2 EVOLUTION AFTER REFACTORING CHANGE SMELLS

After our refactorings, we observe the software again for 15 months, which is exactly the period we analyzed the system before the refactoring. We inspect the system for such a long period to gain more accurate assessments. Fig. 4.8 describes the co-change couplings of module *jvision/workers* after the refactorings, in contrast to Fig. 4.7 that presents the couplings, which are used as trigger for the refactorings. In Fig. 4.7 we observe the system from January 2002 until March 2003. At the end of March 2003 we refactor *ImageFetcher* and Fig. 4.8 represents the development from April 2003 until June 2004.

During the second 15 months the development of the module *jvision/workers* continued on a high level. A lot of functionality was added and improved. As a result, new classes such as *VisualWorker*, *VisualWorkerData*, *VolVisualWorkerData*, and *HiSpdFetcher* were added during that time. However, Figure 4.8 exhibits no strong co-change coupling for the classes of module *jvision/workers*. Hence, several classes are changed during the second observation period, but not even two classes have been changed more than six times together. The refactored classes *ImageFetcher* and *FetchWorker* have fewer then 4 common changes with other classes. The external coupling to other classes can be reduced significantly. When asking developers for the reason of this evolution, they stated that the interfaces of the new classes were now much clearer and the classes could be developed more individually. As a result of the refactoring based on co-change couplings we can improve the structure of evolutionary hot spots and the evolvability of the software system.

Fig. 4.8 contains a web of co-change couplings within module *jvision/workers*. Especially, *ImageFetcher* is connected with many other classes. One of these classes is the newly refactored class *FetchWorker*. These two classes have been changed together twice. Thus, the absolute level is low. What about the entire web of connected classes? Many of the involved classes provide different load strategies to *ImageFetcher*, but they are not organized in a well-designed inheritance hierarchy. Therefore, we need a second refactoring step to build up an inheritance hierarchy for different image loading approaches. Again, this situation can be detected with the help of our approach visualizing co-change couplings.

## 4.3 RÉSUMÉ

This chapter focused on project assessment from an architectural perspective. The relationship between elements within an object-oriented system is an important information for software engineers. Several dependencies are discoverable through statical analysis of the source code such as inheritance or invocation. Others are related to the development process and are discoverable through software evolution analysis. Our extraction of co-change couplings is an example for such an analysis. We use a technique to discover common change behavior that is similar to association mining [124], where we identify correlations between items in a dataset (i.e. co-change coupling). This enables the extraction and visualization of evolution patterns and provides the foundation for the identification of undesired software evolution.

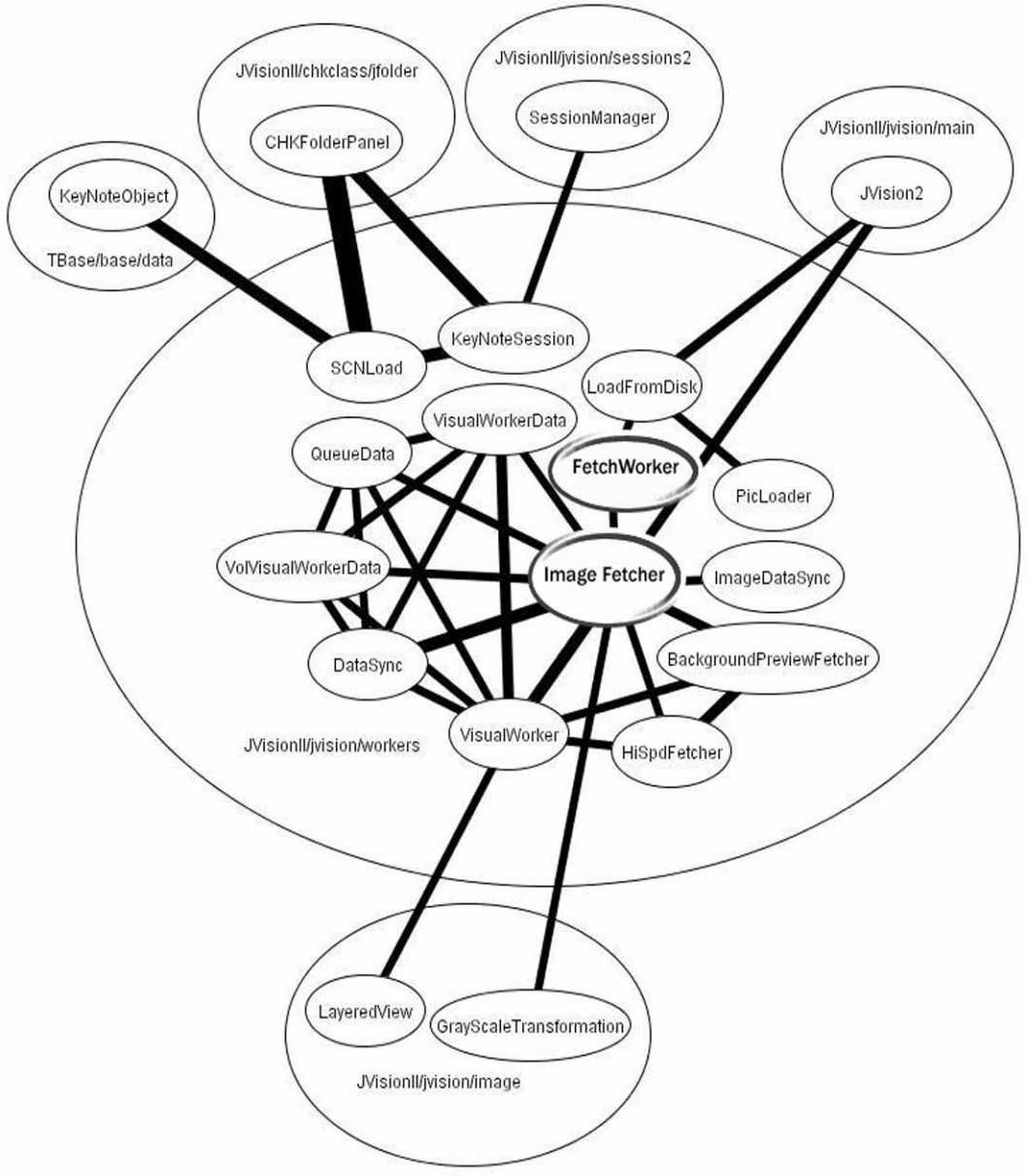



Figure 4.8: Evolution after refactoring of change smell

- Regarding the research questions stated in Chapter1.2 this chapter focuses in particular on Q3: *How can data mining be efficiently utilized in software projects to improve software evolution?* We adopted the concept of ”bad smells” and provided additional *change smells* based on change coupling analysis. Such a smell is hardly visible in the code, but easy to spot when viewing the change history. With our approach of visualization we could identify location of the anti-patterns *Man in the Middle*. Based on the co-change couplings and the proposed change smells, the developer obtains support where to apply refactorings efficiently. In an industrial case study we have shown how these change smells can be cured and how refactoring can be based on them. It turned out that after the refactorings had been implemented, the evolution of the system that we observed for another 15 months was facilitated and did not lead to the originally strong change couplings or change smells. When talking to the developers, they stated that the directed refactorings were effective for them and the new interfaces and classes were much clearer and easier to use. From this we conclude, that such an approach can help in improving the evolvability of a large software system.

- To be able to address the research question Q3 we have also to focus on Q1: *How to set up metrics from sources such as modification reports and process/project management tools?* In this chapter, we have only extracted data from software repositories (i.e. CVS), which provide information about changes to files. To identify work steps of developers as a whole we reconstruct transactions when a developer commits changes to a group of files. For this task we implement a technique similar to association mining, where we ”deduce” co-change coupling from commit transactions. The frequency of common commits of files describes the strength of the coupling that we visualize for the analysis of change smells.

- This leads to Q2: *How do software evolution metrics relate to external software product attributes?* We use co-change coupling to investigate how well a system can be evolved to fulfill customer needs, which is one of the external product attributes relevant in software projects in addition to software defects. Evolvability of software systems addresses quality attributes such as understandability (i.e. the source code can be easily read and understood) and changeability (i.e. the effort needed to implement modifications). If we find very strong coupling or a large network of interrelated classes, these are indicators for the necessity for an in-depth analysis. Our visualization approach supports software engineer in the exploration of the historical development of a system. They can inspect the dependencies that influence the progress of developers and can try to change the situation like we did with the help of refactoring.

# Chapter 5

# Identification of Design Deficiencies: Prediction of Refactoring

Chapter 4 focuses on the retrospective assessment of software systems utilizing co-change coupling between files, where the effect of activities such as refactoring is assessed after these activities take place. In this chapter, we provide indicators for the future trends in software projects. How well can we predict the events that will take place within the next few months based on the evolution metrics defined before? We show that it is possible to predict files that are prone to refactoring in the next two months with the help of evolution metrics from just three months of development time.

Refactoring is a state-of-the-art practice in software development to improve the design of existing software systems without changing the external behavior. Developers often use this technique to prepare object-oriented systems for further improvements and extensions of functionality. The identification of hot-spots where refactorings will take place improves the effectiveness of engineers in focusing on the relevant classes that will undergo changes in future [65]. For project managers it is interesting to know which locations are likely to demand refactoring. Refactoring improves the understandability of the code, but on the other hand requires development time [29]. The prediction of future refactorings allows project managers a better coordination of software development and project management, a more accurate budgeting, and efficient manpower utilization.

As refactoring is a technique to improve the design, the need for refactoring points to locations that have to be improved in terms of design. Thus, the prediction of refactorings is also a prediction of files that are likely to contain design deficiencies. Certainly, source code inspection can be used to reveal the demand of refactorings, too. However, this work is very labor intensive and must be carried out by specialists. As a result, the identification of software entities that need refactoring is expensive. Therefore, we describe an automation of this approach by utilizing data mining. A source to determine required refactorings of software entities such as classes can be their past evolution history. An entity, which undergoes many changes in the past, bears a certain probability for refactoring. For this technique for project assessment we use the evolution metrics defined in Chapter 3.

| | Hypotheses |
|---|---|
| H0 | Information about software evolution can be provided in a form that is suitable for software project assessment. |
| H1 | Evolution data is a good predictor of future refactoring. |
| H2 | It is possible to predict refactoring on short time frames. |
| H3 | We can accurately predict the number of future refactorings for each file. |
| H4 | There is a common subset of features essential for predictions in different projects. |

Table 5.1: Research hypotheses

## 5.1 HYPOTHESES

Previous research activities investigated the predictability of quality measures such as error proneness, defect densities and time to failure [55, 70, 93]. We create models for the prediction of another type of event within software engineering projects such as refactorings. Similar to defects, refactorings provide an insight into the necessary rework we have to expect in future development activities. Table 5.1 summarizes our hypotheses to guide our research. The hypothesis H0 applies to all our prediction techniques and is addressed in Chapter 3.

## 5.2 PREDICTION TARGET: REFACTORING

With the features described in Section 3.2 we predict the number of refactorings. Our predictions are based on nominal models where two different groups of elements can be distinguished. Thus, we group files in one of the two classes: with vs. without refactoring. Further, we investigate the refactoring prone files in more detail and divide this group again in two classes: having one refactoring vs. having several refactorings (see Figure 5.1). In the field of data mining this method is called *classifier stacking* [39, 121]. For the evaluation in this chapter we use only two out of the three projects of our field study described in Section 1.2. This is due to the fact that the developers of commercial PACS project do not apply refactorings regularly and we cannot identify enough refactorings (more than 1%) to create prediction models. As a result, we focus on ArgoUML and the Spring framework, where the number of refactorings is approximately 10% of all changes.

### 5.2.1 IDENTIFICATION OF REFACTORINGS.

The number of refactorings is obtained from the commit messages of the versioning system. We use evolution data not only for the computation of data mining features, but also for the identification of change events as refactorings. For our prediction models we do not distinguish different types of refactorings (e.g. create super class, rename of method/class, extract method, etc.). We only assess the fact that developers intend to improve the design, where they state that they applied refactoring. As a result, for our investigated projects we identify refactorings based

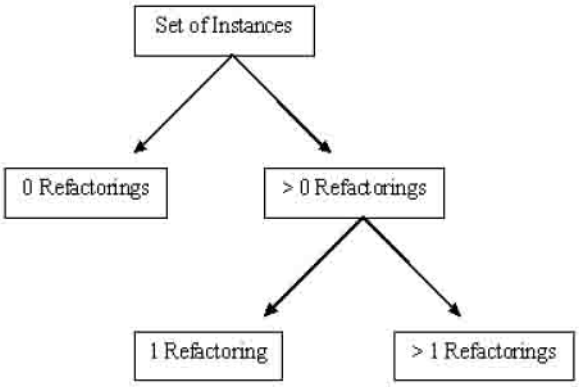



Figure 5.1: Analysis setup

on the commit message, which are provided by developers as comments for their modifications of source code. We start our identification by search for the term ”refactor”. With this simple approach we find 2070 changes to files (i.e. revisions) for ArgoUML and 1959 changes for the Spring framework. We analyze the results and discovere that the code is not a refactoring, when ”needs refactoring” is included in the commit message. Thus, we optimized our search in an interative approach, where we leave out changes that are not refactoring and contain phrases such as ”needs refactoring”. Finally, based on the term ”refactor” we find 1467 changes for ArgoUML and 1798 for the Spring framework. As next step we add new terms to our iterative search approach. For ArgoUML we focus on the terms ”refactor”, ”restruct”, ”clean”, ”not used”, ”unused”, ”reformat”, ”import”, ”remove”, ”replace”, ”split”, ”reorg”, ”rename”, and ”move”. For the Spring framework we use ”refactor”, ”restruct”, ”clean”, ”not used”, ”unused”, ”reformat”, ”import”, ”remove”, ”replace”, ”split”, ”reorg”, ”rename”, and ”move”. With several refinements we used for each project 15-20 SQL queries to mark modifications as refactorings.

EVALUATION OF REFACTORING IDENTIFICATION.

With our SQL queries based on the commit messages we labeled 7758 of 60369 changes as refactorings for ArgoUML (13%) and 6251 of the 56050 changes for the Spring framework (11%). To estimate the number of refactorings we marked correctly with our method, we used a statistical evaluation. For each project we randomly selected a subset of 100 modifications and checked whether or not it is a refactoring and if we labeled them as refactoring. Table 5.2 shows that many changes can be labeled correctly with our SQL queries: For ArgoUML only one modification (in the random set of 100) was labeled as refactoring, which turned out not to be a refactoring (false positive) and two refactorings were missed (false negative). For the Spring framework we received even better results. Although Spring exhibits a very unbalanced distribution of only 11% refactorings in our random selection, we missed only one refactoring in our labeling with the help of SQL queries and identified only one modification wrong as refactoring.

| Project | Modifications | Identified Refactorings | Other Types | False Positives | False Negatives |
|---|---|---|---|---|---|
| ArgoUML | 100 | 16 | 84 | 1 | 2 |
| Spring | 100 | 11 | 89 | 1 | 1 |

Table 5.2: Evaluation of classifying modifications as refactorings

## 5.3 DATA MINING

### 5.3.1 CLASSIFIERS: DATA MINING ALGORITHMS FOR PREDICTION MODELS

For the generation of prediction models we use several data mining algorithms:

- *J48.* [1] This classifier builds its decision nodes based on entropy information. It includes improvements for dealing with numeric attributes, missing values, and noisy data (pruning). The great advantage of decision tree compared to other algorithms is that they can be easily interpreted by humans.
- *LMT.* This is a data mining algorithm for building logistic model trees, which are classification trees with logistic regression functions at the leaves. It uses validation to determine how many iterations to run, when fitting the logistic regression function at a node of the decision tree. Thus it is a classification algorithm where first regression is built and the result is converted into classes of elements.
- *Rip.* Repeated Incremental Pruning is a propositional rule learner. It uses a growth phase, where antecedents are greedily added until the rule reaches 100% accuracy. Then in the pruning phase, metrics are used to prune rules until the defined length is reached.
- *NNge* is a Nearest Neighbor generalization. In this case a nearest-neighbor algorithm is used to build rules using non-nested generalized exemplars.

#### EVALUATION OF CLASSIFICATION.

To evaluate our prediction models we use 10-fold cross validation. In our analysis of prediction models for refactoring we use *precision*, *recall*, and *F-measure* — three essential markers characterizing model performance. These evaluation measures are defined based on formulas regarding different rates such as true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). True positives describe the predictions that are correctly classified. False positives are the ones that are classified to be in a particular group (e.g. number refactorings = 0), but the classification is wrong (e.g. number refactorings = 1 or $\geq 2$). The number of elements that is correctly classified not to belong to the given group forms the true negatives. False negatives are elements that belong to the group of interest, but are erroneously classified to belong outside of the group. (see Table 5.3)

[1] J48 is the WEKA implementation of the state-of-the-art decision tree learner C4.5

| | | Predicted | |
|---|---|---|---|
| | | yes | no |
| Actual | yes | true positive (TP) | false negative (FN) |
| | no | false positive (FP) | true negative (TN) |

Table 5.3: Outcome of prediction of two groups

| Learning Period | Target Period | Ref.=0 | Ref.=1 | Ref.>=2 | $\sum$ Files |
|---|---|---|---|---|---|
| Oct.04 - Dec.04 (3 months) | Jan. - Feb.05 | 603 | 181 | 129 | 913 |
| Jul.04 - Dec.04 (6 months) | Jan. - Feb.05 | 603 | 181 | 129 | 913 |

(a) ArgoUML

| Learning Period | Target Period | Ref.=0 | Ref.=1 | Ref.>=2 | $\sum$ Files |
|---|---|---|---|---|---|
| Aug.05 - Oct.05 (3 months) | Nov. - Dec.05 | 750 | 110 | 35 | 895 |
| May 05 - Oct.05 (6 months) | Nov. - Dec.05 | 750 | 110 | 35 | 895 |

(b) Spring framework

Table 5.4: Refactoring distribution for analyzed periods by project

- *Precision* describes the percentage of correctly classified entities.

$$precision \; = \; \frac{TP}{TP \; + \; FP} \cdot 100\% \; = \; \frac{predicted\; correct}{total\; predicted}$$

  The higher the precision the more predictions are correct.

- *Recall* describes the percentage of entities classified from the group of positive entities.

$$recall \; = \; \frac{TP}{TP \; + \; FN} \cdot 100\% \; = \; \frac{predicted\; correct}{total\; positive}$$

  The higher the recall the more elements are found.

- *F-measure* is a dimensionless measure combining precision and recall by the formula.

$$F - measure \; = \; \frac{2 \; \cdot \; TP}{2 \; \cdot \; TP \; + \; FP \; + \; FN} \; = \; \frac{2 \; \times \; recall \; \times \; precision}{recall \; + \; precision}$$

  We use the F-measure to compare the performance of our prediction models.

## 5.4 Results

To investigate our hypotheses in Table 5.1 we carried out several trials. For the investigation of the following research questions we focus particularly on the results of ArgoUML. Tables 5.5,5.6, and 5.7 show that the values of ArgoUML and Spring are comparable.

| Algorithm | Refactoring = **0** | | |
|---|---|---|---|
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.819 | 0.834 | 0.827 |
| LMT | 0.81 | 0.801 | 0.806 |
| Rip | 0.768 | 0.844 | 0.804 |
| NNge | 0.804 | 0.849 | 0.826 |
| Algorithm | Refactoring >= **1** | | |
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.666 | 0.642 | 0.654 |
| LMT | 0.621 | 0.635 | 0.628 |
| Rip | 0.624 | 0.503 | 0.557 |
| NNge | 0.67 | 0.597 | 0.631 |

(a)ArgoUML

| Algorithm | Refactoring = **0** | | |
|---|---|---|---|
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.884 | 0.937 | 0.91 |
| LMT | 0.874 | 0.961 | 0.916 |
| Rip | 0.876 | 0.975 | 0.923 |
| NNge | 0.893 | 0.913 | 0.903 |
| Algorithm | Refactoring >= **1** | | |
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.53 | 0.366 | 0.433 |
| LMT | 0.586 | 0.283 | 0.381 |
| Rip | 0.689 | 0.29 | 0.408 |
| NNge | 0.492 | 0.434 | 0.462 |

(b) Spring framework

Table 5.5: Predicting non-refactoring-prone vs refactoring-prone classes

### 5.4.1 Is evolution data a good predictor of future refactorings?

To answer this question we take a look at Table 5.5, which describes the quality of the prediction models for ArgoUML and the Spring framework. We analyze the prediction of two groups of object-oriented classes, the ones having no refactoring in the target period (defined in Table 5.4) and classes that have one or more refactorings. For both open source projects we list four different classification algorithms: J48, LMT, Rip, and NNge.

We can see that the prediction of classes that are non-refactoring-prone have better quality indicators than classes exhibiting refactorings. For ArgoUML both precision and recall are about 0.8, which results in a high F-measure of 0.8. For classes with refactorings the value range is 0.5 to 0.67, which results in F-measures of approximately 0.6 for the ArgoUML project. These values express that classes with refactorings are more difficult to predict than classes that are not

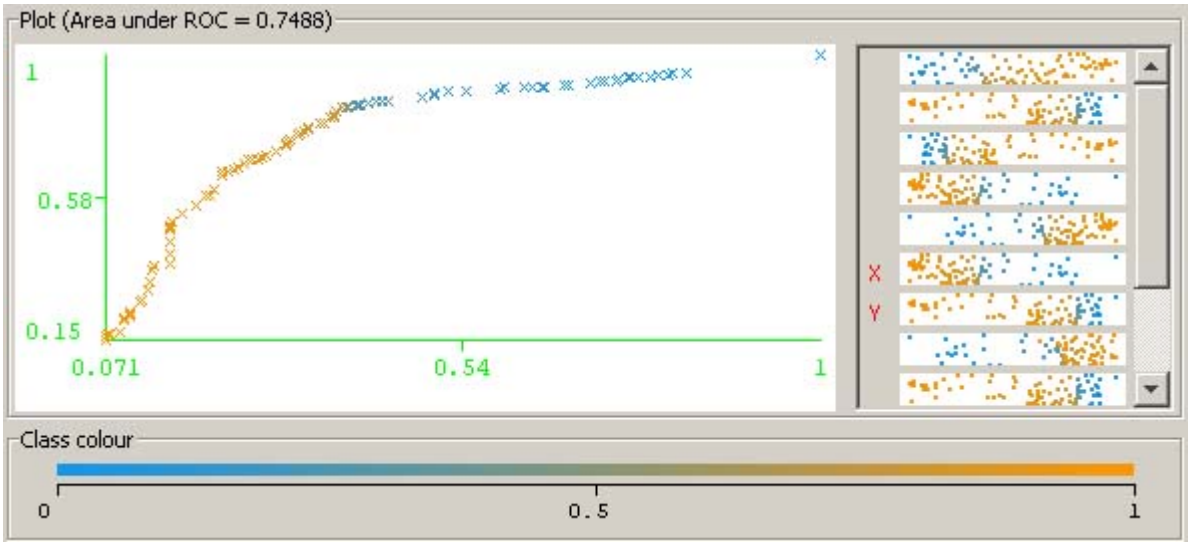



Figure 5.2: Predicting non-refactoring-prone vs refactoring-prone classes: ROC of ArgoUML with J48

prone to refactoring. One possible explanation is that we do not distinguish between different types of refactorings. Thus, changes to variables, methods and classes are weighted equally. We are solely interested in the fact that refactoring takes place. However, the discrepancy between the prediction measures of classes with zero refactoring and classes subject to refactoring is due to the fact that the distribution of these two groups is not equal.

As in both projects the precision is above 0.5 (except the NNge model for Spring) the number of correctly predicted instances is high, which is important for practical application of our approach. When the developer wants to be proactive and to take care of those classes that are prone for refactoring, one has a high probability (in most cases more than 2/3) to investigate relevant files.

Table 5.4 shows that for the first period of ArgoUML that we investigate the number of classes not being refactored is 603 and the ones with refactoring is 310 (181 + 129). The algorithms are biased towards the dominant class distribution (prior) and therefore overestimate classes with no refactoring. To assess the algorithms on equally distributed data sets, we adjust the number of classes by randomly ignoring non-refactoring-prone files. Now, the prediction algorithms perform even better: both refactoring and non-refactoring-prone classes are predicted very well with a F-measure better than 0.85 for ArguUML and 0.75 for the Spring framework.

For a visual illustration of the results we use ROC (receiver operating characteristic) curves, where the tradeoff between the hit rate and the false alarms is shown [119]. Figure 5.2 describes the prediction results of refactoring-prone classes for ArgoUML using the J48 algorithm. In ROC curves put false positive values of each prediction on the x-axis and the true positive values on the y-axis. The area under the ROC curve (AUC) is a value that can be used to compare the performance of the predictions. For ArgoUML we get an AUC value of 0.75; the closer this value is to 1 the better the predictions.

Therefore, we confirm our hypothesis H1:

| Algorithm | Refactoring = **0** | | |
|---|---|---|---|
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.811 | 0.826 | 0.818 |
| NNge | 0.799 | 0.849 | 0.823 |
| Algorithm | Refactoring >= **1** | | |
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.581 | 0.556 | 0.568 |
| NNge | 0.593 | 0.507 | 0.547 |

(a) ArgoUML

| Algorithm | Refactoring = **0** | | |
|---|---|---|---|
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.874 | 0.912 | 0.893 |
| NNge | 0.887 | 0.899 | 0.893 |
| Algorithm | Refactoring >= **1** | | |
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.514 | 0.349 | 0.416 |
| NNge | 0.481 | 0.413 | 0.444 |

(b) Spring framework

Table 5.6: Predicting refactoring proneness based on a larger time frame (6 months)

It is possible to predict refactoring with evolution data with a high accuracy.

### 5.4.2 Is it possible to predict refactorings on short time frames?

To answer the question we compare Table 5.5 with Table 5.6. The first one describes the prediction of refactorings happening in two months based on features taken from three previous months and the second describes the prediction of the same two months based on features from six months (for exact period definition see Table 5.4). The prediction with the help of three months shows even better results than the prediction based on six months. Why do we obtain these interesting results? Most open source projects, also ArgoUML, work with development practices, where refactoring is used to improve design of source code that has been introduced lately. Therefore, the last few months before refactoring takes place are the ones with the most relevant attributes.

Figure 5.3 shows the ROC curve of ArgoUML with J48 for the predictions taking six months of development time into account. Also the ROC curve confirms the trend that the prediction performance decreases, when taking six months instead of three. The F-measure decreased for ArgoUML when using the J48 algorithm from 0.827 (refactoring-prone: 0.654) to 0.818 (refactoring-prone: 0.568) and the Area under the ROC curve (AUC) also decreases from 0.75 to 0.71.

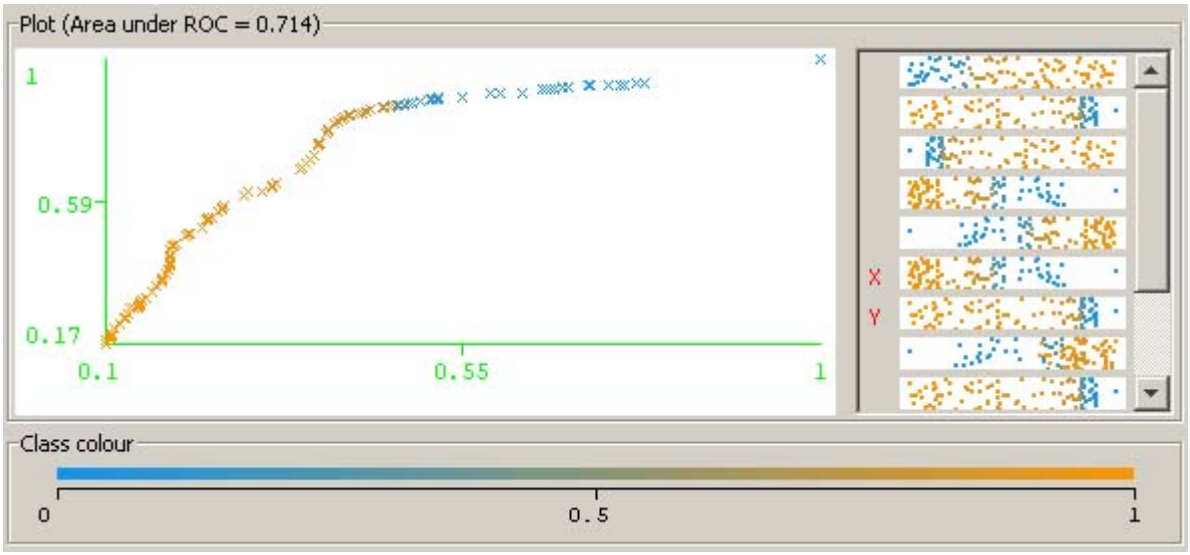



Figure 5.3: Predicting refactoring proneness based on a larger time frame (6 months): ROC of ArgoUML with J48

Thus, we conclude:

| It is possible to predict refactorings of the next two months based on the last three months of development time. |
|---|

### 5.4.3 Is it possible to distinguish between different groups of files: Without refactoring, with just one refactoring, and with several refactorings?

We investigate this research question with the help of two classification tasks: First we distinguish between classes without refactorings and classes having refactorings. Then we take the second group and examine if we can distinguish classes with just one refactoring from classes with several refactorings (see Figure 5.1). Table 5.5 shows the quality values for the prediction of non-refactoring-prone vs. refactoring-prone. We obtain high values for the F-measure, which indicates the overall performance of the prediction models. In Section 5.4.1 we describe that we could get even better measures, if the number of classes in each group is similar. As a result we can distinguish classes with and without refactoring very well.

Table 5.7 shows the results of the prediction models distinguishing classes with one refactoring from classes with several refactorings. The F-measures are not as high as the ones for the prediction of refactoring-proneness. An F-measure of 0.75 and 0.65 for the two groups of files having refactoring is still good, as the number of classes is much lower than for the prediction models of Table 5.5, which we can see in Table 5.4.

When developers take care of classes that are highly refactoring-prone, they would investigate the group with $>= 2$ refactorings. In this group the precision is quite high being close to or above

| Algorithm | Refactoring = **1** | | |
|---|---|---|---|
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.747 | 0.735 | 0.741 |
| NNge | 0.767 | 0.746 | 0.756 |
| Algorithm | Refactoring >= **2** | | |
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.636 | 0.651 | 0.644 |
| NNge | 0.657 | 0.682 | 0.669 |

(a)ArgoUML

| Algorithm | Refactoring = **1** | | |
|---|---|---|---|
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.694 | 0.718 | 0.708 |
| NNge | 0.713 | 0.725 | 0.719 |
| Algorithm | Refactoring >= **2** | | |
| | Prec.(%) | Recall(%) | F-measure |
| J48 | 0.624 | 0.617 | 0.62 |
| NNge | 0.593 | 0.638 | 0.615 |

(b) Spring framework

Table 5.7: Predicting classes with one refactoring vs. classes with several refactorings

0.6, which is important, because developers have a high probability to look at relevant classes. As the recall is also around 2/3, developers have the opportunity to analyze many refactoring-prone classes to assess their design quality and their impact on the software architecture.

Figure 5.4 shows the ROC curve for the distinction between classes with one refactoring and classes with several refactorings based on the J48 algorithm. We can see that this curve is not as good as the one predicting refactoring-prone classes (Figure 5.2). However, the trend of the F-measures is also noticeable in the area under the ROC curve (AUC). The AUC of the prediction of classes with several refactorings is 0.69, whereas the AUC for the prediction of refactoring-proneness is 0.75. Similarly the F-measure of between these two experiments decreases from 0.827 (refactoring-prone: 0.654) to 0.741 (several refactorings: 0.644) for ArgoUML using the J48 algorithm.

We come to the following conclusions:

Refactoring-prone/non-refactoring-prone classes can be identified accurately.

The distinction between classes with one or several refactorings is possible.

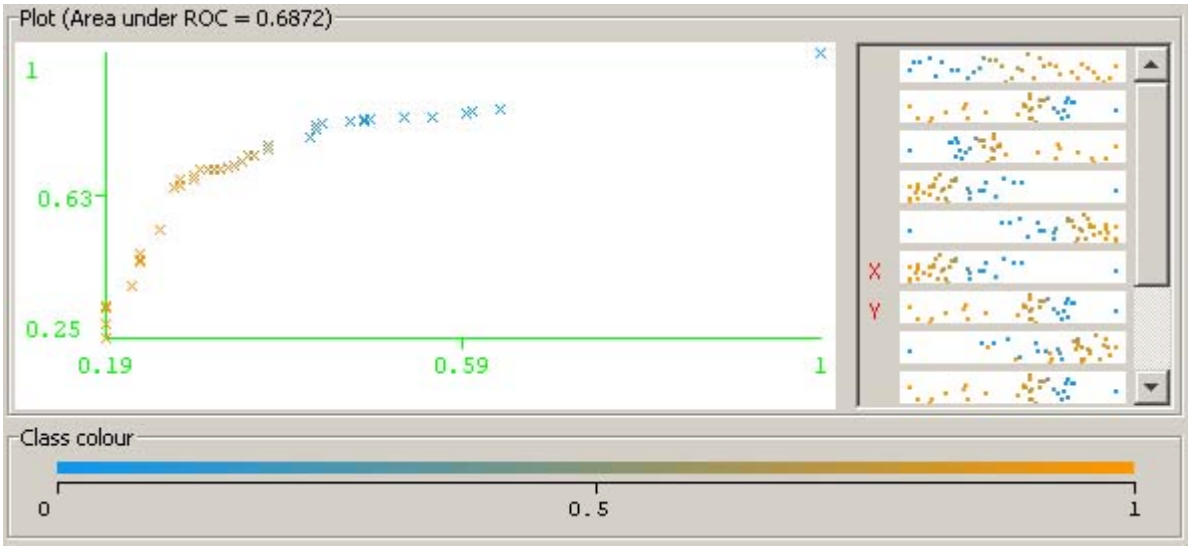



Figure 5.4: Predicting classes with one refactoring vs. classes with several refactorings: ROC of ArgoUML with J48

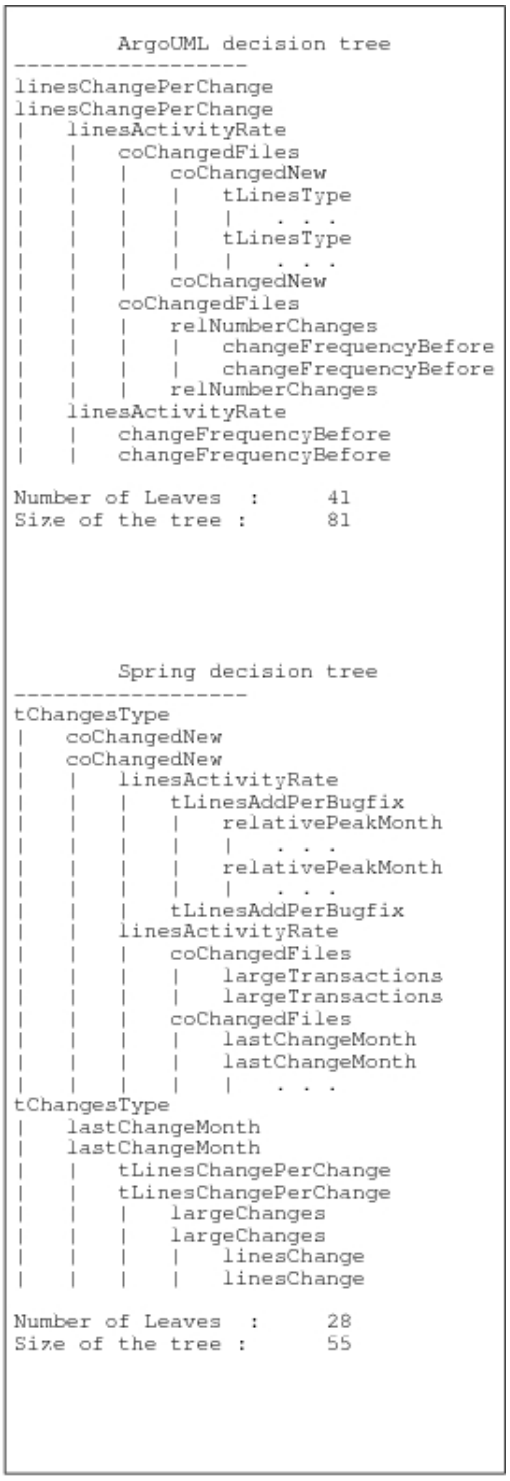
### 5.4.4 Is there a common subset of attributes for different projects?

To answer this question we take a look at the decision trees of the two projects in our case study (Figure 5.5). The trees represent the result of the classification of instances containing no refactoring vs. instances with one or more refactorings. The higher the nodes in the tree the more relevance they have for the prediction. We restrict our trees in Figure 5.5 to five levels out of twelve to investigate only the most important features.

**ArgoUML**
The topmost attributes of model ArgoUML, starting from the root are: *linesChangePerChange*, *linesActivityRate*, *coChangedFiles*, *changeFrequencyBefore*, *coChangedNew*, *relNumberChanges*, and *tLinesType*.

**Spring framework**
The topmost attributes of model Spring, starting at the root are: *tChangesType*, *coChangedNew*, *lastChangeMonth*, *linesActivityRate*, *tLinesChangePerChange*, *tLinesAddPerBugfix*, *coChanged-Files*, *largeChanges*, *relativePeakMonth*, *largeTransactions*, *lastChangeMonth*, and *linesChange*.

**Common** Both tree models of ArgoUML and Spring framework have the following attributes in common: *coChangedNew*, *linesActivityRate*, and *coChangedFiles*. The first one *coChangedNew* describes the number of files that are created (newly introduced) together with changes to the inspected instance. This feature indicates that new functionality is added, because new classes are introduced together with modifications of the inspected class. If *linesActivityRate* describes that lines are changed often during the entire lifetime of the class, then also the probability for the number of refactorings rises. The number of classes changed together with the inspected one is described by *coChangedFiles*, which takes into account the importance of interrelationships in object-oriented software systems.



```
        ArgoUML decision tree
------------------
linesChangePerChange
linesChangePerChange
|   linesActivityRate
|   |   coChangedFiles
|   |   |   coChangedNew
|   |   |   |   tLinesType
|   |   |   |   |   . . .
|   |   |   |   tLinesType
|   |   |   |   |   . . .
|   |   |   coChangedNew
|   |   coChangedFiles
|   |   |   relNumberChanges
|   |   |   |   changeFrequencyBefore
|   |   |   |   changeFrequencyBefore
|   |   |   relNumberChanges
|   linesActivityRate
|   |   changeFrequencyBefore
|   |   changeFrequencyBefore

Number of Leaves  :     41
Size of the tree :      81




        Spring decision tree
------------------
tChangesType
|   coChangedNew
|   coChangedNew
|   |   linesActivityRate
|   |   |   tLinesAddPerBugfix
|   |   |   |   relativePeakMonth
|   |   |   |   |   . . .
|   |   |   |   relativePeakMonth
|   |   |   |   |   . . .
|   |   |   tLinesAddPerBugfix
|   |   linesActivityRate
|   |   |   coChangedFiles
|   |   |   |   largeTransactions
|   |   |   |   largeTransactions
|   |   |   coChangedFiles
|   |   |   |   lastChangeMonth
|   |   |   |   lastChangeMonth
|   |   |   |   |   . . .
tChangesType
|   lastChangeMonth
|   lastChangeMonth
|   |   tLinesChangePerChange
|   |   tLinesChangePerChange
|   |   |   largeChanges
|   |   |   largeChanges
|   |   |   |   linesChange
|   |   |   |   linesChange

Number of Leaves  :     28
Size of the tree :      55
```

Figure 5.5: Decision trees based on classifier J48 for classification 0 vs. $>=$ 1 refactoring

The trees have more commonalities than just these features. *linesChangePerChange* is the topmost feature in the tree of ArgoUML, which describes the average number of altered lines within change events, which is measured for each predicted instance. A similar measure appears in the Spring framework where *tLinesChangePerChange* is located on the third level in the second half of the tree, which describes the number of altered lines within the files of the entire transaction where the file of interest was changed. It is surprising that people related features like the *number of authors* are not represented in the trees of our case study.

We conclude that:

| There is a common subset of attributes for different projects: *coChangedNew, linesActivityRate*, and *coChangedFiles*. |
|---|

## 5.5 RÉSUMÉ

In this chapter, we focused on one particular aspect of software evolution: Refactoring to improve the design of existing code without changing the external behavior. In particular it supports understandability as it aims to clarify object-oriented systems through restructuring. Also methods such as renaming help to express what the code is intended to do. Other quality aspects can be addressed with the help of refactoring too. The basis of refactorings are often unit tests to ensure that the changes do not introduce unintended side effects. Thus, testability can be also supported with the regular application of refactoring on software projects. In general the flexibility of a project is leveraged with refactoring, as necessary extensions and adaptations can be prepared to fit in the overall design. With our prediction models for future refactoring activites we can identify classes that are prone to refactoring and therefore have design deficiencies. The following research questions of Chapter 1.2 are covered by the current chapter:

- Q2: *How do software evolution metrics relate to external software product attributes?* In contrast to previous studies where quality measured by the number of defects is predicted (e.g. [103]), we created classification models for refactoring, which reflects the need for design improvements. Refactoring is an essential element to keep the quality high and to allow further evolution based on new customer needs. We demonstrated that several features such as *lines activity rate* and *number of lines altered per commit* provide substantial information for the assessment of refactorings. But also the structure of the system is crucial for refactorings, as the *number of co-changed files* and the *number of files introduced during the maintenance* are relevant features in the established prediction models.
- Q3: *How can data mining be efficiently utilized in software projects to improve software evolution?* This question is covered in this chapter through several aspects: With our approach of refactoring prediction we can point to locations of a system, where this kind of preventive maintenance usually takes place. The software engineers can pro-actively take a look at the predicted classes and investigate the necessity for the improvement of

the existing design. Thus, the knowledge that refactoring is probable to happen in the near future can be used to work on the items as early as possible, which helps to focus on the design on these parts that need more effort. Additionally, the effort can be estimated by the project manager. Based on the predictions one obtains an overview how many refactorings will take place in the next few months and also how the maintenance activities will be scattered over the system's architecture. Then the required time for refactorings can be planned in accordance with the project time lines and the largest effort for refactoring coordinated with release dates, as around the release the time runs out and the focus is directed to release the software and not to improve the design for future changes.

# Chapter 6

# Predictability of Different Defect Categories

In the previous chapters we focused on the quality of the design a certain system has from a technical perspective. In the current chapter we take the first steps towards an important aspect of customer perceived quality: The number of defects. First we show that defects can be predicted on a short-term basis like the predictions of refactorings as an indicator for design deficiencies in the previous chapter. This time we not only distinguish different groups of files (refactoring-prone vs. non-refactoring-prone), but create numeric prediction models for the exact number of defects in a file. Afterwards, we compare the predictability of pre-release defects with post-release defects. The second group is then divided into the ones that are discovered by internal staff (e.g. in testing even after a release) and the files containing defects reported by external parties. The analysis of different defect categories is rounded off by an investigation of the predictability of defects with high severity.
Successful prediction models have to take into account many aspects of the software development and maintenance [34]. In contrast to previous attempts on defect prediction based on software evolution (e.g. [70]), we use detailed data from a versioning system as well as an issue tracking system to create new prediction models. As a result we consider different aspects such as difficulty of problem, complexity of existing solution, team structure, process orientation, testing process (see Chapter 3) to build up an effective prediction model.

## 6.1 Hypotheses

To guide the metrics selection for defect prediction and our evaluation with a case study, we set up several hypotheses. Previous studies discovered that data about software processes can be efficiently used to build quality prediction models, where an increase in relative change events results in a higher defect density of the analyzed software [86]. In contrast to these research approaches we aim our hypothesis at a fine-grained level, to get an in-depth analysis of aspects related to software quality.
In the following, we present our five hypotheses:

- *H1: Defect density can be predicted based on a short time frame.* Previous research activities often focused on prediction of longer time frames such as releases. In our research we focus on months as time scale. We predict defect densities for the next month or the next couple of months.

- *H2: Critical defects with high severity have a low regularity.* Prediction models build on the regularity of the underlying data and can predict events better that correspond to this regularity. We expect that defects that are critical are more difficult to detect as they ”hide better” during the testing and product delivery.

- *H3: Quality predictions of pre-release defects are more accurate than for post-release defects.* Project quality can be estimated in different stages of the development process. Some stages are more difficult to appraise than others. Previous studies have already indicated that the accuracy of data mining in software engineering varies over time (e.g. [106]). We expect that defects that are detected before a release date are easier to predict than defects that are reported afterwards.

- *H4: Defects discovered by internal staff have more regularity than defects reported by the customer.* For prediction model creation it is an important input to know where the defect comes from. Was it recognized by internal staff (e.g. during testing) or does the defect report come from customer sites? We expect that internally and externally detected defects have different characteristics. As a result one group can be easier predicted than the other one.

- *H5: Software and development team data have to be considered for an accurate defect prediction.* We use a large amount of evolution indicators for defect prediction. These indicators can be grouped into several categories such as size and complexity measures, indicators for the complexity of the existing solution, and team related issues. For defect prediction we expect that data mining features from many different categories are important.

## 6.2 Defect Data

For our defect prediction models we are counting known defects during the history of source code files, which approximately correspond to classes in Java. We estimate defects within a given period (usually two months) based on the previous development history. To obtain the defect rate for each source file we use the data from the issue tracking system where the relationship between versioning system and issue tracking is established as described in Section 3.1.2. As defect we interpret each revision of a file that has an issue attached of type bug.

## 6.3 DATA MINING

For model generation and evaluation we use the data mining tool called Weka [119]. It provides many algorithms for different data mining tasks such as classification, clustering, and association analysis. For our prediction and classification models we selected linear regression, regression trees (M5), and classifier C4.5. The regression algorithms are used to predict the number of defects for a class from its evolution attributes.

- *Linear regression* is a method of estimating the conditional expected variable given the values of some other variables, which are called features. It is called ”linear” because the relation of the response to the explanatory variables is assumed to be a linear function of some parameters. In contrast, a multi-layer neuronal network is an example of a nonlinear regression model [119].
- *Regression trees (i.e. M5)* produce decision trees with numeric output for leaf nodes, where the average numeric value is used for the prediction. Such trees are built through binary recursive partitioning, which is an iterative process of splitting the data into partitions, and then splitting it up further on each of the branches. The algorithm chooses the split that partitions the data into two parts such that it minimizes the sum of the squared deviations from the mean in the separate parts [97, 119].
- The *C4.5 classifier* includes improvements for dealing with numeric attributes, missing values, and noisy data. This classifier compares one of the input attributes against a threshold and partitions the input space with axis parallel splits [98, 119].

### 6.3.1 PREDICTION ASSESSMENT

The following metrics are used to assess the quality of our numeric prediction models:

- *Correlation Coefficient* (Corr. Coef.) ranges from -1 to 1 and measures the statistical correlation between the predicted values and the actual ones in the test set. A value of 0 indicates no correlation, whereas 1 describes a perfect correlation. Negative correlation indicates inverse correlation, but should not occur for prediction models. We use the Spearman correlation coefficient, as it is independent of the distribution of the underlying data. The correlation coefficient is computed with the following formula:

$$1 - \frac{6 * \sum_i (x_i - y_i)^2}{n * (n^2 - 1)}$$

  where
  $x_i$ are the rankings of the predicted values and
  $y_i$ are the rankings of the actual values.

  The correlation coefficient is our primary performance indicator.

- *Mean Absolute Error* (Abs. Error) is the average of the magnitude of individual absolute errors. This assessment metrics does not have a fixed range like the correlation coefficient (Corr. Coef.), but is oriented on the values to be predicted. In our case the number of defects per file is predicted, which ranges from 1 to 6 and 16 respectively (see Table 6.3 and Table 6.4). As a result, the closer the mean absolute error is to 0 the better. A value of 1 denotes that on average the predicted value differs from the actual number of defects by 1 (e.g. 3 instead of 4). The mean absolute error is computed with the following formula:

$$\frac{|p_1 - a_1| + \ldots + |p_n - a_n|}{n}$$

- *Mean Squared Error* (Sqr. Error) is the average of the squared magnitude of individual errors and it tends to exaggerate the effect of outliers - instances with larger prediction error - more than the mean absolute error. The range of the mean squared error is oriented on the ranges of predicted values, similar to the mean absolute error. But this time the error metrics is squared, which overemphasizes predictions that are far away of the actual number of defects. The quality of the prediction model is good, when the mean squared error is close to the mean absolute error. The formula for mean squared error is:

$$\frac{(p_1 - a_1)^2 + \ldots + (p_n - a_n)^2}{n}$$

## 6.4 EVALUATION

For the evaluation of the predictability of different classes of defects we analyzed the commercial software system (PACS) from our field study described in Section 1.2. For our experiments we investigate 8 months of software evolution in our case study. Figure 6.1 depicts that we use two months of development time to predict the defects of the following two months, which builds up a 4 months time frame. We compare the predictions of pre-release defects with predictions of post-release ones, which results in a period of 8 months. Before the release we create prediction models for defects in general and for defects with high severity. These models can be compared to the ones for post-release defects. After the release date we additionally distinguish defects discovered by internal staff vs. defects reported from the field (customer). With this experimentation set up we test our hypotheses from section 6.1.

### 6.4.1 RESULTS

This section presents the results of our evaluation distinguishing the different cases.

#### SHORT TIME FRAMES

Our analysis focuses on short time frames. To evaluate H1 of Section 6.1 we use two months of development time to predict the following two months. Table 6.1 shows several models predicting pre-release defects where the two months period for defect counting are laid directly before

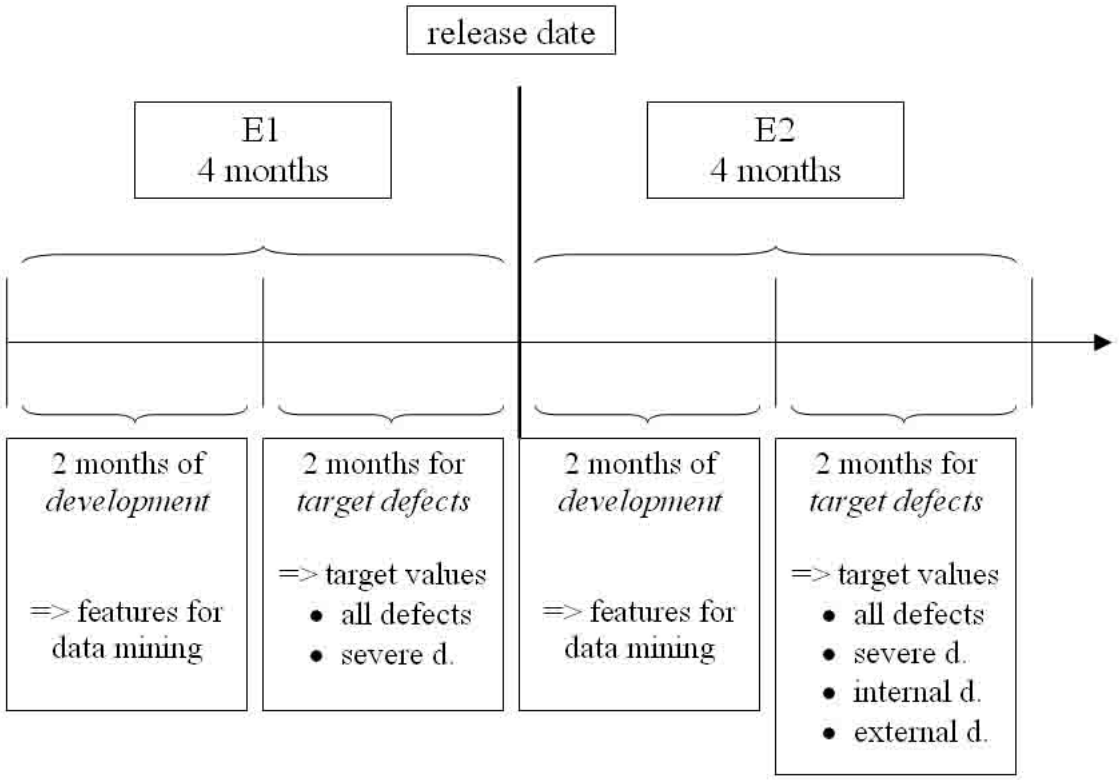



Figure 6.1: Experiment setup. Goal is to compare accuracy of E1 with E2

| | Corr. Coef. | Abs. Error | Sqr. Error |
|---|---|---|---|
| Lin. Reg. | 0.4778 | 0.4604 | 0.7881 |
| M5 | 0.6645 | 0.3602 | 0.6674 |

(a) All defects

| | Corr. Coef. | Abs. Error | Sqr. Error |
|---|---|---|---|
| Lin. Reg. | 0.5012 | 0.1352 | 0.3173 |
| M5 | 0.4844 | 0.0792 | 0.2589 |

(b) High severity defects

Table 6.1: Prediction pre-release defects

| | Corr. Coef. | Abs. Error | Sqr. Error |
|---|---|---|---|
| Lin. Reg. | 0.4985 | 0.9443 | 1.5285 |
| M5 | 0.4959 | 0.7743 | 1.4152 |

(a) All defects

| | Corr. Coef. | Abs. Error | Sqr. Error |
|---|---|---|---|
| Lin. Reg. | 0.5055 | 0.9012 | 1.5151 |
| M5 | 0.5232 | 0.688 | 1.3194 |

(b) Defects discovered internally (through test + development)

| | Corr. Coef. | Abs. Error | Sqr. Error |
|---|---|---|---|
| Lin. Reg. | 0.4675 | 0.3663 | 0.5699 |
| M5 | 0.5266 | 0.2606 | 0.4574 |

(c) Defects discovered externally (through customer + partner companies)

| | Corr. Coef. | Abs. Error | Sqr. Error |
|---|---|---|---|
| Lin. Reg. | 0.4741 | 0.1973 | 0.3175 |
| M5 | 0.449 | 0.1492 | 0.3048 |

(d) High severity defects

Table 6.2: Prediction post-release defects

the release date and the other two months before this two target months are taken to collect feature variables for the prediction models. In Table 6.1(a) we can see that we obtain a correlation coefficient larger than 0.6, which is a quite good correlation. The mean absolute error is low with 0.46 for linear regression and 0.36 for M5 and the mean squared error is also low with 0.79 for linear regression and 0.67 for M5. In order to assess these prediction errors, Table 6.3 describes the defect distribution of the two target months. As mean squared error emphasizes outliers, we can state that the overall error performance of the prediction of all pre-release defects is good.

To confirm our first hypothesis Table 6.2(a) lists the quality measures for the prediction of post-release defects. There the values are not as good as for pre-release defects, but the correlation coefficients are still close to 0.5. Therefore, we confirm H1:

| Failures all severities | Number files | | Failures high severity | Number files |
|---|---|---|---|---|
| 1 | 46 | | 1 | 10 |
| 2 | 11 | | 2 | 2 |
| 3 | 5 | | 3 | 1 |
| 4 | 7 | | 4 | 0 |
| 5 | 2 | | 5 | 0 |
| 6 | 1 | | 6 | 0 |

Table 6.3: Pre-release: Number of files per defect class

| Number failures | Files (all) | Files (internal) | Files (external) | Files high severity |
|---|---|---|---|---|
| 1 | 46 | 30 | 32 | 21 |
| 2 | 21 | 12 | 7 | 1 |
| 3 | 8 | 6 | 1 | 0 |
| 4 | 6 | 4 | 1 | 0 |
| 5 | 5 | 4 | 0 | 0 |
| 7 | 1 | 1 | 0 | 0 |
| 12 | 1 | 1 | 0 | 0 |
| 16 | 1 | 1 | 0 | 0 |

Table 6.4: Post-release: Number of files per defect class

We can predict defects on a short time frames of two months with feature data taken from the predecessing two months.

#### HIGH SEVERITY

Table 6.1(b) shows the results for the prediction models on pre-release defects with high severity. We get the severity level of each defect from the issue tracking system, where the defect reporter assigns severity levels. The quality measures for the predictions of high severity defects differ from the prediction of all defects, because the number and distribution of high severity defects have other characteristics (see Table 6.3). It is interesting that linear regression has a higher correlation coefficient for defects with high severity than for all defects together. M5 can only reach lower value of 0.48 for defects with high severity, which is much lower than the value of 0.65 for all defects. The overall error level is low because of the small defect bandwidth of 0 up to 3 defects for the high severity class.

For the post-release prediction of high severity defects in Table 6.2(d) we can see that the correlation coefficient is even lower with 0.47 for linear regression and 0.45 for M5. The prediction errors are also poorer, which is due to the fact that there are more post-release defects with high severity than pre-release. However, in general we can conclude:

Failures with high severity cannot be predicted with such a precision as overall defects.

#### PRE- VS. POST-RELEASE PREDICTIONS

Our hypothesis H3 states that pre-release defects can be better predicted than the post-release ones. When we compare Table 6.1(a) with Table 6.2(a) we see that our hypothesis seems to be confirmed. The correlation coefficients of linear regression are better for all post-release defects than for the pre-release ones, but all other correlation values are better for pre-release defects. This situation is even more remarkable for M5, as the pre-release correlation coefficient

reaches 0.65 whereas the post-release stays at 0.49. For these prediction models also the two error measures are much higher for post-release.

What about high severity defects? Do we get also better pre-release predictions than post-release ones in the context of defects with high severity? When we look at Table 6.1(b) and Table 6.2(d) we see a similar picture for high severity defects to the overall prediction performance. The correlation values of pre- and post-release defects are closer together for defects with high severity, but overall pre-release defects can be better predicted than post-release ones. This could be because the defects reported from customers are ranked higher than when they are discovered internally, to stress the fact that the defects from customers have to be fixed fast. Therefore, we can confirm the third hypothesis:

> Predictions of post-release defects have higher errors than for models generated for pre-release.

#### INTERNAL VS. EXTERNAL ORIGIN OF DEFECTS

We can see the difference between prediction of defects discovered by internal staff (testers, developers) vs. defects discovered externally (e.g. customer, partner companies) in Table 6.2(b) and Table 6.2(c). The correlation coefficients are very similar for both cases. For linear regression it is higher with 0.51 for internally discovered defects than 0.47 for externally discovered defects. With M5 the results are almost equal. Although it seems that the prediction error is lower for external defects than for internal ones, this result may be caused by the fact that there are no files with many externally discovered defects (see post-release defect distribution in Table 6.4). As a result, we can partly reject H4 and conclude that:

> The predictability of defects that are discovered internally by testers and developers is comparable to the predictability of defects that are reported externally by customers and partner companies.

#### ASPECTS OF PREDICTION MODELS

To analyze the aspects of prediction models in more detail we created two cases using the C4.5 tree classifier: The first model distinguishes between files containing defects and files without defects. The second model separates the files with just one defect from the ones with several defects. The following tree represents the model for pre-release defect-prone files. At each node in the tree a value for the given feature is used to divide the entities into two groups. For each file the tree has to be traversed according to its features to obtain the predicted class: defect-prone vs. non-defect-prone.

The feature with the most information concerning pre-release defect-proneness is the location of the *relativePeakMonth*, which is the month that exhibits the most change events for the analyzed file. Features on the second level are *changeActivityRate* and *authorcount*. As we can see in Figure 6.2 the tree is composed of features from many different categories. *relativePeakMonth* and *changeActivityRate* represent the category of time constraints. *authorCount* and

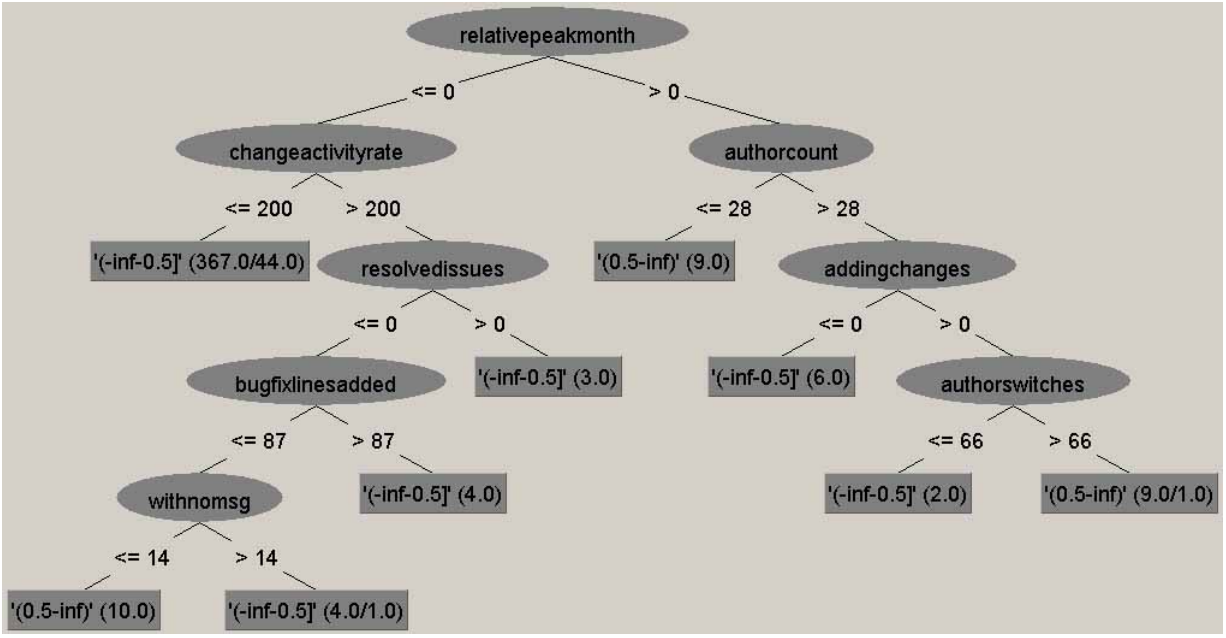



Figure 6.2: Pre-release: with/without defects

*authorSwitches* belong to the team category. The *issueCount*, the number of resolved issues in relation to all issues referenced by source code revisions is an indicator for the process category, similar to the number of *addingChanges* in relation to the overall change count. Also the ratio of revisions without a commit message (*withNoMessage*) describes the process orientation of the development. The number of *linesAddedPerBugfix* provides insight into the development process itself. It is interesting that not size and complexity measures are dominating pre-release defect-proneness but process orientation and discipline of the developers.
Figure 6.3 describes the prediction model to evaluate if a defect-prone file contains more than one defect. This classification tree is much smaller than the previous one for prediction of defect-prone files. Nevertheless, it contains data mining features from many categories. It is interesting that the top level and the bottom level both regard lines edited during bug fixing, but on the first level the *linesAdded* to the file are of interest whereas at the bottom the relational aspect is central with *tLinesDeleted* in all files of common commit transactions. Additionally, the team aspect plays an important role, as the number of *authorSwitches* is the feature on the second level. The model is completed by features indicating the ratio of adding and changing modifications (*changeType*).
From the pre-release classifications we can conclude that:

| Time constraints, process orientation, team related and bug-fix related features play an important role in prediction models of pre-release defects. |
|---|

Figure 6.4 describes the classifier for distinguishing post-release files as defect-prone and non defect-prone. We can see that most information bears the ratio of revisions *withNoMessage*

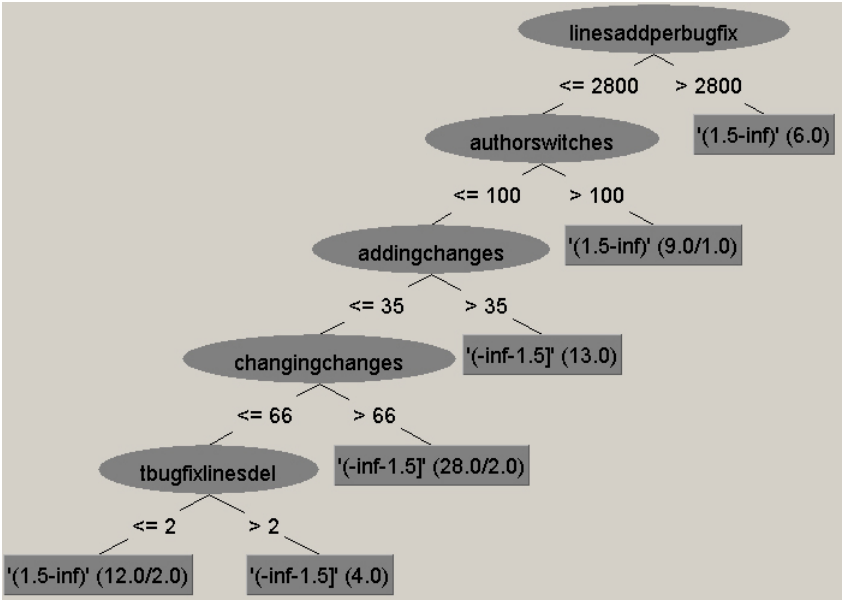



Figure 6.3: Pre-release: one vs. several defects

to the ones with issues related to them (*issueChanges*). The first sub tree is even reduced to this process related feature. The number of changes during the *relativePeakMonth* of each file defines the second level of this tree. Another process oriented feature counting revisions with issues attached (*issueChanges*) is located on the third level. The other feature on the third level is an indicator for the complexity of the existing solution, which is defined as the relative number of changes measured for prediction. Several other features of this tree define the relationship of the analyzed file with other files changed on the same occasions by the same author such as: *coChangedFiles*, *tLinesAdd*, *tLinesDel*, *tLinesChangePerBugfix*.

Figure 6.5 describing the classifier of post-release defect amounts has on top the location of the *relativePeakMonth* within the prediction period. This feature was already an important information source for pre-release defect-proneness. Several other features are new for defect predictions. The average days needed to close major issues seems to provide a general indicator for the efficiency of bug fixing. It is surprising that on this second level the feature fileNew (*coChangeNewFiles*) is located, as it describes if a file was newly introduced in the prediction period. This feature is an indicator of new functionality. We would not expect such a high impact in the post-release prediction of defect rates, because this short period after the release (together with a versioning branch) is probably be used for stabilization and not too often for feature implementation. These new features should be included in the subsequent release and therefore should not influence the defect rate in the current release.

On the other hand it is understandable that *largeChanges* influence code stability. Again surprising is the appearance of the measure related to defect discovery *bugFixesDiscoveredByDeveloper* in this late phase of the software life cycle. We would expect to see this feature in the pre-release phase. Hence, the stabilization for the release seems still to go on after the release date.

In this case study the number of categories contributing to the post-release classifications is even

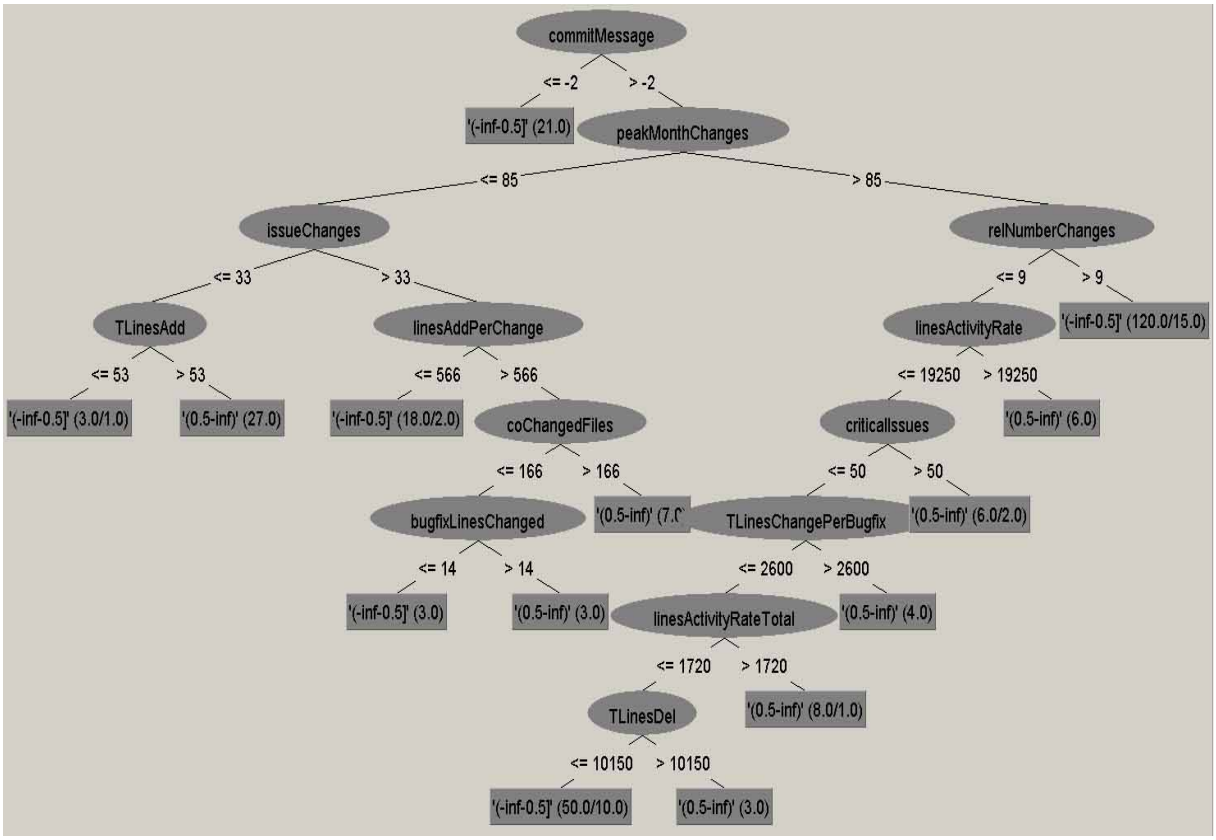



Figure 6.4: Post-release: with/without defects

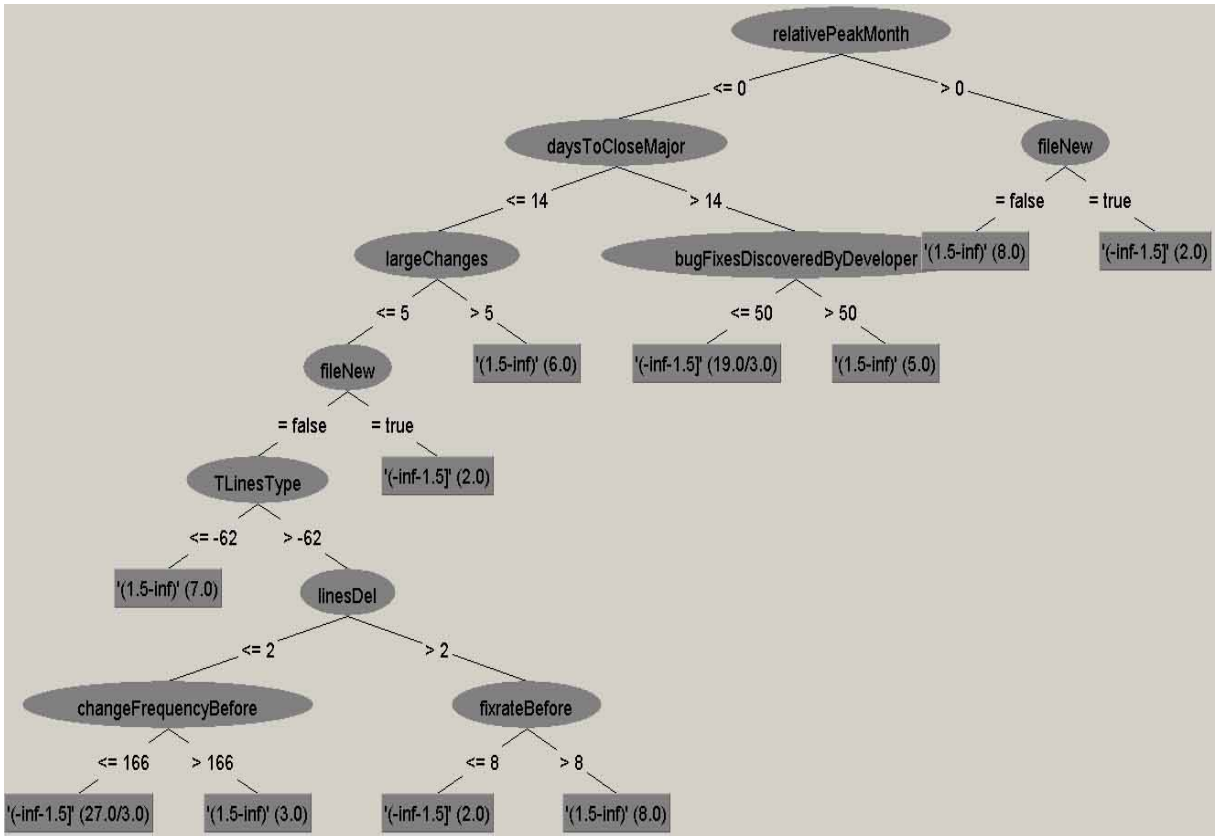



Figure 6.5: Post-release: one vs. several defects

higher than for the pre-release ones:

| Process orientation, time constraints, change amount and timing, relational aspects, as well as defect discovery measures are relevant for post-release defect classification. |
|---|

## 6.5 RÉSUMÉ

This chapter takes the first step towards defect prediction models. It incorporates the data mining features defined in Chapter 3 and focuses on fine-grained issues in the prediction of short term defects. With this prediction approach we address the following research questions from Chapter 1.2:

- Q2: *How do software evolution metrics relate to external software product attributes?* In this chapter, we described a study dealing with fine-grained predictions of defects. We estimate the defect proneness based on a short time frame, where data from two months is used to predict defects of the following two months. With this approach a project manager can decide on the best time frame for release and take preventive actions to improve user satisfaction after a release. Additionally, we compared defect prediction before and after releases of our case study and discovered that in both cases an accurate prediction model can be established. We could show that defects reported by customers need other prediction models than defects discovered by internal staff such as testing. In general we can conclude that software evolution metrics have a strong relationship with software product attributes. Prediction models can be built to predict defects on short time intervals for the next two months with the help of evolution measures from two months of development time.

- Q3: *How can data mining be efficiently utilized in software projects to improve software evolution?* In the experiments of this chapter we use three different data mining algorithms (i.e. Linear Regression, M5 Regression Tree, and C4.5 Decision Tree). The two regression related algorithms predict the exact number of expected defects in a file. The C4.5 Decision Tree distinguishes between two groups of instances, for example between the defect-prone files and the files in which no defects are predicted. Both strategies provide valuable discoveries. In a first step the focus on defect-prone classes can help to structure the verification process more efficiently, where more effort is spent on the classes in which defects are expected. The prediction of the exact number of defects provides a ranking of classes in the order of most expected defects. Classes with multiple predicted defects could be tested thoroughly to identify more than one defect per class. Additionally, the relationship between evolution metrics and defects can help software engineers to understand influences of their work on the results and can help them to define better development processes.

- Q5: *How to set up effective models that result in prediction with high accuracy?* To create well-balanced prediction models we inspected different aspects of software projects. As

size was already used in many other studies it is still an important input for prediction. Other aspects of our approach are the complexity of the existing solution and the difficulty of the problem that the particular piece of software addresses, as they are causes of software defects. We included people issues of different types such as *authorCount* and *authorSwitches* in our analysis to cover another important cause of defects. When a developer has to work on software that somebody else has initially written, mistakes could arise, because he has to understand the design of her colleague. Factors such as author switches are covered by our team features. The discipline of a developer does also influence defect probability. As a result we used indicators for process related issues. Finally, we included time constrains and testing related features into our defect prediction models. The multiple aspects of software evolution enable the creation of accurate prediction models. However, especially features related to process orientation and time constraints had a deep impact on defect prediction. Thus, in the next chapters we will focus on the time and sequence aspect of evolution measures (Chapter 7) and relationship between different event types of evolution processes (Chapter 8), which yields to prediction models with very high accuracy.

# Chapter 7

# Optimizing Predictions with Series Mining

Sequential patterns are important in many domains, because they can be exploited to improve the prediction accuracy of classifiers. A sequence $x = \langle x_1, x_2, x_3, \ldots, x_n \rangle$ of change events during software development contains the information on the course of development additionally to the pure attributes of the sum of all change events describing the state at the final point in time. As one of the first studies we analyze value series of evolution data to create defect prediction models.

Defect prediction models of previous studies often rely on metrics that represent the state of the software system at a defined moment in time (e.g. [30, 55, 70, 87]). For instance such metrics describe the sum of changes implemented in a certain part of the system or are other types of measures such as size and complexity metrics (e.g. [18]).

In the previous chapters we have seen how prediction models can be set up with evolution metrics. However, the previous models regarded time only as one of several aspects that are incorporated in form of metrics into the model. The change over time is an essential aspect of software evolution. Software evolution is a continuous process where different types of activities are applied in a sequential manner on software entities to satisfy customer requirements. Therefore, we describe the change over time as sequences of metrics, where the data points are captured for each day of the training period (two months). Based on this information we create prediction models utilizing genetic programming and linear regression with very high accuracy.

## 7.1 Knowledge Discovery Process

Several consecutive steps are executed in our knowledge discovery process to obtain prediction models based on value series. The basic workflow is as follows:

1. The data collection steps extract evolution data from two systems: versioning systems such as CVS and issue tracking systems such as Jira. Data items taken from different systems have to be assembled into a joined data model to establish an evolution database. Additionally, a relationship is established between data items from a single data source (e.g. co-changed coupling based on commit transactions).

2. The evolution database is used to compute change attributes such as the *number of lines added for bug fixes*, the *number of co-changed files*, or the *number of modifications without a commit message*. These are the characteristics of our data items that are used to create value series for defect prediction. Fenton and Neil [34] pointed out that a sound prediction model has to incorporate different types of attributes. Accordingly, we analyze several types of attributes, where a value series is created for each attribute type. Additionally, series containing attributes of all categories represent the changes over time from a single instance (i.e. a file).

3. We take the value series of evolution attributes as the basis of our defect prediction models. To be able to apply classification algorithms to the value series we extract features describing the relevant characteristics of the value series. Such features can be the *maximum number of co-changed files*. The feature extraction is done automatically with the help of genetic programming, in which several operations are applied on the data points in the value series. The genetic algorithm searches the feature space guided by a fitness function (i.e. the correlation coefficient of our defect prediction models). The best features discovered through genetic programming are the input of the regression algorithms to create the prediction model. The platform for our series mining activities is the YALE machine learning environment [82].

4. We describe the results of a field study, in which we applied the prediction models to several projects taken from three different domains to evaluate the accuracy of the prediction. The following sections describe each step in detail and present our results.

## 7.2 Generating Evolution Series

We focus on the change of metrics of source files during software evolution. For this we measure a set of evolution attributes for each source file over time and compose multiple value series describing the data points of the attributes as a sequence of measures. We investigated the project of our field study (see Section 1.2) where we use two months of development time (series period) to predict the defects of the following two months (target period). The first two months comprise 61 days, where for each day in this series period we measure the attributes for each file. For example the number of lines added within one day is summarized for the data points of this attribute in the value series. As a result many values in the series are zero, as in a development project not all source files are modified every day. The number of defects is predicted for the entire period of the following two months for each source file. Thus, the instances for the prediction models are files. In the following we describe the different evolution attributes and the generation of series in detail.

A definition of generalized series is used for value series: In a series each element $x_i$ is composed of two components. The first is the index describing a position on a straight line (e.g. time); the second is a vector of values. In our case we use two types of vectors: one is a reduced case where only one attribute represents the vector; in the second case the dimension of the vector is given by the number of all evolution attributes used in our approach of series mining.

### 7.2.1 VALUE SERIES

For each day the relative attribute value is computed and added to the value series. For example, we use the number of authors relative to the number of changes on each day in our series period. We give a small sample for the computation of the data points for the number of authors: The sequence 1/1, 0, 2/3, 1/1 would be the result for four days of the series period, when one change is committed on the first day, no changes are done on the second day, two developers implemented a total of three changes on the third day, and one change is committed on the fourth day.

For our series mining approach we use only a subset of the metrics described in Chapter 3, because the computation of the prediction models based on value series is very time consuming. As a result the following metrics are computed for each file and for each day in the series period. We use slightly different names for the metrics in this chapter to show the fact that the metrics have another meaning as this time they are computed for each individual day instead of a sum over all days as described in Chapter 3.

- *LinesAdd*: Lines of code added within a day / Total lines of code until this day.
- *LinesDel*: Lines of code deleted within a day / Total lines of code until this day.
- *ChangeCount*: Number of changes within a day / Total number of changes in the history of the file until this day.
- *Authors*: Number of authors within a day / Number of changes within this day
- *AuthorSwitches*: Number of switches of the author / Number of authors
- *CommitMessages*: Number of different commit messages / Number of changes
- *WithNoMessage*: Number of changes without commit message / Number of commit messages
- *BugfixCount*: Number of bug fixes / Number of changes
- *BugfixLinesAdd*: Lines added for bug fixes / Number of lines added (any type)
- *BugfixLinesDel*: Lines deleted for bug fixes / Number of lines deleted (any type)
- *CoChangeCount*: Number of co-change changes (changes that involve other files) / Number of changes
- *CoChangedFiles*: Number of co-changed files / Number of changes
- *CoChangedNewFiles*: Number of newly introduced files that are co-changed / Number of co-changed files
- *TLinesAdd*: Number of lines added in all co-changed files / Number of couplings
- *TLinesDel*: Number of lines deleted in all co-changed files / Number of couplings

- *TBugfixLinesAdd*: Number of lines added in all files for bug fixes / Number of lines added
- *TBugfixLinesDel*: Number of lines deleted in all files for bug fixes / Number of lines deleted

## 7.3 PREDICTING DEFECTS BASED ON EVOLUTION SERIES

Given the value series of evolution attributes as described in the previous section, the aim of our approach is to derive models for predicting the number of defects in source files. For the model generation we use ”classical” data mining algorithms such as linear regression, which are not able to handle value series in the explicit representation, but can operate on sets of attributes instead of ordered series of values.

Given the ordered values of attributes, how can we use these evolution series as input to data mining algorithms such as linear regression? We generate a new representation of our series information that is suitable for prediction algorithms. This task is called *feature extraction*, where each series is described by a set of relevant characteristics that make different evolution series distinguishable. In a similar manner we could describe a value series containing positions of the sun on earth with the following features: one cycle lasts for 24 hours, the maximum is reached at noon, sunrise and sunset are related with the degree of latitude on earth, etc. Features are then used as input attributes to data mining algorithms such as linear regression.

The feature extraction itself is decomposed into a sequence of basic operators, which delivers a set of features in the end. For example functions returning the minimum, average, or maximum of the values in a series are basic operators. Other basic operators return an index such as the location of a peak value within a given series. Such basic operators have to be combined into a sequence generating the final features for the data mining algorithms. However, the manual selection of an optimal set of operators for feature extraction is a tedious task. Therefore, machine learning can be used to select appropriate operators and group them into an operator tree. The selection of suitable operators is done with the help of genetic algorithms, which is described in the next section.

Thus we have to carry out two learning tasks for our defect prediction:

1. Learning of a set of basic operators for the feature extraction utilizing genetic programming. The resulting features describe relevant characteristics of evolution series for data mining algorithms such as linear regression.
2. Learning a model for defect prediction from the extracted features.

### 7.3.1 FEATURE GENERATION

In the process of feature extraction a number of basic operators is organized into a tree, where each operator uses the output of the predecessor. The output of the operators at the leaves produce

the features of the series. We distinguish two types of basic operators: Transformations and functions:

*Transformations* convert a series into another series. Different types of transformations are available for our defect prediction approach such as filters (e.g. smoothing), frequency transformations (e.g. fourier transformation), generalized windowing, etc. Windowing operators apply a given function on a range of values within the series and additionally slide the window over the series. Others are branches that pass on the interim results to two successing sub-trees.

*Functions* generate single values based on the entire value series and are always the last step of the feature extraction process (i.e. the leaf nodes of the operator tree). Examples of functions are statistics such as average, variance, standard deviation. These functions may be applied on the values themselves or on the indexes of the values, where for example the index of a peak value could be extracted. For an extensive list of transformations and functions see [82].

#### GENETIC PROGRAMMING

The (locally) optimal feature extraction approach (i.e. operator tree of transformations and functions) is elicited with genetic programming utilized on the operator trees.

*Mutations* randomly insert a new operator, delete an operator, replace an operator, or change the parameters of an operator.

*Crossover* switches a sub-tree from one feature description tree by a sub-tree from another tree. According to the standard process of genetic programming the instances with the highest fitness are selected for the next generation.

*Selection* is done based on a tournament between all members of a generation in the genetic algorithm.

*Fitness* of the operator trees for the tournament selection is assessed based on the defect prediction capability of the resulting features. Our fitness function is the regression algorithm itself that is used for the generation of the prediction model. Thus, for each operator tree a regression function is generated based on a training set of a random sample containing 50 evolution series instances and the accuracy of the prediction of defects is used as the fitness value. As a result, the operator trees generating features that predict the defects best are selected for the next generation.

*Initiation* of the first generation in the genetic algorithm is based on 50 operator trees, where the operators are randomly selected from the pool of available transformations and functions.

We limited the maximal number of generations to 8, which yields good results of the predictions and still keeps the time for the computations quite short. We have set the parameters for the generic algorithm higher than the default values given in YALE [82], the machine learning platform we use for our experiments. As a result, the following parameters are defined for our approach: probability of adding a new operator = 0.4 to enable frequent use of new operators; probability of adding a branching operator to create new sub-trees = 0.05, because branches did not prove to increase the performance of the predictions very much and branches make the computation even more time consuming; probability of changing an operator = 0.4, because we

aim to use many different combinations of operators in the genetic algorithm; probability of removing an operator = 0.2, because the number of operators should not decrease to much to give the genetic algorithm more flexibility; probability of performing a crossover = 0.5, exchange of information between feature description trees often improves the predictions; probability of changing a parameter = 0.1, because parameter changes have not proven to change prediction results very much.

### 7.3.2 Applying Data-Mining Algorithms to Series Features

The best features selected by the genetic programming algorithm are used for the creation of the prediction of defects. The primary data mining algorithm for our prediction is linear regression, as our outcome as well as our features from value series are numeric. This is a staple method in statistics where the predicted value is represented by a linear combination of the input attributes (i.e. features) with weights $w_0, w_1, w_2, \ldots, w_n$ and attributes $a_0, a_1, a_2, \ldots, a_n$:

$$x = w_0 + w_1 a_1 + w_2 a_2 + \ldots + w_n a_n$$

The weights are derived from the training data set minimizing the sum of squares of the distance between the predicted value $x$ and the actual one $y$. The distance is summarized for all instances $k$ of the training data set:

$$\sum_k (y - \sum_n w_i a_i)^2$$

The numeric prediction algorithms are used twice in our process. In the previous section we described the generation of features from value series, where the correlation coefficient is used as fitness function for the quality of the feature generation. Finally, we apply the prediction algorithms on the extracted features taking all instances of the training set into account to create the final prediction model.

## 7.4 Evaluation

We evaluated the approach of defect prediction based on series mining with the help of our field study (see Section 1.2), where we use three different real world projects (two open source projects: ArgoUML and the Spring framework, and a commercial PACS system) and analyzed the predictability of defects in the near future.

### 7.4.1 Evaluation Setup

To estimate the accuracy of our defect prediction approach we use the same time periods for all projects, regardless in which development state the project is. In Chapter 6 we have shown that pre-release defects can be better predicted than post-release defects. Thus, if we select the time periods for our evaluation independent of the release dates, we obtain reliable prediction

results that have a high probability to be reproducible within the entire project life time. Only the periods for the commercial system are related with the release dates, because they are the same as in Chapter 6 to be able to directly compare the prediction performance. This gives additional insight into improvement of predictions through the employment of data mining based on value series of evolution metrics. In our approach on series mining we use two periods:

- *Series Period:* November-December 2005. In this period we take evolution attributes from the versioning system and construct value series to represent the flow of the development over time. Each series of the attributes from Section 7.2.1 has a length of 61 days given the two months of the series period. This information is used in our series mining to predict the defects of a source file in the next period.
- *Target Period:* January-February 2006. With our prediction models based on series mining we try to predict the number of all defects in the target period, where the defects are counted based on the information from the issue tracking system and are mapped to files as described in Section 3.1.2.

### 7.4.2 Measuring Prediction Performance

For the assessment of our prediction models we use the following performance measures, which are the ones that we use in Chapter 6:

- *Correlation Coefficient* (Corr. Coef.) measures the statistical correlation between the predicted values and the actual ones.
- *Mean Absolute Error* (Abs. Error) is the average of the magnitude of individual absolute errors.
- *Mean Squared Error* (Sqr. Error) is the average of the squared magnitude of individual errors and it tends to exaggerate the effect of outliers.

As validation method we use 10-fold cross validation to estimate the performance of our prediction models. In this method the set of source files is randomly split into 10 disjoined sets of equal size. The validation is executed 10 times, where the linear regression is trained on 9 of the 10 folds and the remaining one is used to calculate the error rates and the correlation coefficient. After the 10 turns the final performance estimates are generated through averaging the results of the 10 turns.
The validation used two times: First it is used for the assessment of the fitness of the features during genetic programming and finally it is used for the assessment of the prediction models resulting from linear regression with the best features (see Section 7.3).

### 7.4.3 Results

In the following we describe the field study with the selected software projects and discuss performance measures of our prediction models. Furthermore, we investigate the significance of evolution attributes.

| | *Corr. Coef.* | *Abs. Error* | *Sqr. Error* |
|---|---|---|---|
| Commercial system | 0.946 | 0.306 | 0.508 |
| Spring framework | 0.716 | 0.229 | 0.770 |
| ArgoUML | 0.730 | 0.208 | 0.624 |

Table 7.1: Defect prediction with series including all evolution attributes

| Number of defects per file | Commercial system | Spring framework | ArgoUML |
|---|---|---|---|
| 1 | 46 | 80 | 47 |
| 2 | 11 | 15 | 9 |
| 3 | 5 | 3 | 2 |
| 4 | 7 | 2 | 0 |
| 5 | 2 | 0 | 0 |
| 6 | 1 | 0 | 0 |

Table 7.2: Defect distribution

HOW WELL CAN WE PREDICT THE NUMBER OF DEFECTS IN SOURCE FILES WITH SERIES MINING?

To answer this question we take the entire evolution series containing values of all attributes such as *LinesAdd* or *Authors* (see Section 7.2.1). Table 7.1 describes the performance measures of our defect prediction models. The first remarkable number is the very high correlation coefficient of the commercial system from the healthcare domain. A correlation coefficient of 1 would indicate perfect correlation of the prediction with the actual value, where the received 0.946 indicates that very strong prediction models can be built based on evolution series. The other two projects reach a correlation coefficient of more than 0.7, which is still good.

According to the first performance indicator also the mean absolute error of all projects is low. The absolute error has to be measured in relation with the predicted quantities. In our case we predict the number of defects that lie in the range of 0 up to 7. As a result, the measured mean absolute errors of 0.208 to 0.306 are low. The commercial project has a higher absolute error than the two open source projects because it has more files with multiple defects (e.g. 5 or 6 defects), which can be seen in Table 7.2.

The good prediction measures are supported by the mean squared error, which emphasizes outliers more than the mean absolute error. The squared error is lowest for the commercial project with a value of 0.508. This corresponds with the high correlation coefficient and indicates that the prediction is very accurate. However, also the mean squared errors of Spring with 0.770 and of ArgoUML with 0.624 are low. Thus, we conclude:

| Accurate prediction models can be developed based on series mining of evolution data. |
|---|

### WHICH ATTRIBUTES ARE MOST SIGNIFICANT FOR DEFECT PREDICTION?

In the previous section we presented prediction models based on series mining with a very high correlation coefficient and good error measures. These models are created from an evolution series containing all attributes described in Section 7.2. We are interested to find out which attributes are most significant to create accurate prediction models. For this we create prediction models on value series for each single evolution attribute. Table 7.3 presents the correlation coefficients of all generated models, as this performance indicator represents the relationship between the predicted values and the actual ones.

All three projects of the field study exhibit high values for the correlation coefficient on the series containing the number of authors. In the commercial system as well as ArgoUML this single series is even the one with the highest correlation coefficient. For the Spring framework it is only exceeded by the series with *ChangeCounts*, which describes the number of changes per day in relation to total number of changes for this particular file. In the two other projects the *ChangeCount* is ranked only in the middle-field.

*Authors* seems to provide good input to series mining, which contrasts the results of Graves et al. [55]. In our knowledge discovery process we use value series for defect prediction. Therefore, we measure how many authors have implemented modifications to a given file and set this measure in relation to the number of modifications implemented by these authors. We use relative measures, as they have shown to be better predictors than absolute measures [86]. Moreover, we observe the alteration of the number of authors implementing modifications over time, which can provide more accurate data to the prediction models than metrics focusing on a fixed point in time.

Another interesting sub-series is the one containing the number of commit messages in relation to the number of changes. This *CommitMsgs* series has even the second highest correlation coefficient in the commercial project and ArgoUML. In the Spring framework it is on position five with a correlation coefficient of 0.48. We see the commit message as an indicator for the discipline of developers, as we recognized the pattern that developers sometimes tend to reuse the message of the last commit. This work pattern is supported by IDEs that allow one to select a message from a list of the last few messages or to type in a new one before committing changes.

It is quite surprising that the highest performance measures are not reached by size or complexity metrics, but by process and workflow related aspects such as *Authors* and *CommitMsgs*. However, on the third position for ArgoUML and Spring appears the series of *TLinesAdd* (see Table 7.3). This attribute incorporates the number of lines changed within a commit transaction counting added lines of all files that are involved in the transaction. This series reflects an aspect of the interdependency in object oriented software systems, as we take changes to other (related) files within a transaction into account. Contrary, the pure size measure of added lines of a particular file is represented by *LinesAdd*. Although this sub-series plays a remarkable role for the commercial system, it has a very low correlation coefficient in the open source projects. For the sub-series we conclude:

| Projects have different rankings of sub-series, where common aspects can be identified, such as the number of authors or commit messages. |
|---|

| | Commercial system | Spring framework | ArgoUML |
|---|---|---|---|
| | *Corr. Coef.* | *Corr. Coef.* | *Corr. Coef.* |
| LinesAdd | 0.616 | 0.195 | 0.161 |
| LinesDel | 0.305 | 0.111 | 0.234 |
| ChangeCount | 0.517 | 0.653 | 0.268 |
| Authors | 0.946 | 0.628 | 0.760 |
| AuthorSwitches | 0.622 | 0.210 | 0.357 |
| CommitMsgs | 0.943 | 0.480 | 0.459 |
| WithNoMsg | 0.273 | 0.008 | -0.054 |
| BugfixCount | 0.455 | 0.290 | 0.253 |
| BugfixLinesAdd | 0.437 | 0.294 | 0.295 |
| BugfixLinesDel | 0.736 | 0.319 | 0.244 |
| CoChangeCount | 0.548 | 0.336 | 0.388 |
| CoChangedFiles | 0.481 | 0.240 | 0.409 |
| CoChangedNew | 0.426 | 0.171 | 0.233 |
| TLinesAdd | 0.598 | 0.622 | 0.442 |
| TLinesDel | 0.586 | 0.579 | 0.225 |
| TBugfixLinesAdd | 0.482 | 0.318 | 0.362 |
| TBugfixLinesDel | 0.460 | 0.319 | 0.296 |
| series of all attributes | 0.946 | 0.716 | 0.730 |

Table 7.3: Correlation coefficients of series with a single attribute and the summarizing series including all attributes

In defect prediction based on value series of evolution metrics the number of authors and the number of commit messages are significant for defect prediction and are superior to size metrics such as the number of lines added or complexity measures such as number of bug fixes.

## 7.5 RÉSUMÉ

In our approach an entire series of measurements is used to predict a single label (i.e. the number of defects in a file containing object-oriented entities). The series of values describes the flow of development over time and improves the prediction in relation to fixed metrics describing a particular point in time. Thus, we get an impressive improvement in the accuracy of our predictions in comparison to Chapter 6. In this chapter, we especially focused on the following research questions from Chapter 1.2:

- Q1: *How to set up metrics from sources such as modification reports and process/project management tools?* Daily data points of our evolution attributes (see Section 7.2) were captured over a period of two months to predict the defects in the subsequent two months in a project. The data points represent the development within the entire day. If several changes are committed to a file within one day then the attribute values are summed up and result in one data point of the evolution series. As we establish series for each file, many data points are zero. Nevertheless, the series of many zero values also contain much information. For example it is possible to extract how many non-zero values are in a series or how many zero values on average are between two expressive values. As a result, value series of evolution attributes provide valuable information for project assessment, where we exploited it for one possible scenario: the prediction of software defects.

- Q2: *How do software evolution metrics relate to external software product attributes?* In contrast to other studies that employed size or complexity measures as predictors, we have identified the number of authors and the number of commit messages to versioning systems as excellent predictors of defect densities. This indicates that additional to product related aspects we should take into account process related ones that are sometimes called ”soft issues”. The teamwork in software projects is based on our results important for software quality. Not only the productivity of a team is strongly dependent on its internal structure, but also the quality of the product.

- Q5: *How to set up effective models that result in prediction with high accuracy?* Our results showed that evolution series are excellent predictors of defect densities. We described an analysis focusing on sub-series, where the prediction models based on series of a single variable are sometimes even superior to the overall model. An interesting proponent of this category is the number of authors, where good models can be created (up to a correlation coefficient of 0.946). These high accuracy is reached through the combination of linear regression with genetic programming. First the genetic algorithm selects feature extraction

methods from a pool of basic operators such as transformations and functions. Based on the best features a prediction model is set up with the help of linear regression. An important factor in this approach is the input of evolution series, where samples of evolution attributes are taken per day.

# Chapter 8

# Process Events for Quality Assessment

In Chapter 5 we could show that the commit messages software engineers enter when submitting changes to a software repository provide important information about refactoring activities [104]. Now, we show how we can use the information within commit messages to recover different types of events during software evolution and utilize the ratio between different types for software defect prediction. We address the question: What is the right time/ratio of different change types for software quality assessment?

Reading a lunar calendar you may have come across predictions like: ”When the planet Jupiter stays in the star constellation of Aquarius and afterwards Pluto crosses the path of Neptune it is the best time to bake a delicious cake for your wife.” or ”When Mars is in line with Jupiter then during the following full moon it is finally time to cut down a tree in the garden.” These statements have in common that they expect a certain series of events that is the optimal precondition for some following actions.

In software projects we have similar lunar cycles, which refer to the different iterations and phases during development and evolution of software systems. To create a high quality product we should follow a defined series of actions. A lunar calendar gives advise for the right time for certain actions to reach the best result. What is the right time in software development processes for one type of a modification or another? The software life cycle and its stages were subject to many discussions. One of the models defining the different steps software passes is the ”staged model for the software life cycle” [99]. After delivering the software to the customer several different types of actions are performed on the system [94]:

- Corrective: This includes the fixing of bugs discovered during the use of the system.
- Perfective: Perfective maintenance aims to improve certain attributes of the software, like the performance, the logging facilities, or the administration.
- Adaptive: This type of activities supports the coping with changing environment.
- Preventive: These changes combine all actions necessary for improvements to avoid future problems. Refactoring is one such type of changes in this area [38].

## 8.1 EVENTS OF THE SOFTWARE EVOLUTION PROCESS

To analyze software evolution processes we have to recover the events during the process execution. To accomplish this task we utilize information provided by software engineers within their daily work when implementing changes during software evolution. Text messages are inspected for recurrent chunks of information and then word vectors are created to represent the content of each piece of text. Afterwards, we use clustering techniques as a method in data mining for unsupervised learning task to divide the word vectors into groups of different text messages. Each group describes a particular type of process events applied during the workflows of the software project. We applied our approach for event extraction and defect prediction on our field study consisting of three different software projects as described in Section 1.2.

In our approach we use three time periods for the different data elements:

- *Event Period.* During this period the commit messages are evaluated and event types are recovered. In our field study this period is May 2005 - May 2006.
- *Series Period.* We use the evolution events within this period to generate value series representing relationships between events of different type (see Section 8.2). This lays the foundation for the defect prediction models. This period is Sep.-Dec. 2005 in our study.
- *Target Period.* This is the time frame immediately after the series period, where we count the number of defects we want to predict. In our study we use Jan.-Feb. 2006.

### 8.1.1 WORD VECTORS FROM COMMIT MESSAGES

The commit messages taken from the CVS provides information from the developers about the applied modifications. We decompose the text from all messages into their word or parts of words. For this task we use the word vector tool (WVTool) of the YALE machine learning environment [82]. We apply statistical language modeling, which allows us creating word vector representations of text documents in the vector space. Each commit message is represented by a vector of terms, which are entities such as entire words or parts of words reduced to some linguistic base form. Additionally, abstract concepts are used, as for example any number in a text is described by the term ”number”.

The resulting word vector for a commit message contains the weights describing the relevance of the word according to the following formula:

$$w_{ij} = \frac{f_{ij}}{fd_j} \log(\frac{|D|}{ft_i})$$

where
$w_{ij}$ is the weight of the term $i$ in document $j$,
$f_{ij}$ is the number of occurrences of term $i$ in document $j$,
$fd_j$ is the total number of terms occurring in document $j$,

| cluster name | ArgoUML | Spring | Commercial |
|---|---|---|---|
| cluster c0 | 459 | 327 | 6 |
| cluster c1 | 194 | 964 | 57 |
| cluster c2 | 435 | 138 | 136 |
| cluster c3 | 170 | 602 | 65 |
| cluster c4 | 341 | 162 | 767 |
| cluster c5 | — | 287 | 183 |
| cluster c6 | — | 200 | 397 |
| cluster c7 | — | — | 3 |
| cluster cx | — | — | 1 |
| $\sum$ | 1599 | 2680 | 1615 |

Table 8.1: Number of distinct commit messages per event type (cluster).

$|D|$ is the total number of documents, and
$ft_i$ is the total number of documents in which term $i$ appears at least once.

Each commit message is counted only once for the creation of the process model. As a result, the term are not weighted higher, if a developer uses the same message for several commit activities.

### 8.1.2 Event Types: Clustering Word Vectors

The set of commit messages supplied by software engineers is converted according to the description of the previous section into word vectors representing the relevance of the terms within the message. Thus, each instance is composed of a list of numerical attributes, which is the input to a clustering algorithm. It groups similar items that seem to belong together. Clustering is an unsupervised learning technique in contrast to classification and regression, which we use for defect prediction. As we do not want to prescribe a given number of clusters (i.e. event types), we apply the expectation maximization (EM) clustering algorithm. The expectation is described by the probability belonging to each of the clusters, which is assigned by the EM algorithm to each instance. Then 10-fold cross validation is used to decide how many clusters to create with the goal to maximize the likelihood of distributions. The algorithm works in the following steps:

1. Only one cluster is created.
2. The set of instances is randomly split into 10 folds.
3. The EM algorithm is performed 10 times on the 10 folds to generate the probabilities.
4. The loglikelihood is averaged over all 10 results.
5. If the loglikelihood has increased the number of clusters is increased by 1 and execution is continued at step 2.

| cluster name | ArgoUML | Spring | Commercial |
|---|---|---|---|
| cluster c0 | 2389 | 4459 | 46 |
| cluster c1 | 754 | 5866 | 187 |
| cluster c2 | 1226 | 918 | 874 |
| cluster c3 | 599 | 3265 | 66 |
| cluster c4 | 1052 | 755 | 5724 |
| cluster c5 | — | 1825 | 792 |
| cluster c6 | — | 1248 | 1734 |
| cluster c7 | — | — | 6 |
| cluster cx | — | — | 10186 |
| $\sum$ | 6020 | 18336 | 19615 |

Table 8.2: Number of file changes per event type (cluster).

With the help of the clustering and validation algorithm we divide the commit message of the select software projects into different event types. Based on the probability measures of the EM algorithm we obtain 5 event types (=clusters) for ArgoUML, 7 for the Spring framework and for the commercial system we could identify 9 clusters, where the cluster *cx* summarizes all modifications for which no commit message was supplied. It is interesting that only the commercial system contained modification without commit message in the inspected periods. Table 8.1 shows the distribution of distinct messages on the event types for each project. The number of messages accumulated into a particular cluster vary widely, as for example for the commercial system the EM algorithm created two groups with less than 10 messages (*c0*:6, *c7*:3) whereas other clusters contain several hundred instances. However, the number of clusters identified by the EM algorithm is related with the diversity of the text in the commit messages. Thus, in the commercial project the developers use more different terms to describe their work. This could be based on the number of people that is certainly higher for this project than for the open source projects.

In contrast to that, Table 8.2 presents for each project the number of file modifications (i.e. revisions) for each event type. When comparing the two tables we can see that there seems to be correlation between the number of modifications and the number of messages. There are only a few outliers such as cluster *c3*, where 66 modifications where committed with 65 different messages.

### 8.1.3 Resulting Term Frequencies

Based on the word vector representation of the commit messages and the EM clustering algorithm we have subdivided the modifications of software engineers into different event types. These types are analyzed based on the most frequent terms within each group. The following list shows for each project and each event type the top seven terms:

**ArgoUML**
+ *c0*: remov (166), issu (111), method (66), class (53), test (49), fix (48), model (46)
+ *c1*: issu (50), model (35), event (35), chang (31), number (26), make (23), action (23)
+ *c2*: issu (199), fix (91), stereotyp (71), diagram (56), multipl (50), except (47), allow (46)
+ *c3*: comment (57), issu (45), make (26), chang (25), improv (23), text (22), updat (20)
+ *c4*: issu (89), number (64), chang (55), review (43), name (43), code (43), delet (40)

While focusing on ArgoUML we can see that there are certain terms that appear in each group such as ”issu”. Thus, the developers within this project seem to follow a defined process model, where they provide a reference to the issue tracking system in many commit messages. For the first cluster *c0* the terms such as ”method”, ”class’, and ”model” could be references to the domain of the project itself instead of the actual modification, because ArgoUML as a UML tool is strongly related with design and development support. The remaining terms ”remov”, ”test”, and ”fix” probably point to the direction of the intension of the modification. Thus, the terms of *c0* relate to testing and bug fixing.

The second event type *c1* is not related to corrective actions, but the terms ”chang” and ”make” indicate that these are some kind of enhancements. *c2* contains similar to *c0* terms describing fixes such as ”fix” and ”except”. In contrast to *c0*, *c2* points to the repairing of the exception handling. Preventive actions are implemented through *c3* based on the terms ”comment”, ”chang”, ”improv”, and ”updat”. The terms ”chang”, ”review”, and ”delet” of *c4* indicate enhancements similar to *c1*. However, *c1* seems more to be adaptive and to add properties whereas *c4* is more perfective with reviews and cleanup activities.

**Spring framework**
+ *c0*: ad (101), test (92), fix (80), method (47), polish (45), work (35), action (25)
+ *c1*: ad (205), support (118), updat (69), remov (69), class (54), spring (40), move (37)
+ *c2*: introduc (30), synchron (20), ad (19), factor (17), view (16), access (15), refactor (15)
+ *c3*: ad (116), method (114), test (73), properti (62), except (49), check (49), flow (48)
+ *c4*: javadoc (76), correct (46), fix (45), typo (32), minor (28), method (20), error (18)
+ *c5*: name (64), support (42), ad (36), rework (35), chang (34), paramet (34), refin (33)
+ *c6*: ad (79), javadoc (51), file (37), java (33), renam (31), branch (23), mbranch (23)

For the Spring framework the first event type refers to corrective actions with the terms ”ad”, ”test”, ”fix”, and ”polish”. *c1* indicates a clean up attitude with the terms ”support”, ”updat”, ”remov”, and ”move”. In addition, the term ”ad” indicates improvements in general. The third event type *c2* of the Spring framework an interesting element, because it explicitly references refactoring with the term ”refactor”. Based on a detailed inspection of the commit messages the term ”factor” refers to factoring-out, factories, and factory. Thus, together with the refactoring are also other programming improvements introduced within the scope of *c2*. Similar to *c2* of ArgoUML, *c3* of the Spring framework is related to exception handling. But this time the exception handling is improved preventively based on the terms ”ad”, ”test”, ”check”, and ”flow”.

An interesting event type is summarized by *c4*, as it combines improvements in understandability (”javadoc”, ”correct”, and ”typo”) and also improvement of the stability (”fix”, ”method”, and ”error”). The terms ”support”, ”rework”, ”chang”, and ”refin” describe that *c5* implements adaptations to changing requirements on the system. *c6* focuses on administrative tasks with the terms ”branch” and ”mbranch”. Additionally, ”javadoc” and ”renam” indicate improvements of the documentation and readability.

**Commercial system**
+ *c0*: chang (5), java (4), fix (3), prepar (1), compil (1), enum (1), sort (1)
+ *c1*: handl (21), updat (16), file (14), error (11), call (6), pjvi (6), ad (5)
+ *c2*: ad (97), plugin (24), pjvi (23), imag (15), chang (11), fix (7), layout (7)
+ *c3*: file (65), java (65), initi (65), ad (65), branch (65), head (10), rawdatacontain (2)
+ *c4*: chang (101), pjvi (87), fix (51), remov (48), refactor (34), toolbar (33), call (33)
+ *c5*: fix (144), pjvi (98), problem (43), dialog (19), command (6), updat (6), solv (5)
+ *c6*: pjvi (175), make (65), view (43), work (35), ad (35), herd (34), load (29)
+ *c7*: chang (3), click (2), plugin (2), herd (1), allow (1), remove (1), segment (1)
+ *cx*: ”no commit message”

The reference to the issue tracking system is this time denoted by the term ”pjvi”, which is the project identifier, instead of the term ”issu”. It is interesting that in the commercial system the term ”fix” appears in the context of several event types. *c0*, *c2*, *c4*, and *c5*. The group of *c0* is very small with 46 modifications to files and is related to work-in-progress based on the terms ”chang” and ”prepar”. Also the event type *c1* references to problems with the term ”error”. Further, improvements in general are indicated by the terms ”handl” and ”update”. Although the third type *c2* contains the term ”fix” it seems more related to functional extensions with ”ad”, ”plugin”, and ”chang”.

*c3* is the first element not referring to problems, but it focuses on administrative tasks, as ”branch” and ”head” are common words in the context of CVS. Additionally, ”file”, ”java”, and ”initi” support the administrative nature of *c3*. Although *c4* contains the term ”fix” it probably is related to the preventive side of the actions based on the terms ”chang”, ”remov”, and ”refactor”. Contrary to *c4*, *c5* is the classical corrective activity with the terms ”fix”, ”problem”, ”updat”, and ”solv”. The event type *c6* is focused on the constructive work with functionality (”make”, ”work”, ”ad”). *c7* is the smallest group with only six modifications. It seems related with the GUI based on the terms ”click” and ”plugin”, because many graphical extensions are implemented through plug-ins in this architecture. For the last event type *cx* we cannot find any statements on the content, as no commit message was provided by the developers for these modifications.

## 8.2 Defect Prediction based on Evolution Events

After extracting events from software evolution data, we create a prediction model for software defects. First a value series is created representing ratios between pairs of event types. The series

are generated by accumulating the number of events per type and then by pairwise dividing the sum of events. These value series of event ratios are input to the construction of the prediction models. With the help of genetic programming we identify the best set of transformations and functions that create an abstract description of the value series. The resulting abstraction of the value series provides the features, which are input attributes for data mining algorithms. We use linear regression as basic algorithm to set up our defect prediction models that are based on events of software evolution.

### 8.2.1 VALUE SERIES OF EVENT RATIOS

The foundation of our prediction models are series of event ratios, where we use the event reconstruction described in Section 8.1. For the series we add up the occurrences of each event type on each day. For the whole series period (see Section 8.1) we obtain the number of events of a particular type from the beginning of the series period until the day of interest within the period. Thus, the events happened on a particular day are added to the number of events already accumulated before that day.
For each day the ratios of the number of events are computed pairwise between all different event types. For example for ArgoUML the clustering algorithm identified five different event types (*c0*, *c1*, *c2*, *c3*, and *c4*). These are used to generate the following event ratios: *c0/c1*, *c0/c2*, *c0/c3*, *c0/c4*, *c1/c2*, *c1/c3*, *c1/c4*, *c2/c3*, *c2/c4*, and *c3/c4*. All ten ratios are used to create ten value series, where for each day of the value of the ratio is entered into the particular series.

### 8.2.2 EVALUATING PREDICTION PERFORMANCE

For the assessment of our prediction models we use the following performance measures, which are the ones that we use in Chapter 6:

- *Correlation Coefficient* (Corr. Coef.) measures the statistical correlation between the predicted values and the actual ones.
- *Mean Absolute Error* (Abs. Error) is the average of the magnitude of individual absolute errors.
- *Mean Squared Error* (Sqr. Error) is the average of the squared magnitude of individual errors and it tends to exaggerate the effect of outliers.

As validation method we utilize 10-fold cross validation to estimate the performance of our prediction models.

### 8.2.3 PREDICTION RESULTS

The defect prediction based on evolution events is based on the series mining described in Chapter 7. The value series of event ratios from the previous section is used as an input to the data mining technique for the generation of prediction models.

| | *Corr. Coef.* | *Abs. Error* | *Sqr. Error* |
|---|---|---|---|
| ArgoUML | 0.843 | 0.218 | 0.457 |
| Spring framework | 0.681 | 0.189 | 0.473 |
| Commercial system | 0.581 | 0.237 | 0.475 |

Table 8.3: Performance measures of defect prediction based on value series of evolution events.

As we see in Table 8.3 all three projects from our field study show good results on defect prediction. The correlation coefficient is the performance measure with the largest variation between different projects. The lowest value was obtained with the commercial software system. However, even a correlation of 0.581 is good in comparison to previous results of Chapter 6 where we could only reach a maximal correlations of 0.6. The Spring framework already reaches a correlation of 0.681 and the best one was obtained based on the data of ArgoUML with a value of 0.843.

In addition to these good correlation results, the error levels are low. The mean absolute error is always below 0.25, which means that on average the predicted number of defects for a file is only one quarter away from the actual value. This means that if all predictions would be just one defect above or below the real number of defects, three of four predictions would be precise and just one would miss the correct number of defects. The results of the mean squared error emphasize the good prediction accuracy of our approach, as they exaggerate outliers more than the mean absolute error. The values of the mean squared error of all three projects in our field study are between 0.45 and 0.5, which are low measures. The mean absolute errors together with the mean squared errors indicate that many predicted values are equivalent with the actual ones and the predictions the miss the exact number of defects provides predictions that are very close.

Regarding the good prediction measures we conclude:

| Software defects can be accurately predicted based on event ratios from evolution processes. |
|---|

## 8.3 Event Assessment

Let us now take a look at separate series of event ratios and their predictive power. The correlation coefficient describes the relationship between the predicted and the actual values. Therefore, we use this performance measure to estimate which separate series can accurately predict the defects in files.

For our defect prediction we use a mining approach containing two steps. This approach can be applied on individual series as well as on the entire set. Table 8.4 presents the correlation coefficient for each value series of event ratios. It is interesting that a few individual series can outperform the entire set of value series measuring the prediction based on the correlation coefficient. We can see that ArgoUML and the commercial system reach high correlation values of more than 0.8 and even more than 0.9. The event series of ArgoUML seem to be particularly

| Corr. Coef. | ArgoUML | Spring framework | Commercial PACS |
|---|---|---|---|
| $> 0.3$ | c0 / c2 | c0 / c2 | |
| | c0 / c4 | | |
| $> 0.5$ | c0 / c1 | c2 / c3,...,c6 | c0 / c1,c2,c3 |
| | | c3 / c5 | c0 / c5,c6,c7 |
| | | c4 / c6 | c1 / c2,...,c7,cx |
| | | | c2 / c3,...,c7 |
| | | | c3 / c4,...,c7,cx |
| | | | c4 / c6 |
| | | | c5 / c6,c7,cx |
| | | | c6 / c7,cx |
| $> 0.6$ | | c0 / c1 | c0 / cx |
| | | c0 / c3,...,c6 | c2 / cx |
| | | c1 / c2,...,c5 | c4 / cx |
| | | c3 / c4 | c7 / cx |
| | | c3 / c6 | |
| | | c4 / c5 | |
| | | c5 / c6 | |
| $> 0.7$ | c0 / c3 | c1 / c6 | |
| | c1 / c3 | | |
| $> 0.8$ | c1 / c2 | | c4 / c7 |
| | c1 / c4 | | |
| | c2 / c3,c4 | | |
| $> 0.9$ | c3 / c4 | | c0 / c4 |
| | | | c4 / c5 |

Table 8.4: The correlation coefficient of individual ratios between different event types.

| | ArgoUML | Spring | Commercial |
|---|---|---|---|
| cluster c0 | 907 | 1370 | 34 |
| cluster c1 | 190 | 1645 | 104 |
| cluster c2 | 392 | 358 | 355 |
| cluster c3 | 214 | 1377 | 13 |
| cluster c4 | 341 | 233 | 1595 |
| cluster c5 | — | 608 | 157 |
| cluster c6 | — | 543 | 331 |
| cluster c7 | — | — | 0 |
| cluster cx | — | — | 196 |

Table 8.5: Number of file changes per event type (cluster) within the series period (Sep.-Dec. 2005).

suitable for defect prediction. Only most series in relation with the event type *c0* have lower correlation of 0.3 and 0.5, although this event type is related with bug fixing (see Section 8.1.3) and most changes in the series period (see Table 8.5) are assigned to this type. The highest correlation can be reached with the ratio *c3/c4*, where *c3* is related to improvements such as comments and *c4* represents enhancements related to the terms ”chang”, ”review”, and ”delet”.
For the commercial system the relations *c0/c4*, *c4/c5*, and *c4/c7* exhibit high correlation coefficients. From Section 8.1.3 we know that *c4* is related to refactoring and *c0*, *c5*, and *c7* are related to corrective actions. Refactoring seems to provide valuable information for defect prediction, as the related event type *c4* has still a high correlation of more than 0.6 with the event type *cx*, which represents unknown modifications.
For the Spring framework the only value series that can reach a correlation coefficient of more than 0.7 is *c1/c6*. The event type *c1* refers to clean up attitude with terms such as ”support”, ”updat”, ”remov”, and ”move”. *c6* supports the relevance of understandability for defect prediction, because it contains the term ”javadoc” and ”renam”.
Based on the results of all three projects we draw the following conclusion:

| Perfective actions with related terms such as ”refactoring”, ”comment”, ”javadoc”, and ”renam” are important input to defect prediction. |
|---|

## 8.4 RÉSUMÉ

This chapter focuses on the software evolution process as an input to project assessment. We identify the type of changes based on the commit messages of developers and create defect prediction models on top of this information. Our approach based on software evolution guides project managers on the right time for the release dates of a software product, which is best when the expected number of bugs is low in the near future. The following research questions are particularly addressed in this chapter:

- Q1: *How to set up metrics from sources such as modification reports and process/project management tools?* We developed an approach for the reconstruction of evolution events that describe the intensions of software engineers while implementing modifications on a software system. This data extraction is done with the help of a data mining technique called clustering, in which groups of examples are sought that seem to belong to each other based on their attributes. First the commit messages of developers are taken apart into the contained terms, where similar words such as ”comments” and ”commenting” are put together in one term. The terms and their frequencies are used to put each commit message into a particular cluster. The number of clusters then describes the different event type in the reconstructed evolution process. For the defect prediction we arrange the events into value series, where the number of the events are recorded for each day. To be exact we use ratios between two event types at a time to establish the value series. For example one such series describes the course in the development between perfective and preventive evolution activities.

- Q3: *How can data mining be efficiently utilized in software projects to improve software evolution?* The results of our fine-grained analysis of event ratios indicate that activities for perfective maintenance are important for defect densities. The highest correlation values could be reached with event types containing terms such as ”refactoring” and ”comment”. This is an important message for managers to invest more in ”soft aspects” such as well-designed software and not only one that implements all requested features. Additionally, comments indicate that understandability is crucial for defects and the resulting maintenance cost. It is interesting that both commercial as well as open source projects of our case study exhibited the most accurate prediction models when focusing on these ”soft issues”. From the developers perspective especially commenting source code seems to be a very tedious task, but it definitely pays of when looking at the resulting defects. We already discovered in Chapter 7 that discipline on the process is an important attitude in the work of software engineers. This again is emphasized by the results of the current findings, as regular refactoring and commenting are very much dependent on discipline.

- Q4: *How can models on evolution processes be effectively adapted for the assessment of software projects?* We conducted one of the first studies using data about software evolution events as an input for defect prediction. Software processes are usually defined up front to guide development projects. We do an retrospective reconstruction of process events from change descriptions provided with commits of modifications. Then we tear the evolution process apart and regard ratios between pairs of event types and analyze them as a value series over time. Only the information about the type of change is sufficient to predict defects of the resulting products. In the experiments of the current chapter we do not integrate any other attributes such as size or complexity measures. The pure evolution process is investigated to assess the resulting product.

- Q5: *How to set up effective models that result in prediction with high accuracy?* To create accurate prediction models we utilize data mining techniques related to sequential data, which makes predictions possible with high correlation coefficients of more than 0.8 and low error measures. The extracted information about the evolution process is organized into value series to represent the trends in the project over time. This enables the creation of prediction models with high accuracy. Although in this chapter we do not reach the impressive numbers of Chapter 7, it is still surprising that the type of events and the time of their appearance are sufficient to obtain good predictions.

# Chapter 9

# sPACE - Discussion

In this thesis we approached the topic of software project assessment from different angles. Table 9.1 provides an overview on the research questions raised in Chapter 1.2 and the chapters of this thesis that address them.

This thesis contributes to the body of knowledge in software evolution in several ways. First, it defines a large set of evolution metrics. Thus, the question Q1: *How to set up metrics from sources such as modification reports and process/project management tools?* is answered especially in Chapter 3, where we describe the data gathering and processing steps. The co-change coupling as an important element of evolution measures is exploited in Chapter 4. As a next step, Chapter 7 adds a new dimension in evolution metrics. As software evolution is a continuous process over time, we also incorporate the timing and trend aspect in our measures. This is refined in Chapter 8, where we extract event types of software evolution and investigate the ratios between different types over time.

In Chapter 4 we could show that co-change coupling is able to describe software attributes such as evolvability to answer Q2: *How do software evolution metrics relate to external software product attributes?* Co-change smell was introduced to describe undesired evolution patterns that should be handled by software engineers to improve productivity. However, refactorings themselves can be predicted based on information about software evolution, which is described in Chapter 5. In a contradictory view refactoring could be described as additional re-work that the customer has to pay for. Our prediction models of refactoring allow an understanding of the influences for the necessity of such changes. Moreover, Chapter 6 presents an analysis of several sub-categories of software defects and describes the differences in the prediction models. This provides an in-depth view on the relationship of evolution attributes and defect occurrences. This analysis is rounded off by Chapter 7 that presents a study on different evolution series, where the number of authors and the number of commit messages are important inputs to defect prediction.

Evolvability is explicitly worked out in Chapter 4 to answer question Q3: *How can data mining be efficiently utilized in software projects to improve software evolution?* The identified locations of change smells are treated with refactoring to obtain a smoother evolution. The research question is answered in Chapter 5 and Chapter 6, which enable a better understanding of the influences of evolution measures on software attributes such as the necessity for refactoring and

| | Q1 Metrics | Q2 External Attributes | Q3 Improved Evolvability | Q4 Process Assessment | Q5 Prediction Accuracy |
|---|---|---|---|---|---|
| Chapter 3 Evolution Metrics | X | | | | X |
| Chapter 4 Change Smells | X | X | X | | |
| Chapter 5 Design Deficiencies | | X | X | | |
| Chapter 6 Defect Prediction | | X | X | | X |
| Chapter 7 Series Mining | X | X | | | X |
| Chapter 8 Process Events | X | | X | X | X |

Table 9.1: Relationship between research questions and corresponding chapters

the probability of defects. Evolvability can be improved when keeping the influences in mind while developing software systems. A higher level is described in Chapter 8, where we relate software process events with the resulting number of defects.

Q4: *How can models on evolution processes be effectively adapted for the assessment of software projects?* This question is particularly answered in Chapter 8. There we reconstruct events of the evolution process and use this information as an input to the prediction of defects. Value series of pair-wise ratios describe the balance within the evolution process that leads to certain product attributes such as an increased number of defects. It is interesting that in all projects of the case study soft issues related to the terms ”refactor” and ”comment” have the highest impact on defect prediction.

Within the entire thesis the question Q5: *How to set up effective models that result in prediction with high accuracy?* is of major interest. We have laid the foundation with the establishing of a large number of evolution measures in Chapter 3, which are extensively used in Chapter 6 to create our first defect prediction models. Our basic approach is refined in Chapter 7 where the explicit focus is directed on the timing aspect in software projects. This enables the creation of the best prediction models in this thesis. Finally, Chapter 8 uses only the information about the types of changes without data describing the details to predict defects. It is astonishing that with this input accurate prediction models can be constructed. This should raise the attention of software engineers on software processes to develop products of desired characteristics.

We evaluated the techniques of sPACE in several case studies. Table 9.2 shows the time periods that where used for the different experiments throughout this thesis. Our first approach to project assessment based on co-change coupling that was presented in Chapter 4 demanded the longest time period. In this chapter, we analyzed change smells as an indicator for the need of refactoring. To spot such locations in a system, we had to investigate long time periods (15 months).

| | Analyzed Project | Input Period | Target Period |
|---|---|---|---|
| Chapter 4<br>Change Smells | PACS | Jan. 2002 - Mar. 2003<br>(15 months) | Apr. 2003 - Jun. 2004<br>(15 months) |
| Chapter 5<br>Design Deficiencies | ArgoUML<br><br>Spring framework | Jul/Oct. - Dec. 2004<br>(3/6 months)<br>May/Aug. - Oct. 2004<br>(3/6 months) | Jan. - Feb. 2005<br>(2 months)<br>Nov. - Dec. 2005<br>(2 months) |
| Chapter 6<br>Defect Prediction | PACS (pre-release)<br><br>PACS (post-release) | Nov. - Dec. 2005<br>(2 months)<br>Mar. - Apr. 2006<br>(2 months) | Jan. - Feb. 2006<br>(2 months)<br>May - Jun. 2006<br>(2 months) |
| Chapter 7<br>Series Mining | All 3 projects | Nov. - Dec. 2005<br>(2 months) | Jan. - Feb. 2006<br>(2 months) |
| Chapter 8<br>Process Events | All 3 projects | Sep. - Dec. 2005<br>(3 months) | Jan. - Feb. 2006<br>(2 months) |

Table 9.2: Time periods used in the experiments

To evaluate the improvements of evolvability through the application of refactoring we had to observe another 15 months of the development history.

In Chapter 5 we predicted locations that are prone to refactoring and are therefore likely to contain design deficiencies. The prediction models were built with the help of data mining techniques, which enable us to do project assessment with short time frames. Only three months of development time were required to predict refactorings in the following two months. An extension of the input period from three to six months did not help to get better results, but provided even less accurate perdition models.

Chapter 6 drew the attention to defects the first time in this thesis. As defects seem to be easier predictable than refactorings, we needed just two months of development time to predict defects of the following two months. The same time periods were used in Chapter 7, where we optimized the predictions with the help of value series of evolution metrics.

Finally, we reconstructed events of the software evolution process in Chapter 8 and used this data for defect prediction. As we focused only on the event types and did not use any other information, we had to extend the input period from two to three months. However, in general we are able to predict future activities in software project based on short time periods.

## 9.1 Roles in Software Projects

We have argued that several roles in software projects can benefit from this thesis in different ways

### 9.1.1 Project Manager

In their daily work project managers have to make manifold decisions and demand a sound basis to make the right decisions. This thesis can help them to estimate product quality on a high level where the number of expected defects can be predicted for the entire product. This can help to define appropriate dates for releases and other milestones in the course of the project.

Further the relationship between elements of evolution processes such as different event types and the resulting product attributes can be derived from the results of this thesis. This can help project managers to understand the necessity for certain activities such as refactoring and provides them material to convince others.

### 9.1.2 Software Engineer

Software engineers need a fine-grained picture of software evolution for their work. This thesis describes through multiple prediction models the complex relationship between detailed evolution aspects and resulting product attributes. For example Chapter 7 shows that teamwork and discipline are important to develop successful software products. Chapter 6 provides more details on the related evolution measures and their importance for evolvability.

Visual representations often help to establish a better understanding. Therefore, our EvoLens tool that is described in Chapter 4 enables the explorative navigation through software architecture and evolution to provide software engineers a clear picture of the system and the relationships of its elements.

### 9.1.3 TESTING ENGINEER

Testing specialists usually do not need such a deep understanding of the software internals as software engineers. On the other hand the high level view of the project manager is probably also not sufficient for the work of the testing specialist. For this role the thesis provides detailed prediction models on file basis to indicate locations with increased probability for defects. For these parts of the system more effort could be invested as the chance to discover defects is higher. This allows one a more efficient investment of people and time.

## 9.2 THREATS TO VALIDITY

In this section we present the threats to validity of our research. According to Gay and Airasian [47] the threats to validity are divided into a group of internal threats and a group of external ones (see also [117]).
The following list describes different types of internal threats to validity:

- **History.** Unexpected events occur between the pre- and posttest, affecting the dependent variable.
- **Maturation.** Changes occur in the participants, from growing older, wiser, more experienced, etc. during the study.
- **Testing Effects.** Taking a pretest alters the result of the posttest.
- **Instrumentation.** The measuring instrument is changed between pre- and posttesting, or a single measuring instrument is unreliable.
- **Statistical Regression.** Extremely high or extremely low scorers tend to regress to the mean on retesting.
- **Selection of Participants.** Participants in the experimental and control groups have different characteristics that affect the dependent variable differently.
- **Mortality.** Different participants drop out of the study in different numbers, altering the composition of the treatment groups.
- **Selection-Maturation Interaction.** The participants selected into treatment groups have different maturation rates. Selection interactions also occur with history and instrumentation.

The following list describes different types of external threats to validity:

- **Pretest-Treatment Interaction.** The pretest sensitizes participants to aspects of the treatment and thus influences posttest scores.
- **Selection-Treatment Interaction.** The nonrandom or volunteer selection of participants limits the generalizability of the study.
- **Multiple Treatment Interference.** When participants receive more than one treatment, the effect of prior treatment can affect or interact with later treatments, limiting generalizability.
- **Specificity of Variables.** Poorly operationalized variables make it difficult to identify the setting and procedures to which the variables can be generalized.
- **Treatment Diffusion.** Treatment groups communicate and adopt pieces of each other's treatment, altering the initial status of the treatments comparison.
- **Experimenter Effects.** Conscious or unconsious actions of the researcher affects participants' performance and responses.
- **Reactive Effects.** The fact of being in a study affects participants from their normal behavior. The Hawthorne and John Henry effects are reactive responses to being in a study.

### 9.2.1 Internal Threats to Validity

Our data rely strongly on automated processing. On one hand this ensures consistency, but on the other hand it is a source blurring effects. As such we missed defects that were not managed through the issue tracking system. We could try to handle this absence of information through heuristics on commit messages or other metrics, but we would still miss data such as the severity level of the defects. Furthermore, the mapping of issues to source code can only be done based on heuristics. Thus, we extracted issue numbers from commit messages to map the two information systems. To improve the situation we could try to map from bug reports to code changes based on commit dates and issue dates as described in [106]. In our field study this approach does not provide any valuable mappings, which we discovered on a random sample of 100 discovered matches.

We identify refactorings based on the commit messages of revisions entered by developers, when committing changes to files. To assess the quality of our identification technique, we tested our labeling of refactoring with randomly selected revisions. As described in Section 5.2 the number of false positives as well as the number of false negatives is very low.

Additionally, we do not distinguish between the type of refactorings such as class refactorings or method refactorings. Instead we only try to predict the number of future refactorings based on the past, independently from their nature. As a result, simple refactorings such as *rename* are counted as one of many refactorings, which gives them the same relevance as more complex refactorings

such as *extract super-class* or *introduce new parameter*. We found that we can predict refactoring for the Spring framework quite well, but could get even better results for ArgoUML. This is mainly due to their different project histories. ArgoUML is an older project and started in 1998, whereas the Spring framework followed later and started in 2003. Spring exhibits a dynamic evolution based on its young development history. For that, the results of ArgoUML are slightly better, but we still get predictions for the Spring framework with a precision between 0.53 and 0.89.

In large search spaces such as the set of possible features for our prediction approach it is not easy to find the global optimum. Genetic programming is one of the possible solutions, which has the drawback that it sometimes does not find the global optimum. But in many real world tasks genetic algorithms provide sufficient solutions. In such a manner the genetic algorithm provides good results on the project in our study, where high correlation and low error values could be achieved. Thus it seems suitable for the given task.

For our empirical study we selected software applications of different types such as graphical workstations, UML modeling tools, and application servers. Nevertheless, we still cannot claim absolute generalization of our approach on all different kinds of software systems. Therefore, we advise to evaluate the applicability of our approach on each specific software project. To provide some insight in the applicability to different domains we have selected different projects for our field study: commercial vs. open source and different domains such as health care, UML and application server. However, we cannot claim that these projects are representative for all different types of software projects. As a result the application of our approach to other software systems has to be re-evaluated on a per project basis.

### 9.2.2 External Threats to Validity

In general our approach relies on the consistency and quality of the underlying data. As a result, validity of our findings is related to the data of the versioning and issue tracking system. Versioning systems register single events such as commits of developers, which has certain effects on our analysis. It depends on the work habits of developers, if they commit small parts of their work or if they accumulate changes until a certain level. Thus, for example the measurement of large transactions vs. small ones could be a threat. However, we could show that an averaging effect supports statistical analysis [101] in general. Additionally, the data about work habits of people is on its own interesting information that we use for our quality prediction, where our balanced prediction models heavily rely on such features.

Especially, the reconstruction of evolution events (see Chapter 8) is dependent on the habits and the discipline of the developers. Developers feel the need to phrase the commit messages differently depending on whether they have the feeling that the fulfilled work had to be a quick hack or whether the piece of code was well-developed according to the given processes. When this information is reflected in the commit messages it even improves the reconstruction of event types.

Defect data has to be reliable to create plausible defect models. In our study the defect related information is extracted from issue tracking systems and mapped to files based on commit mes-

sages from the versioning systems. Thus, there can be certain inconsistencies so that we miss some defects. This approach is reliable enough to be suitable in this domain and it is commonly used by many researchers (e.g. [87], [103], [69]).

Further, we can only identify locations of defects and derive prediction models for components from this information. Bug fixes can take place at locations different to the source of defects. Similar approaches are used by other researchers [86, 92, 106]. Although the search for defects could be hindered through the fact that we predict defect corrections, but on the other hand we could provide insight into review efforts as defects fixing locations could be places of necessary code stabilizations.

The data points of our value series are computed as sums for each day. As a result, if a developer works through the night and commits some modifications before midnight and the remaining parts of modifications after midnight, we count the work on two days. Although this influences our value series, such information could still be valuable for defect prediction, because the working over night might have consequences on the level of concentration and the resulting software quality.

# Chapter 10

# Conclusions

In this thesis we presented the assessment of software projects from different perspectives taking especially evolution data into account. Evolutionary events are maintenance activities such as bug fixes, adding or enhancing a feature, refactorings, etc. Knowing them and evaluating their effect on the software project allow us assessing the impact on the project and the involved software modules. In project assessment we focus on the identification of elements of the system that are likely to be critical for the evolution. This outcome *enables guidance for project steering based on the extracted evolution characteristics*. To support the steering, we consider the evolutionary events based on a cost function, which describes the necessary rework that slows down the satisfaction of functional customer needs, as bug fixes use up a large amount of development effort.

As software evolves, it changes its size, complexity, and characteristics through modifications. The major costs do not arise because of software bugs, but because new and changing requirements lead to adaptations and enhancements [95]. As a result, it is important to keep software maintainable to ensure adequate responses to the users' needs. This thesis supports smooth and long lasting evolution of software through the presentation of assessment approaches for different aspects. In several studies we could show that evolution data is valuable for the anticipation of future trends.

## 10.1 Lessons Learned

Based on our experience with sPACE we can provide the following lessons learned:

- Many entangled aspects of software projects exist that contribute to the overall quality and should be subject of assessment methodologies. We defined a large number of over 60 evolution metrics for the assessment of software. These are related to different aspects such as size and complexity, team issues, process orientation, testing, etc. Our prediction models show that metrics of several topics should be included to build meaningful models.

- Metrics related to process orientation and team issues are important for defect prediction. They can outperform a simple prediction model where entities with low quality in the past will retain a high defect rate. To create better predictions we have to consider multiple aspects of software process in addition to pure technology-related issues such as size and complexity. The interconnection between people and their discipline are important for software quality.

- Although the exact data items that create optimal results in prediction models vary from project to project, we can still identify key aspect in development projects. When regarding prediction models based on series mining we can identify common aspects between projects such as the number of authors and commit messages.

- Prediction of different process events is possible. We can predicte bugs as well as maintenance activities such as refactorings to assess software projects from several perspectives. These event types can be predicted with a high accuracy and low error levels.

- Software events can be predicted on a short term basis. Two or three months of software evolution are sufficient to predict evolution events within the succeeding two months. We compared prediction models using information from either three or six months of development time to predict future events. The results of both time periods were similar in terms of accuracy. Thus, with the help of short time periods, project managers can better estimate the optimal time schedule based on the predictions.

- The focus on progress of evolution attributes over time helps to improve significantly prediction accuracy. Based on our primary approach to defect prediction we extended the idea by focusing on the course of time in software projects. Timing information has already been identified as an important aspect in defect prediction (e.g. [55, 103]). High correlation coefficients of more than 0.8 or even 0.9 can be reached with the help of series mining, where the temporal relationship of software events is taken into account.

## 10.2 Future Research

This thesis can be the foundation for several future extensions:

- *Software structure.* As we currently use evolution measures for quality estimations, we intend to enrich our models with information about software structures. Object-oriented inheritance hierarchies as well as data and control flow information provide much insight into software systems, which we will include in our quality considerations. The research work of Fluri et al. [36, 37] provide interesting starting-points for the incorporation of fine-grained information into predictions based on software evolution.

- *Understandability of prediction models.* We want to enrich our series mining approach to be able to analyze software projects in more detail and to get a better understanding of the

forces that influence software quality. To accomplish this goal we also look for improvements of series mining and the understandability of the resulting prediction models. For example, classification and regression trees provide the benefit that they provide a clear picture of the prediction model and the relationships of the used features. Kadous [64] presents interesting ideas in that direction.

- *Automation.* Our analysis relies on automated data processing such as information retrieval, mapping of defect and version information, and feature computation. The model creation relies on scripts using the Weka data mining tool [119] as well as the YALE machine learning environment [82]. Integrated tools providing different types of information such as the most important features can help different stakeholders. On one hand, developers can benefit from this information best, when it is available in the development environment. On the other hand, project managers need a lightweight tool separated from development environments to base their decisions on. We will focus on the implementation of appropriate tools for different scenarios.

- *Knowledge for event reconstruction.* Although the clustering algorithms seem to recover reasonable event types (see Chapter 8), we would like to apply more knowledge from the software engineering domain [3] to improve the identification of different software modification types.

- *Level of granularity.* In this thesis we focused on single files as items for defect prediction. In future we will investigate more fine-grained items such as methods and also the defect densities of more coarse-grained items such as modules or plug-ins.

- *Quality related factors.* As already indicated by our defect prediction models, refactoring is related to defect densities. Refactoring should be used to improve the design of existing code. As such it is not surprising that it is related with software quality assessment. However, currently we miss more studies regarding the relationship of refactoring and defects. We will focus in our future research activities on this topic.

# Bibliography


[1] Deepak Advani, Youssef Hassoun, and Steve Counsell. Refactoring trends across n versions of n java open source systems: an empirical study. Technical report, University of London, 2005.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[3] Sarabjot S. Anand, David A. Bell, and John G. Hughes. The role of domain knowledge in data mining. In *Proceedings of the CIKM*, pages 37–43. ACM Press, November 1995.

[4] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 31–40, Kyoto, Japan, 2004.

[5] Robert Arnold and Shawn Bohner. *Software Change Impact Analysis*. IEEE CS Press, 1996.

[6] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[7] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, September 1999.

[8] Jennifer Bevan, E. James Whitehead Jr., Sunghun Kim, and Michael W. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 177–186, Lisbon, Portugal, September 2005.

[9] J.M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the Symposium Software Reusability*, pages 259–262, April 1995.

[10] Christian Bird, Alex Gourley, and Prem Devanbu. Detecting patch submission and acceptance in oss projects. In *Proceedings of the International Workshop on Mining Software Repositories*, page 26, Minneapolis, USA, May 2007.

[11] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 137–143, Shanghai, China, May 2006.

[12] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merritt. Characteristics of software quality. 1978.

[13] Lionel C. Briand, Victor R. Basili, and William M. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, 18(11):931–942, November 1992.

[14] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *Proceedings of the International Conference on Software Engineering*, 1999.

[15] Elizabeth Burd and Malcolm Munro. An initial approach towards measuring and characterizing software evolution. In *Proceedings of the Working Conference on Reverse Engineering*, pages 168–174, Atlanta, Georgia USA, October 1999.

[16] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. Evolution of understandability in oss projects. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 58–66, Tampere, Finland, March 2004.

[17] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 197–211, Phoenix, Arizona, USA, 1991.

[18] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[19] M. O. Cinnéide. Automated refactoring to introduce design patterns. In *Proceedings of the International Conference on Software Engineering*, 2000.

[20] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[21] Stephen Cook, He Ji, and Rachel Harrison. Software evolution and software evolvability. Technical report, University of Reading, 2000.

[22] Michael A. Cusumano. *Japan's Software Factories*. Oxford University Press, 1991.

[23] Marco D'Ambros, Michele Lanza, and Harald C. Gall. Fractal figures: Visualizing development effort for cvs entities. In *Proceedings of the International Workshop on Visualizing Software For Understanding and Analysis*, pages 46–51. IEEE CS Press, 2007.

[24] M. K. Daskalantonakis. A practical view of software measurement and implementation experiences within motorola. *IEEE Transactions on Software Engineering*, 18(11), November 1992.

[25] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, 2000.

[26] Serge Demeyer and Stèphane Ducasse. Metrics: Do they really help? In *Proceedings Languages et Modèles à Objets*, pages 69–82. Hermes Science Publications, 1999.

[27] Serge Demeyer, Stèphane Ducasse, and Michele Lanza. A hybrid reverse engineering approach combining metrics and program visualisatio. In *Proceedings of the Working Conference on Reverse Engineering*, pages 69–82. IEEE Computer Society Press, October 1999.

[28] Serge Demeyer, Stèphane Ducasse, and Oscar Nierstrasz. *Object-oriented Reengineering Patterns*. Morgan Kaufmann Publishers, An Imprint of Elsevier Scienence: San Francisco CA, USA, July 2002.

[29] Serge Demeyer, Filip Van Rysselberghe, Tudor Girba, Jacek Ratzinger, Radu Marinescu, Tom Mens, Bart Du Bois, Dirk Janssens, Stèphane Ducasse, Michele Lanza, Matthias Rieger, Harald Gall, and Mohammad El-Ramly. The lan-simulation: A research and teaching example for refactoring. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 123–131, September 2005.

[30] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the International Conference on Software Engineering*, pages 241–251. ACM Press, May 2002.

[31] John Dill, Lyn Bartram, Albert Ho, and Frank Henigman. A continuously variable zoom for navigating large hierarchical networks. In *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, pages 386–390, October 1994.

[32] R. G. Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[33] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurth, 1994.

[34] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, September 1999.

[35] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press.

[36] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the International Conference on Program Comprehension*, page to appear, Athen, Greece, June 2006. IEEE Computer Society Press.

[37] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):to appear, November 2007.

[38] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, June 1999.

[39] J. H. Friedman. Another approach to polychotomous classification. Technical report, Department of Statistics, Stanford University, 1996.

[40] George W. Furnas. Generalized fisheye views. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 16–23, September 2003.

[41] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 190–198, Bethesda MD, USA, November 1998. IEEE Computer Society Press.

[42] Harald Gall, Mehdi Jazayeri, Rene Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 160–166, Bari, Italy, October 1997.

[43] Harald Gall, Mehdi Jazayeri, and Jacek Ratzinger (former Krajewski). CVS release history data for detecting logical couplings. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 13–23, Lisbon, Portugal, September 2003. IEEE Computer Society Press.

[44] Harald Gall, Mehdi Jazayeri, and Claudio Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the International Conference on Software Maintenance*, pages 99–108, Oxford, England, August 1999. IEEE Computer Society Press.

[45] Keith Brian Gallagher and James Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[46] Pankaj K. Garg, Mehdi Jazayeri, and Michael L. Creech. A meta-process for software reuse, process discovery, and evolution. In *Proceedings of the International Workshop on Software Reuse*, Oswego, USA, November 1993.

[47] L. R. Gay and Peter Airasian. *Educational Research: Competencies for Analysis and Application*. Merrill/Prentice Hall, seventh edition, 2003.

[48] David Gefen and Scott L. Schneberger. The non-homogeneous maintenance periods: A case study of software modificaitons. In *Proceedings of the International Conference on Software Maintenance*, pages 134–141, Monterey, USA, 1996.

[49] Daniel M. German, Abram Hindle, and Norman Jordan. Visualizing the evolution of software using softchange. In *Proceedings of the Internation Conference on Software Engineering and Knowledge Engineering*, pages 336–341, June 2004.

[50] Pierre Geurts. Pattern extraction for time series classification. In *Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery*, pages 115–127, 2001.

[51] J. Myers Glenford, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 1979.

[52] Michael Godfrey, Xinyi Dong, Cory Kapser, and Lijie Zou. Four interesting ways in which history can teach us about software. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[53] Nicolas Gold and Andrew Mohan. A framework for understanding conceptual changes in evolving source code. In *Proceedings of International Conference on Software Maintenance*, pages 432–439, Amsterdam, The Netherlands, September 2003.

[54] Julio Gonnet. Data mining within eclipse. Master's thesis, University of Zurich, Zurich, Switzerland, January 2007.

[55] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[56] Gianluigi Greco, Antonella Guzzo, and Domenico Saccà Luigi Pontieri. Discovering expressive process models by clustering log traces. In *Proceedings of Pacific-Asia Conference Advances in Knowledge Discovery and Data Mining*, pages 52–62, Sydney, Australia, May 2004. LNCS.

[57] Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, USA, 1977.

[58] Ahmed E. Hassan and Richard C. Holt. The chaos of software development. In *Proceedings of the International Workshop on Principles of Software Evolution*, page 84, Helsinki, Finland, September 2003.

[59] L. Hatton. Re-examining the fault density-component size connection. *IEEE Software*, 14(2):89–98, March/April 1997.

[60] M. Hitz and B. Montazeri. Measuring product attributes of object-oriented systems. In *Proceedings of the European Software Engineering Conference*, Barcelona, Spain, 1995.

[61] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, February 1999.

[62] Andreas Jetter. Assessing software quality attributes. Master's thesis, University of Zurich, Zurich, Switzerland, January 2007.

[63] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. In *Proceedings of the European Conference Software Maintenance and Reengineering*, pages 39–46. IEEE Computer Society Press, 2001.

[64] Mohammed Waleed Kadous. Learning comprehensible descriptions of multivariate time series. In *Proceedings of the International Conference on Machine Learning*, pages 454–463, San Francisco, USA, June 1999.

[65] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance*, pages 576–585, October 2002.

[66] T. Keller. Measurement's role in providing error-free onboard shuttle software. In *Proceedings of the International Applications of Software Metrics Conference*, pages 2154–2166, La Jolla, USA, 1992.

[67] Chris F. Kemerer and Sandra A. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July-August 1999.

[68] Taghi M. Khoshgoftaar, Xiaojing Yuan, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, 7(4):297–318, December 2002.

[69] Sunghun Kim, Thomas Zimmermann, Jr. E. James Whitehead, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the International Conference on Software Engineering*, pages 20–26, Minneapolis, USA, May 2007.

[70] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 119–125, Shanghai, China, May 2006. ACM Press.

[71] K. Koga. Software reliability design method in hitachi. In *Proceedings of the European Conference on Software Quality*, Madrid, Spain, 1992.

[72] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[73] Manny Meir Lehman and Laszlo A. Belady. *Program Evolution - Process of Software Change*. Academic Press, London and New York, 1985.

[74] Manny Meir Lehman, Dewayne E. Perry, and Juan Fernandez Ramil. Implications of evolution metrics on software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 208–217, Bethesda, Maryland, USA, November 1998.

[75] Manny Meir Lehman, Dewayne E. Perry, Juan Fernandez Ramil, Wladyslaw M. Turski, and Paul D. Wernick. Metrics and laws of software evolution - the nineties view. In *Proceedings of the International Software Metrics Symposium*, pages 20–32, Albuquerque, NM, USA, November 1997.

[76] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23:111–122, 1993.

[77] Chung-Horng Lung. Software architecture recovery and restructuring through clustering techniques. In *Proceedings of the International Workshop on Software Architecture*, pages 101–104, Orlando, Florida, USA, November 1998.

[78] Stefanos Manganaris. *Supervised Classification with Temporal Data*. PhD thesis, Computer Science Department, School of Engineering, Vanderbilt University, December 1997.

[79] Thomas J. McCabe. Object-oriented software construction. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[80] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. *Natl Tech.Information Service*, 1, 2, and 3, 1977.

[81] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[82] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. YALE: Rapid prototyping for complex data mining tasks. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 935–940, Philadelphia, USA, 2006.

[83] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, pages 120–130. IEEE Computer Society, 2000.

[84] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April-June 2000.

[85] K.H. Moeller and D. Paulish. An empirical investigation of software fault distribution. In *Proceedings of the International Software Metrics Symposium*, pages 82–90, 1993.

[86] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the International Conference on Software Engineering*, pages 284–292, St. Louis, MO, USA, May 2005. ACM Press.

[87] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering*, pages 452–461, Shanghai, China, May 2006. ACM Press.

[88] R. Najjar, S. Counsell, G. Loizou, and K. Mannock. The role of constructors in the context of refactoring object-oriented systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 111–120, March 2003.

[89] Allen P. Nikora and John C. Munson. Developing fault predictors for evolving software systems. In *Proceedings of the Software Metrics Symposium*, pages 338–350, Sydney, Australia, September 2003.

[90] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[91] International Standardization Organisation. *ISO/IEC 9126 — Software engineering - Product quality*. International Organization for Standardization, Geneva, Switzerland, 1 edition, 1991.

[92] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 55–64, Rome, Italy, July 2002.

[93] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proceedings on the International Symposium on Software Testing and Analysis*, pages 86–96, Boston, Massachusetts, USA, July 2004.

[94] Dewayne E. Perry. Dimensions of software evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 296–303, Victoria BC, USA, September 1994.

[95] Thomas M. Pigoski. *Practical Software Maintenance*. John Wiley and Sons, New York, 1997.

[96] Adam A. Porter and Richard W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, March 1990.

[97] J. Ross Quinlan. Learning with continuous classes. In *Proceedings of the Australian Joint Conference on Artificial Intelligence*, pages 343–348, Singapore, China, 1992.

[98] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[99] Vàclav T. Rajlich and Keith H. Bennet. A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71, July 2000.

[100] Vclav T. Rajlich and Keith H. Bennet. A staged model for the software life cycle. *IEEE Computer*, 33(7):66–71, July 2000.

[101] Jacek Ratzinger, Michael Fischer, and Harald Gall. Evolens: Lens-view visualizations of evolution data. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal, September 2005.

[102] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 69–73, St. Louis, USA, May 2005.

[103] Jacek Ratzinger, Martin Pinzger, and Harald Gall. EQ-Mine: Predicting short-term defects for software evolution. In *Proceedings of the Fundamental Approaches to Software Engineering at the European Joint Conferences on Theory And Practice of Software*, pages 12–26, Braga, Portugal, March 2007.

[104] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining software evolution to predict refactoring. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, page to appear, Madrid, Spain, September 2007.

[105] Walt Scacchi. *Process Models in Software Engineering*, pages 993–1005. John Wiley and Sons, Inc, New York, 2nd edition, December 2001.

[106] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 18–27, Rio de Janeiro, Brazil, September 2006. ACM Press.

[107] Jelber Sayyad Shirabad, Timothy C. Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *Proceedings of the International Conference on Software Maintenance*, pages 95–104, Amsterdam, The Netherlands, September 2003.

[108] Frank Simon, Frank Steinbrückner, and Claus Lewernetz. Metrics based refactoring. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 30–38, March 2001.

[109] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, May 1974.

[110] Ryouei Takahashi, Yoichi Muraoka, and Yukihiro Nakamura. Building software quality classification trees: Approach, experimentation, evaluation. In *Proceedings of the International Symposiumi on Software Reliability Engineering*, page 222, Albuquerque, USA, 1997.

[111] Jeff Tian. Integrating time domain and input domain analyses of software reliability using tree-based models. *IEEE Transactions on Software Engineering*, 21(12):945–958, December 1995.

[112] Nikolaos Tsantalis, Alexander Chatzigeorgiou, and George Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, 31(7):601–614, July 2005.

[113] W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE TKDE*, 19(9):1128–1142, 2004.

[114] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Working Conference on Reverse Engineering*, pages 97–108, Richmond, Virginia, USA, October 2002. IEEE Computer Society Press.

[115] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing bug finding tools with reviews and tests. In *Proceedings of the International Conference on Testing of Communicating Systems*, pages 40–55, Montreal, Canada, May 2005.

[116] Stefan Wagner and Tilman Seifert. Software quality economics for defectdetection techniques using failure prediction. In *Proceedings of the International Conference on Software Engineering*, pages 1–6, St. Louis, Missouri, May 2005.

[117] John Wasson. Methods of educational research. An Internet Based Course.

[118] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the International Workshop on Mining Software Repositories*, page 1, Minneapolis, USA, May 2007.

[119] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, USA, 2 edition, 2005.

[120] Alexander L. Wolf and David S. Rosenblum. A study in software process data capture and analysis. In *Proceedings of the International Conference on Software Process*, pages 115–124, Los Alamitos, USA, 1993. IEEE Computer Society.

[121] David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.

[122] Annie T. T. Ying, Gail C. Murphy, Raymong Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.

[123] Ping Yu, Tarja Systa, and Hausi Müller. Predicting fault-proneness using oo metrics: an industrial case study. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 99–107, Budapest, Hungary, March 2002.

[124] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering*, volume 00, pages 563–572, Edinburgh, Scotland, UK, May 2004.

# Curriculum Vitae

Jacek Ratzinger

**Personal Data**

| | |
|---|---|
| Place of birth: | Gdansk, Poland |
| Date of birth: | 5th of January 1979 |
| Citizenship: | Austria |
| Family status: | Married |
| Graduation: | Dipl.-Ing. (MSc.) at Vienna University of Technology |
| Address: | Birneckergasse 48, A-1210 Vienna, Austria |
| E-Mail: | ratzinger@infosys.tuwien.ac.at |

**Education**

| | |
|---|---|
| 2004 - 2007 | PhD study in Informatics<br>Vienna University of Technology, Austria |
| 06/2003 | MSc. Graduation in Information Science (1st class honours)<br>Vienna University of Technology, Austria |
| 09/2002 - 02/2003 | Study leave for 6 months<br>University of Gent, Belgium |
| 1997 - 2003 | Study of Informatics<br>Vienna University of Technology, Austria |
| 1989 - 1997 | Secondary school (AHS)<br>BG/BRG 21, Franklinstraße 21, Austria |
| 1985 - 1989 | Primary school<br>GTVS 12, Am Schöpfwerk 27, Austria |

**Work Experience**

| | |
|---|---|
| 07/2003 - | Quality Management of Radiological Department<br>Agfa Healthcare, Austria |
| 04/2002 - 06/2003 | Quality Assurance for Software Reliability<br>Tiani Medgraph AG, Austria |
| 08/2000 | Project co-worker<br>Department of Program Development for Mobile Phones<br>Siemens Austria |
| 02/2000 - 07/2000 | Software Development for Radiological Systems<br>Tiani Medgraph AG |