Automated Support for Process Assessment in Test-Driven Development

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Christian Wege
aus Porz am Rhein

Tübingen
2004

Tag der mündlichen Prüfung:	21. Juli 2004
Dekan	Prof. Dr. Ulrich Güntzer
1. Berichterstatter:	Prof. Dr. Herbert Klaeren
2. Berichterstatter:	Prof. Dr. Wolfgang Küchlin

Abstract

Test-Driven Development (TDD) is a style of agile software development that has received much attention recently in the software development community.

Agile software development methods stress the importance of software as the most significant *output* of a development team, leading to a continuous flow of source code changes. The view on past source code changes as *input* for a better understanding of how a team has produced the software is a topic that deserves much more attention than it has received thus far.

In this dissertation, I claim that an analysis of past software changes can indicate TDD process violations. I propose a tool to prepare and analyze software changes from a source code repository. I propose process compliance indices (PCIs) to interpret the analysis results in order to focus a manual process assessment effort.

This dissertation facilitates a better understanding of how TDD developers change software, where they are lazy in following the process discipline, and to help them improve their development practices.

Zusammenfassung

Testgetriebene Entwicklung (engl. Abk. TDD) ist ein Stil agiler Software-Entwicklung, der in letzter Zeit viel Beachtung erfahren hat.

Agile Software-Entwicklungsmethoden betonen die Bedeutung von Software als dem wichtigsten *Produkt* eines Entwicklungs-Teams, was zu einer kontinuierlichen Abfolge von Quelltext-Änderungen führt. Die Sicht auf vergangene Quelltext-Änderungen als *Quelle* für ein besseres Verstehen wie ein Team die Software erstellt hat, verdient viel mehr Beachtung als sie bislang erfahren hat.

In dieser Dissertation stelle ich die These auf, dass die Analyse vergangener Software-Änderungen auf TDD-Prozessverletzungen hinweisen kann. Ich schlage ein Werkzeug vor, das Software-Änderungen aus einem Quelltext-Versionsspeicher geeignet aufbereitet um sie anschließend zu analysieren. Ferner schlage ich Prozessbefolgungs-Indices (engl. Abk. PCI) vor, um die Analyse-Resultate zu interpretieren und die manuelle Prozess-Bewertung zu fokussieren.

Diese Dissertation ermöglicht ein besseres Verstehen, wie TDD-Entwickler Software ändern, wo es ihnen an Prozess-Disziplin mangelt und hilft, deren Entwicklungs-Praktiken zu verbessern.

Acknowledgements

I thank my supervisor, Prof. Dr. Herbert Klaeren, for his support, guidance and patience during my studies at the University of Tübingen. I learned much from his feedback in our bi-weekly meetings. I was amazed by both his insights and his stamina. He always asked the right questions at the right time. And he invested his most valuable resource on my behalf: his time.

I also thank my examiner, Prof. Dr. Wolfgang Küchlin. He inspired me to see the value of my ideas for education and gave valuable feedback.

Being a TDD process mentor, Jens Uwe Pipka is one of the persons TddMentor aims to support. I owe him thanks for his comments and valuable feedback on the dissertation draft.

Frank Gerhardt was my thought catalyst on many of the ideas that came to my mind. He helped me filter out the bad ones and keep the good ones. I also owe him thanks for extensive feedback on my dissertation draft.

I thank Wilfried Reimann from DaimlerChrysler who enabled my sabbatical so that I could concentrate on my research and writing. Without taking time off, this project would not have been possible.

I also owe thanks to the participants of the doctoral symposium at OOPSLA. Doug Lea, as the symposium chair, and his co-mentors, namely Brent Hailpern, James Noble, Mary Beth Rosson, and Ron Goldman shared their tremendous experience. Especially Richard Gabriel who gave “sparkling” motivation to follow my research direction.

Wim De Pauw opened the doors of the IBM Watson Research Center, where I presented my early research ideas and results. John Vlissides encouraged me to put the rigor into my results that they now have.

I thank the participants of the “Tools I wished I had” open space during the “Dezember” meeting in Sonnenhausen — among others, Peter Roßbach and Bastiaan Harmsen. Especially Tammo Freese who offered valuable input about refactorings and source repositories.

Thanks to Erich Gamma, who allowed me to present my ideas and results to a broader public at EclipseCon. Martin Aeschlimann, Keller Keller, and Michael Valenta from the Eclipse development team who helped me to straighten out some issues in my Eclipse plug-in.

Carsten Schulz-Key offered continuous encouragement. Being a doctoral candidate as well, he provided much critical insight from a peer point of view.

Last but not least, I thank Susanne Kilian for sharing her insight into experimental design in biology and being an invaluable source of inspiration.

To my brother.

Contents

Chapter 1

Introduction

Test-Driven Development (TDD) is a style of agile software development that has received much attention recently in the software development community. A growing number of development teams want to apply TDD in order to benefit from its advantages.

1.1 Problem

TDD requires the execution of given practices with high discipline. An agile software development team, in general, presupposes continuous feedback on many levels. Organizational tools, such as self-organizing teams and continuous delivery, are feedback mechanisms that support a high process discipline. Some software tools (e.g. test coverage monitors) serve the same purpose. Those software tools work in-process and are typically applied by advanced TDD teams. They help a team to improve its TDD practices alone or with external process mentoring.

An agile team also reflects about its development process at regular intervals in order to tune and adjust its behaviour accordingly. Project retrospectives help a team reflect over what has happened and how the work was done. TDD teams aim at improving their TDD development practices.

The degree of discipline with which developers perform TDD practices impacts on how the software (i.e. test and production source code in a source code repository) is changed. This change history of a whole application contains a wealth of information about what has happened in the past of a TDD project.

The problem is, that this information in past software changes is not leveraged for process assessment. The picture of what has happened in the past of a development project stays incomplete. To my knowledge, no prior work has tried to fill this gap.

It is the goal of this dissertation to show that a static analysis of these source code changes can support TDD process assessment. I explain how to perform such an analy-

sis and provide tool support for this purpose.

1.2 Proposed Solution

In this dissertation I claim that an analysis of past software changes can indicate TDD process violations. I propose (and have implemented) a tool to access such software changes in a source code repository and have analyzed the data. This tool is called TddMentor. I propose several process compliance indices (PCI) to interpret the analysis results in order to focus a manual process assessment effort. Figure 1.1 shows the basic activities and information flows.

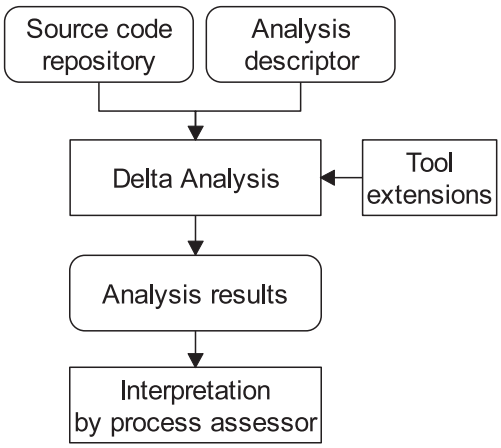

Figure 1.1: *Basic activities and information flows.* The source code repository is the primary information source. The analysis descriptor scopes the information for delta analysis. The analysis results help focus a manual assessment effort.

The source repository contains the required source code revisions and other version information (such as commit comments and time stamps). TddMentor recreates the individual integration versions. An integration version is the result of a basic TDD cycle. The difference between two consecutive integration versions is called an integration delta. It contains the software changes that were applied during a basic development cycle.

The analysis descriptor declares the scope of integration deltas to be analyzed and some other information required for an automated analysis. It must be provided by the user of TddMentor.

The delta analysis calculates and reports a number of PCIs that indicate TDD process violations. It performs static source code analyses of the software changes in test and production code.

The results are provided as time series diagrams for every integration delta within the analysis scope. For the prototypical implementation of TddMentor, some tool extensions might be necessary to achieve better results. For example, at the time of writing, the number of detectable refactoring types was biased towards the documented case studies.

A TDD process assessor is an individual who performs a TDD process assessment. This individual interprets the analysis results to focus the manual analysis effort of individual integration deltas. The analysis results can indicate process violations. However, although they can help to reduce the manual assessment effort dramatically, these indications are of a heuristic nature and still require human expert judgement. When in doubt, the process assessor discusses results with the developers.

A major advantage of this approach is that it can help development teams that did not start with TDD process assessment in mind. It does not require any in-process support except for the fact that the past software changes must be available and must conform to some conventions that are described later. To some degree, this proposed approach can emulate in-process support in retrospect.

Another advantage is that an analysis of past software changes deals with what really happened. It is less dependent on what the developers think has happened in the past. The developer's remembrance could have been biased towards a misunderstanding of the TDD development practices.

For TDD developers, the source code is the most important process output. It is also the most important document for analysing the software design. This dissertation opens up the source code history for reasoning with respect to the TDD process compliance.

TddMentor is a prototypical implementation of the approach proposed by this dissertation. Several case studies show the validity of the research hypothesis.

1.3 Document Structure

This chapter sets the stage for what follows. It describes the problem and the proposed solution of this dissertation.

Chapters 2 and 3 review the literature concerning TDD, agile software changes and software assessment to motivate this dissertation's approach. The goal of this dissertation is to support the assessment of TDD. The static analysis of past agile software changes is the medium chosen to reach that goal. Software process assessments are diagnostic studies concerning the state of a development process. Software measurements of agile software changes feed the software process assessment. These software measurements are performed on change data from software repositories.

The research hypothesis is detailed in Chapter 4.

In Chapters 5 through 7, I propose detection techniques, process compliance indices (PCIs), and TddMentor — a tool that implements the proposed approach. Integration

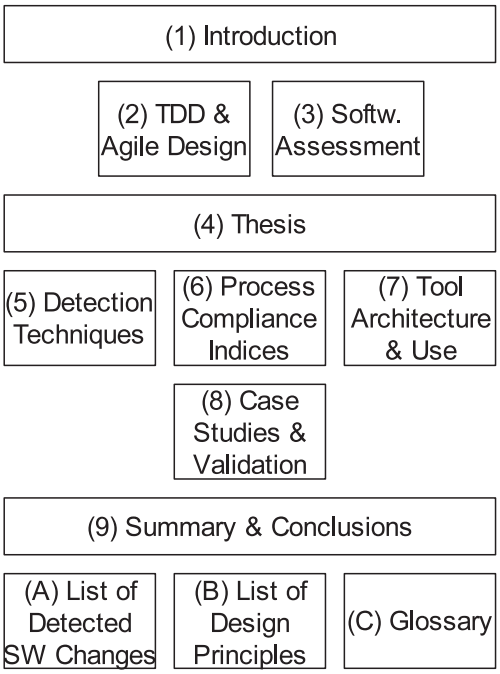

Figure 1.2: *Dissertation structure.* Chapters 1, 4, and 9 are the "fast track" through this dissertation. Chapter 8 contains some case studies that demonstrate how to use the proposed tool. Chapters depicted in parallel can be read in any order.

deltas and safe software changes are reengineered from existing source code repositories. PCIs are software measurements over past integration deltas. They illustrate a project's process compliance over time. TddMentor is a plug-in for the tool integration platform Eclipse.

The case studies in Chapter 8 validate the research hypothesis. Two case studies had applied a TDD process. One case study serves as a counter-example.

Chapter 9 summarizes the contributions, gives a perspective on future work, and concludes this dissertation.

Software changes which TddMentor can detect at the time of writing, are listed in Appendix A. Appendix B lists agile design principles used in the text. A glossary of technical terms which have not been defined elsewhere is given in Appendix C.

1.3.1 Conventions

Source code examples in this dissertation are written in Java. Most of the examples are taken from the documented case studies. Source code, code snippets, and XML files are rendered like `toString()`.

The names of the refactorings are taken from literature where possible. If a refactoring is not listed somewhere, an intention-revealing name was chosen. All refactorings, refactoring participants, and change participants are explained in Appendix A. They are rendered like *Extract Method*.

The safe software changes are calculated from method deltas. Method deltas are rendered like *Rename Parameter*. All detected method deltas are listed in Appendix A.

TddMentor program component and module names are rendered like AST Matcher.

The names of the Java abstract syntax tree nodes correspond to the names inside the Eclipse Java model. They are rendered in the same way as source code (e.g. `MethodDeclaration`).

In all UML diagrams that illustrate a source code change the following conventions apply. The same components in different versions are distinguished via a quote character (') which does not necessarily imply any change. Components that change are coloured in light-grey, while new or removed components are coloured in dark-grey.

References to online resources or web sites are collected at the end of this document. Throughout the text they are rendered like [URL:JUnit].

Chapter 2

Test-Driven Development and Agile Software Changes

This chapter introduces agile software development and Extreme Programming as its most prominent method (see Section 2.1). TDD (see Section 2.2) is a style of agile software development that is an integral part of Extreme Programming but can also be used in other development methods. Source code in TDD evolves by the continuous application of agile software changes (see Section 2.3).

2.1 Agile Software Development

Agile software development has recently become popular.[1] This agile methodology is best exemplified by the Agile Manifesto [All01]:

> We have come to value:
> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
> That is, while there is value in the items on the right, we value the items on the left more.

Boehm *et al.* [BT03] contrast agile approaches to traditional plan-driven approaches such as SW-CMM (see Section 3.2.1), SPICE, and Cleanroom [Cox90]. Agile methodologies have not yet found their way into the so-called *Software Engineering Body of Knowledge* [AM03], that is under development by the IEEE. The agile movement advocates more light-weight processes and encourages programmers to embrace change.

[1] See special issue of IEEE Computer, June 2003, Vol. 36, No. 6.

Instead of executing a constant plan, agile teams constantly plan their success in short cycle times and keep an adaptive rather than a predictive mind-set.

Agile development methods have created a wealth of strategies to respond to change. Some of the principles which help in this endeavour are "The best architectures, requirements, and designs emerge from self-organizing teams" and "At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly" [All01]. Agile methods also address evolutionary delivery as one core practice; an issue typically neglected by plan-driven approaches [Gil96].

The agile movement stresses the importance of the human side of software development. While more traditional methods "try to remove the human factors of intuition, emotion, creative struggle, and conflict from the software process" [CHK+97], agile proponents promote people as non-linear, first-order components in software development [Coc00]. Constant training is as important as regular reflections about their own work (see Section 3.4.1).

The agile manifesto distinguishes between values (as cited above) and principles that are based on this value system. The agile software community has explored techniques to put those principles into practice.

2.1.1 Extreme Programming

Extreme Programming (XP) is the most prominent agile software development method.[2]

XP proponents see four variables to control a software development project: resources, time, quality, and scope [Bec00]. An XP team tries to control resources, time, and quality. The fourth variable depends on the other variables and cannot be controlled directly. In practice, this means that an XP team does not have a fixed feature set at the beginning of a project. New features are added incrementally and the customer regularly reviews the application to refine the next incremental feature set.

XP teams share the values communication, simplicity, courage, and feedback. XP developers communicate with the customers in the planning game; using index cards to plan requirements as so called "user stories". They communicate very much face-to-face with each other in a co-located environment and through source code (in the form of test and production code). Especially the application design but also all other artifacts have to be as simple as possible. An XP project allows the team members to courageously work on the application design to make it better and keep it simple. XP developers seek constant feedback: from the automated tests, from the customer, and from each other.

An XP team applies twelve practices that interact strongly with each other, e.g. pair programming, continuous delivery, coding standards, sustainable pace, and Test-Driven

[2] See the special issue of IEEE Software, May/June 2003, Vol. 20, No. 3 or the IEEE Dynabook about Extreme Programming [URL:Dynabook].

Development. While TDD is part of this development method, it can also be used in other software development methods. It is, however, not considered to be a method in its own right. Beck [Bec03, p. ix] categorizes TDD as a “style of development”. In the context of XP, TDD plays an important role:

- The tests are executable specifications of the user requirements,
- they give regular feedback to the programmers,
- they communicate the intent and use of application features, and
- allow courageous refactoring of the code base.
- Each integration delta implements a new feature and
- adds to the running application as part of each integration version.

This dissertation concentrates on TDD, because the way it is performed by a team is directly visible in the code base. Practices such as continuous delivery or pair programming cannot be traced easily in the technical artifacts of a project. Also, TDD is applied in many other agile software development methods, allowing this research to contribute to a broader scope of projects.

2.2 Test-Driven Development

The basic development cycle of TDD (see Section 2.2.1) demands the creation of automated tests to drive the implementation of new features (see Section 2.2.2). Continuous feedback helps the developers to hold up the process discipline (see Section 2.2.3).

2.2.1 Basic Development Cycle

Like other agile software development methods, TDD values “working software as a primary measure of progress” [All01]. TDD also demands the creation of automated tests as the primary driver for this working software [Bec03].

The automated tests work on the level of programming units such as methods, classes and clusters of classes and are therefore called *unit tests*. The sum of the tests forms a test harness. A *test harness* “is a system that supports effective and repeatable automated testing” [Bin99, p. 957]. Creators of a test harness typically choose a testing framework for the efficient creation and execution of the tests. For the Java programming language, the JUnit [URL:JUnit] framework has become the de facto standard. All case studies of this dissertation use JUnit (see Chapter 8).

In his comprehensive book about testing object-oriented systems, Binder [Bin99, p. 41] states the classical definition of *software testing* as "the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs."

TDD extends the scope of software testing beyond system verification, as described by Martin [Mar03, p. 23]: "The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than of verification". The evolution of the name of the approach also reflects the extended scope of the unit tests. TDD was formerly called "test-first programming". That name reflected the order in which test and production code is written in TDD. However, it neglected the most important role of the tests to actually *drive* new functionality into the application under development — hence the new name.

The importance of testing for application design has also been observed by others. Eickelmann *et al.* [ER96] demand that "testing be initiated prior to development." Beizer [Bei93] observed that "the act of designing tests is one of the most effective error prevention mechanisms known[...]. The thought process that must take place to create useful tests can discover and eliminate problems at every stage of development."

This driving of new functionality into the application goes along with its design evolution. At any point in time the application design is only prepared for the features that it already implements. It is not prepared for features that will be added in the future. Moreover, TDD sets the goal to "always have the simplest design that runs the current test suite" [Bec00]. This might require the evolution of the application design at the end of every cycle. Martin [Mar03] lists a number of principles and patterns that are regarded as good design (see Section 2.3.2). Refactorings can help to actually perform the necessary design changes (see Section 2.3.4).

All the design changes are reflected directly in the source code. In the mind-set of TDD, the "source code is the design" [Mar03, p. 87]. Design models such as UML diagrams might help the programmers when they are designing a new feature into the application. However, the only really accurate record of the design changes is found in the source code.

This quest for simple design also implies that the developers "[D]on't build for tomorrow" [JAH00, p. 129]. The application only has to satisfy the test harness and the design only has to be the simplest design for the current implementation and test harness. This concept is known as "You Aren't Going to Need It" [URL:YAGNI] or visible in the desire to avoid the so-called design smell of "Needless Complexity".[3]

The basic development cycle in TDD consists of the following steps [Bec03]:

[3] A design smells of "Needless Complexity" if "the design contains infrastructure that contains no direct benefit" [Mar03]. If a developer anticipates future changes to the software, he/she is tempted to introduce abstractions into the application which are meant to be used later. This adds complexity to the application which is not needed at that time. The application is therefore harder to understand and more difficult to change.

1. Red — Every development cycle starts with a new test. The objective of the test is to drive the implementation of a new feature. After the implementation of a new test, the developer executes the test harness to see if the new test fails (i.e. receive a red light).

2. Green — Make tests succeed with the simplest and possibly dirty implementation. Maybe duplicate code and violate existing abstractions. The goal of this step is to make the test harness succeed as quickly as possible (i.e. receive a green light). Perfective design changes are deferred to the next step.

3. Refactor — Remove duplication and clean the code. Create new abstractions required by the new test-driven feature. The goal is to evolve the application design into a good state for the features that have accumulated in the application. An important observation is that the tests help to achieve a good design. In order for the units under test to be testable individually, they must be decoupled from the rest of the application. This is regarded as good design.

In summary, the important observation is the pivotal role of the tests. They trigger the implementation of new functionality. Also, the application design may only evolve as part of a basic development cycle because it needs to be driven by the tests.

2.2.2 Interdependence of Test and Production Code

The test and production code is highly interdependent. On the one hand, the tests drive new implementations of the production code. On the other hand, new tests need to build their fixtures by using the API of the production code. Also, refactorings of the application design might also impact the test harness (see Section 2.3.4).

The relationship of automated tests and static analysis of program evolution has become the subject of active research. The Echelon system can analyse the differences between two versions in program binaries to help prioritize the automated tests [ST02]. Echelon works on the granularity of building block in binary files. It is part of the efforts of the Microsoft Programmer Productivity Research Center (PPRC). Their mission is to increase the programmer test effectiveness. For this, they collect information concerning development processes and product cycles [Sta03].

A similar approach uses the program dependence graph to find differences between two versions of a program and to select an adequate subset of the test suite [RH94]. To compute this subset, it needs an execution trace of previous test runs. This approach, however, assumes that the modified test suite T' is a superset of the original test suite T. In TDD, test code evolves in the same manner as production code, and this can lead to structural changes in the tests via refactoring.

Common to those approaches is the goal to increase the test effectiveness. However, the concern of this dissertation is to support the understanding of the development process by leveraging the interdependence of test and production code.

2.2.3 Discipline and Feedback

The dictionary definition of discipline includes both "common compliance with established processes" (external discipline) and "self-control" (internal discipline). Boehm and Turner [BT03, p. 5] identified this multiple definition as a source of perplexity, when comparing agile to plan-driven development approaches.

Typical agile methods require a high level of internal discipline [Coc02]. Even traditionalists such as Paulk assess that "agile methodologies imply disciplined processes" [Pau02].

This internal discipline has to be applied by the ones who do the actual work in the TDD project [BB03]. The developers have to perform the right activities without being controlled by management. This contrasts with the external discipline applied by more heavy-weight methods. For example, the Capability Maturity Model for Software (SW-CMM) [Hum89] suggests the establishment of a process improvement group that enforces the external process discipline by setting up project tracking mechanisms.

While agile approaches require a high level of internal discipline, people are very bad at being disciplined, being consistent, and following instructions [Coc01] [Hig02]. TDD offers many areas that challenge personal discipline, e.g.:

- Every development cycle starts with a test. Not even small additions are allowed without having completed a test first. This rule may be hard to follow, especially in cases where a developer very clearly sees the solution but needs to invest much more time for the creation of a test case first.

- The application may only implement what is needed to satisfy the tests and may not contain duplication. It is hard to always perform the third basic development step thoroughly. The tests are green and so why should one bother? Further, developers might only be tempted to perform code reorganization in a burst if the signs of decay have grown too big.

- Defect reports usually trigger an investigation for the defect's reason. Being at the spot of defect, a developer might be tempted to "fix the bug" directly. TDD, however, demands the reproduction of the defect in the test harness. Jeffries *et al.* [JAH00, p. 161] suggest to talk about Incident Reports, or Customer Information Requests instead of defects. They treat the defect resolution as regular development activity, applying the basic development cycle which includes the writing of a test first.

Agile approaches apply a rich set of feedback mechanisms that help to maintain a high level of discipline; they also help the developers learn from their own errors. TDD can be used for Extreme Programming (see Section 2.1.1) that explicitly encourages regular feedback on all activities. Section 3.4 discusses TDD process assessment that provides feedback to prevent discipline decay.

2.3 Agile Software Changes

Changing software is perhaps one of the most fundamental activities in software engineering [EGK+02]. Traditionally, software changes fall into one of several categories (see Section 2.3.1) and are seen as part of a system's maintenance phase. In agile software development, design (see Section 2.3.2) is the result of an evolutionary process (see Section 2.3.3) which applies unsafe and safe software changes (see Section 2.3.4).

2.3.1 Taxonomy of Software Changes

Traditionally, software changes during system maintenance fall into one of the following categories [AGM87], [Swa76]. *Adaptive changes* adapt a system to a new context (e.g. operating system change). *Perfective changes* are intended to improve the design, usability or non-functional characteristics of the system. *Corrective changes* fix faults in the system. This taxonomy emphasises the intention of a software change.

Newer definitions extend this notion by *Extending changes* that extend the functionality of an existing system and *Preventive changes* that prepare a system for future needs [Dum00].

The remainder of this section discusses these categories of software changes in the light of TDD and discusses related work.

Adaptive changes

Making a software system available for end users requires roughly three steps: (1) *development* which creates the application software, (2) *deployment* which moves the application onto the production environment, and (3) *production* which ensures the availability of the software system to the end user.

TDD, as a development style, does not deal with production and deployment. A development method such as Extreme Programming (see Section 2.1.1) cares about the continuous deployment of the application under development, however, TDD developers seek continuous feedback from production and deployment because their goal is to produce a running application. This feedback contains changes of the production environment that have to be respected in the development environment.

Perfective changes

Perfective changes are intended to improve the design, usability or non-functional characteristics of a system.

Improving the design is clearly part of the third basic development step in TDD (see Section 2.2.1). Such changes are constantly performed on the application under development. They would not be performed without the prior introduction of some new feature. In theory, after each basic development cycle, the application should have a very good design and therefore such design-perfecting changes cannot occur in a stand-alone like fashion.

Reports from practice, however, indicate that sometimes such phases are necessary. In particular, inexperienced teams seem to have problems performing all the perfective changes, along with the other change activities, as part of the basic development cycle [BC01].

Improving the usability or non-functional characteristics of a system would be seen as new feature requests. Such requests would have to run through the full basic development cycle. In the TDD sense, such changes are seen as extending changes (see Section 2.3.1).

All new features have to be driven by tests. In practice, this point of view is not always as easy. Testing some non-functional characteristics might be prohibitively expensive; for example, some errors in concurrent programs. Link and Fröhlich [LF03] assess that sometimes a literature study adds more to error prevention in concurrent programs than non-deterministic test suites.

Corrective changes

Corrective changes are meant to fix faults that have been documented in defect reports. A TDD team takes such defect reports as regular feature requests that run through the basic development cycle. Some TDD proponents even discourage the term “defect report” in favour of descriptions such as “Incident Reports”, or “Customer Information Requests” [JAH00, p. 161]. While this point of view has its own justification, TDD cannot ensure an error-free application. A test-driven application has to “green-bar” against the test suite only.

In practice, bug reports and feature requests are often stored in the same repository. See, for example, the Eclipse bugzilla defect database [URL:EclipeBugs].

From the point of view of the analyses proposed by this dissertation, corrective changes are indistinguishable from other changes. A new test reproduces the race condition and drives the corrective changes into the system.

Extending changes

Extending changes add new functionality to the system. The basic development cycle (see Section 2.2.1) ensures that for every new functionality, new tests have to be created. Those tests actually drive the new functionality into the application under development and cause the extending changes in the production code. Occasionally, new or extended abstractions (or design closure) result from these extending changes.

The term *extending* is ambiguous with respect to the Open-Closed principle (OCP). The OCP states that objects should be open for use but closed to further modification (see a detailed discussion in Section B.2). Extensions, in the sense of this taxonomy, can be either closure-preserving, closure-violating, or closure-extending. Note, that closure-preserving changes extend software entities while staying inside the given design closure. Closure-violating changes modify existing software entities and violate the existing design closure. Closure-extending changes extend the existing design closure so that future changes of the same kind can stay inside the design closure by simply extending the existing software entities.

Extending changes have to be performed very carefully, in order not to cross the border to preventive changes, as discussed next.

Preventive changes

Preventive changes prepare a system for future needs. To TDD developers, they are like a “red rag”. TDD demands to have the simplest design that fulfills the current test harness. Everything on top of this simplest design would smell of “Needless Complexity”.

For a TDD team, having the simplest design at any point in time is the best preparation for fulfilling future needs. Unless the customer explicitly specifies such preventive changes, a TDD developer does not try to predict the future.

Related Work

The seminal empirical work in the area of software evolution is by Belady and Lehman [BL76]. They studied 20 releases of the OS/390 operation system. Their observations at that time and later [LP80], [MDJ+97] led to the postulation of “laws” of software evolution dynamics: Continuing Change, Increasing Complexity, Self Regulation, Conservation of Organizational Stability, Conservation of Familiarity, Continuing Growth, Declining Quality, and Feedback System (in order of publication). The authors and others have provided data to validate these laws. The last law “Feedback System” stresses the importance of feedback loops to achieve significant improvement over any reasonable base. This observation of Lehman matches the aggregation of some agilists that agile software development is “about feedback and change” [WC03]. Lehman’s more recent FEAST project [MDJ+97] seeks to understand those feedback loops better in order to improve existing software processes.

Basili *et al.* [BBC+96] examined 25 software system releases of 10 different systems at the NASA Goddard Space Flight Center. One focus of the study was to assess the effort distribution for individual maintenance activities and for maintenance activities across the projects. The study used the taxonomy of Swanson, as described above, to categorize the change types. The goal of the study was to estimate the cost of maintenance releases of the software system. The study argued against small releases in the context of the software development environment. Methodologically, the authors assessed that "the combination of qualitative understanding and quantitative understanding has been invaluable."

Eick *et al.* [EGK+01] assessed code decay in large software systems by analyzing past change management data. In their conceptual model for code decay, a unit of code is termed "decayed" if it was harder to change than it should have been. They defined code decay indices (CDIs) which quantify symptoms or risk factors or predict key responses. The process compliance indices (PCIs) of this dissertation (see Chapter 6) are similar to CDIs in the sense that they quantify symptoms found in past change data. However, the PCIs support a TDD process assessment whereas the CDIs point to code units that need rework.

Kemerer and Slaughter [KS99] extracted a large number of software change event descriptions from histories or logs that were written by maintenance programmers each time they updated a software module in the application. They proposed a classification scheme similar to this taxonomy with more specific subcategories in each change category. The objective of their research was to provide data for an evaluation of existing scientific theories regarding software evolution (e.g. Lehman's "laws" of software evolution). They envisaged, that an understanding of software changes over time could inform current maintenance and development practice — however, they did not provide concrete advice in that respect. Kemmerer and Slaughter and others [BR00] point out difficulties in the collection of data concerning software evolution and the "lack of much existing theory and models" [KS99].

Lanza [Lan03] proposes a polymetric-view called evolution matrix to visualize software evolution. Lanza describes several specific patterns concerning how classes evolve over time. This technique can provide a fast, high-level overview of an application's evolution history.

In summary, all presented approaches analyze types of software changes on different levels of granularity in time. They all share the desire to improve understanding and hence improve software development processes by providing general advice to software developers. However, all described approaches fall short in giving specific technical recommendations in sufficient detail, as proposed by this dissertation.

This revision of software maintenance activities, in the light of TDD, supports the statement that "Software Maintenance Is Nothing More Than Another Form Of Development" [SKN+99]. TDD is a development style of continuous maintenance; therefore

the study of maintenance activities can support a TDD process assessment. The process compliance indices proposed in this dissertation give specific quantitative guidance to focus the qualitative work of a process assessor (see Section 3.1).

2.3.2 Agile Design

In the view of agile software developers, "the source code is the design" [Mar03, p. 87]; however, this thinking might not be shared by the broader software engineering community. Jack Reeves [Ree92] has already explored this idea in 1992 in his article "What Is Software Design?". He concluded that "the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings". For the practice of agile software development, this means that (1) only the source code can give definitive answers on any design question and (2) all design activities are directly reflected in the source code. Software development is seen as "mainly a design activity" [McB03, p. 96]. This attitude concerning design does not mean that more traditional design approaches should be neglected. Agile developers just take a different view on these approaches.

Design patterns solve common problems in object-oriented design [GHJV95]. They capture good designs that have stood the test of time in solving problems that keep reappearing, independent of the external context. Traditionally, design patterns are designed into the system design. Design pattern collections, in the form of books [BMR+96] [CS95b] or online catalogs [URL:J2EEPatterns] [URL:Hillside], provide a simple path from problem to solution. The observation can be made that developers put design patterns into the system's design with a mind-set that these design patterns will anticipate future needs automatically. This often leads to overly complex designs with too many abstractions that will not be needed. If the future needs are not known for sure, needless design pattern adoption complicates the design and constrains the ability of the application under development. Another problem is that an initially well-factored design pattern implementation might subsequently decay in response to future extensions or bug fixes to the application. In other words, design patterns are often put into a system design to solve a problem that does not initially exist and hence they then become the problem. The design pattern community has addressed this issue[4] by advocating piecemeal architectural growth [Cop99].

In TDD, design patterns are the result of continuous refactoring [Ker04]. In that sense, refactoring is seen as a type of design [Bec03]. An agile developer does not put design patterns into the system in the same manner as design pattern snippets into an UML diagram — called *up-front design* [Wak03]. Instead, design patterns emerge slowly from simple implementations which satisfy the test cases to abstractions that

[4] See special issue of IEEE Software about Architecture and Design, September/October 1999, Vol. 16, No. 5

extract duplication from simplistic production code — called *emergent design* [Wak03]. Also the TDD community regards design patterns as elements of good design. However, they are only included in the system design if they solve a problem that is visible in the code after the second basic development step.

The system design is the result of the application of a set of design principles that are described for object-oriented software — possibly leading to design patterns. Those design principles are the products of software engineering research and practice. Agile software development adopted these principles for meeting the challenge to create the simplest design that fulfils the current user requirements, which are manifested as tests.

One example is the Open-Closed principle (see Section B.2): "Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification." [Mey97] For developers that work test-driven, this means creating abstractions in the system that allow the closure-preserving extension of software entities in the system, as demanded by tests. The design closure should be closed against those kinds of changes that have not yet been demanded by the test harness.

The application of the design principles is triggered by a set of design smells [Mar03, p. 85]. Design smells are comparable to code smells [Fow99, p. 75] that trigger the application of refactorings. In TDD, code and design smells are the result of a quick and dirty implementation in the second basic development step (see Section 2.2.1). The third step is concerned with cleaning code and design and therefore involves an evolution of the system design.

In order to know which changes are likely as soon as possible, an agile development team tries to simulate changes. One possible process looks like this [Mar03, p. 105]):

- Write tests first. Testing is one kind of usage of the system. By writing test first we force the system to be testable. Therefore changes in testability will not come as a surprise later.
- Develop using very short cycles — days instead of weeks.
- Develop features before infrastructure and frequently show those features to stakeholders.
- Develop most important features first (which might be subject to change as well).
- Release early and often.

Strikingly, this process resembles the basic development cycle of TDD. The main difference is that Martin proposed an order on the sequence of requirements. TDD does not cover the area of requirements management. It is just a style of development to implement a given sequence of requirements.

In summary, in agile development the design is always as good as possible to fulfil the current requirements. The requirements are the sum of individual change requests

that have accumulated over time. In other words, the design is closed to the changes that have already occurred. The design cannot be closed to future changes. When these occur, the agile developer extends the design closure to fulfil the changed requirements. Typically, this means putting a new abstraction into the system that has not been there previously. This view of software development as continuous design effort might prevent a team from having to rebuild the system every few years, as recommended by Fred Brooks [Bro95].

2.3.3 Evolution of Code Base

The code base constantly changes over time. Every basic development cycle results in a set of changes on the local copy of the code base. Those changes are comprised as a single integration delta. This integration delta is integrated into the common code base. The result is the next integration version. This is the starting point for the subsequent development cycle.

Definition 2.1

Integration delta is the summary of the code changes of one basic development cycle to implement a feature.

Integration version is the state of the code base at the end of each basic development cycle.

Figure 2.1 shows an example of the code base evolution of the Money Example case study (see Section 8.1).

The Money Example is taken from [Bec03]; its integration deltas were entered manually into a version control system. For the other case studies in Chapter 8, the integration deltas were reengineered directly from the commit log of the underlying version control system. As the case studies show (see Section 8.4), this is a valid approach.

An integration delta is the implementation of a feature that is added to the code base through a basic development cycle. Eisenbarth *et al.* [EKS03] also tried to find features in source code; however, their approach was completely different. They combined dynamic and static analyses in their calculations; this approach runs execution profiles for usage scenarios of a program. Using concept analysis, they calculated the computational units that contributed to the implementation of a feature. It might be worthwhile to compare their results to our more simple approach, which is based on a system's integration history.

2.3.4 Safe Software Changes

Safe software changes in the context of TDD are changes to an application under development that do not need to be validated by tests. In general, refactorings are such safe

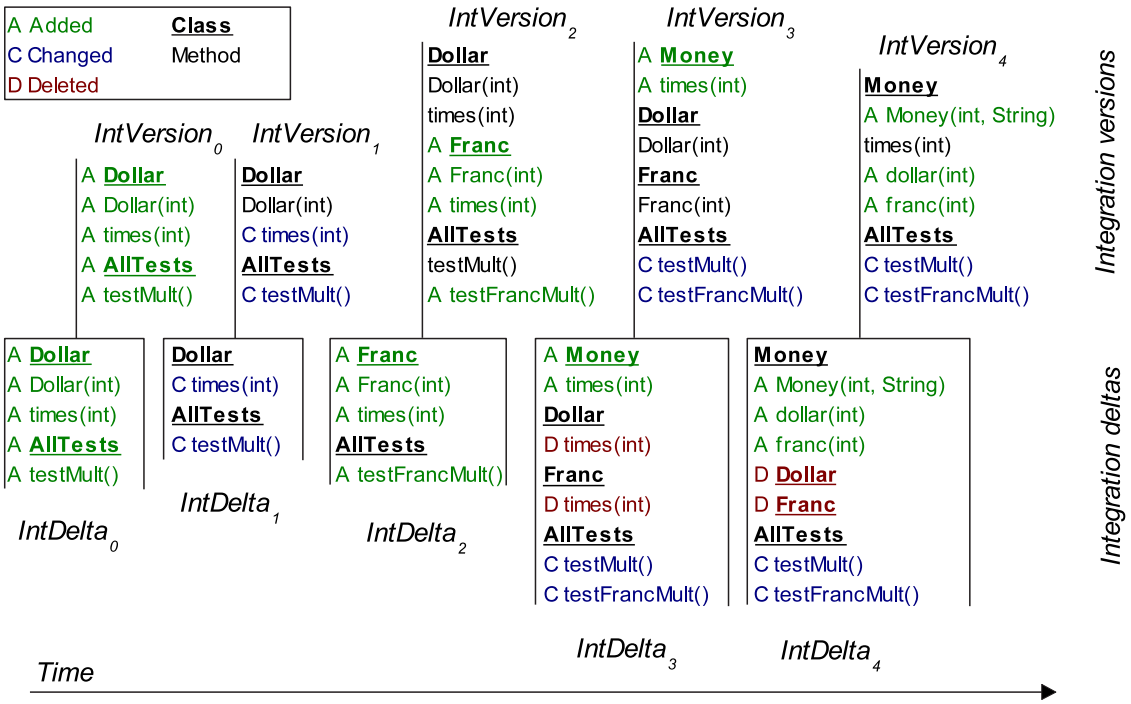

Figure 2.1: *Evolution of Money Example code base.* Integration deltas merge into integration versions over time. Each integration delta contains the changes of one basic development cycle. The integration versions represent the developer's view on the code base between each cycle.

software changes [Wak03, p. 7]. The discussion amongst TDD experts about when a software change is regarded as safe and thus does not need a test has not yet come to an end.[5] The remainder of this section discusses a number of safe software change types.

Refactorings

A refactoring "is the process of changing a software system in such a way that it does not alter the external behaviour of the code and yet it improves its internal structure" [Fow99]. The key idea here is to redistribute classes, variables and methods in order to facilitate future adaptations and extensions [MDB+03]. In essence, refactorings improve the design of existing code [Wak03] [OJ90] [Opd92].

Refactorings are a very important tool in TDD because they provide a structured way for the evolutionary design changes in the third step of the basic development cycle. TDD extends the notion of refactoring in an important way. A refactoring in the traditional sense "cannot change the semantics of the program under any circumstances. In TDD, the circumstances we care about are the tests that are already passing" [Bec03, p. 181]. In other words, in TDD a refactoring has to pass with respect to the existing tests, whereas in general a refactoring has to pass with respect to all possible tests.

Definition 2.2

Refactoring is a semantic-preserving change with respect to the test harness.

For example, the *Move Method* refactoring (see Section A.1.6) is triggered by the smell that a method "is, or will be, using or used by more features of another class than the class on which it is defined." [Fow99, p. 142] Listings 2.1 and 2.2 are taken from the QDox case study (see Section 8.2) and show a method before and after this refactoring.

Listing 2.1 (Method before refactoring)

```
public class JavaClass {
   ...
   private boolean isPropertyAccessor(JavaMethod method) {
   ...
     if (method.getName().startsWith("is")) {
       String returnType = method.getReturns().getValue();
       ...
     }
     ...
   }
}
```

[5] For a lively discussion see for example [URL:WikiUT]

The `isPropertyAccessor()` method is moved from the source class (`JavaClass`) to the target class (`JavaMethod`). References to entities of the source or target class have to be adapted. Possibly a new parameter that references the source class instance needs to be introduced. However, an existing parameter that references the target class instance can be deleted.

Listing 2.2 (Method after refactoring)

```
public class JavaMethod {
   ...
   public boolean isPropertyAccessor() {
   ...
     if (getName().startsWith("is")) {
       String returnType = getReturns().getValue();
       ...
     }
     ...
   }
}
```

Appendix A contains a list of refactorings referenced in this dissertation.

As discussed in Section 2.2.2, test code and production code are highly inter-dependent. Therefore a refactoring of production code might require the refactoring of test code as well, if the refactoring affects code that has been called from the tests. This would invert the order of changes in terms of the production code driving test code. Therefore some authors propose performing refactorings in the test code first [Pip02] [Pip03] [DM02]. Either way, test code and production code have a strong relationship which is sometimes visible within the refactorings. From a retrospective point of view, it is not possible to decide what has changed first — in the context of this dissertation, that order is irrelevant.

Refactoring participants

Sometimes, refactorings cause other methods to change as well. These other methods might not count as refactorings but rather as refactoring participants.

An example should help clarify the concept. Listings 2.3 and 2.4 show a refactoring participant before and after a change. The code is taken from the QDox case study (see Section 8.2).

Listing 2.3 (Refactoring participant before method move)

```
public class JavaClass {
   ...
   private void initialiseBeanProperties() {
     ...
```

```
      if (isPropertyAccessor(method)) {
        String propertyName = getPropertyName(method);
        ...
      }
    }
}
```

Now the methods `isPropertyAccessor()` and `getPropertyName()` move to the class `JavaMethod` — driven by a test in the current integration delta. This means that all clients of these methods have to change. In this case, the two method calls need an additional qualification to an instance of `JavaMethod`.

Listing 2.4 (Refactoring participant after method move)

```
public class JavaClass {
   ...
   private void initialiseBeanProperties() {
     ...
     if (method.isPropertyAccessor()) {
       String propertyName = method.getPropertyName();
        ...
     }
   }
}
```

Refactoring participants[6] capture such changes; within this dissertation these are regarded as safe changes.

Definition 2.3

Refactoring participant is a method that has to change, because a related[7] method undergoes a refactoring.

Change participants

Not only refactorings can cause other methods to change. A test of an integration delta can drive the signature change of an API method. The changed method is covered by the test but not all clients of that changed method are necessarily covered by tests of the current integration delta.

An example should help to clarify this concept. Listings 2.5 and 2.6 show a change participant before and after a change. The code is taken from the QDox case study (see Section 8.2).

[6] Eclipse provides an API for rename refactoring participants. Contributed plug-ins can register for Java-like artifacts (e.g. Java Server Pages) to be informed when a related Java class undergoes a rename refactoring. They can then apply this rename refactoring to the identifiers in the Java-like artifact.

[7] See definition of relationship in glossary.

Listing 2.5 (Change participant before change)

```
public boolean isA(String fullClassName) {
     Type type = new Type(fullClassName, 0, getParentSource());
     return asType().isA(type);
}
```

Now the third parameter type to the constructor of `Type` changes from `JavaSource` to `JavaClassParent` — driven by a test in the current integration delta. This means that all clients of the constructor have to change. In this case from `getParentSource()` to `this`.

Listing 2.6 (Change participant after change)

```
public boolean isA(String fullClassName) {
     Type type = new Type(fullClassName, 0, this);
     return asType().isA(type);
}
```

Change participants capture such changes. In general, change participants cannot be regarded as safe. The change participant in this example is taken from practice and was regarded as a safe software change by its developers, who are TDD experts. Only the invocation argument changed — the remaining method stayed unchanged. Within this dissertation, such a change participant is regarded as a safe change. It is actually the only change participant type for which TddMentor implements detection code. It is an open question if other types of safe change participants exist.

Definition 2.4

Change participant is a method that has to change, because a related method changes and that related method is covered by tests.

A refactoring participant is some kind of change participant. It differs in the fact that the inducing method itself was undergoing a refactoring — not only some change.

Accessors

Accessors are methods that get and set a specific instance variable. Listing 2.7 shows an example of a pair of accessors for the variable `amount`.

Listing 2.7 (Accessor methods)

```
public int getAmount() {
   return amount;
}

public void setAmount(int value) {
   amount = value;
}
```

In the context of this dissertation, I have assumed the creation and modification of accessors to be safe. Accessors play an important role in the calculation of some process compliance indices, as discussed in Section 6.2.

Technical Methods

Some Methods are needed for the sake of easier development. For example Beck [Bec03, p. 47] writes a `toString()` method without a test. This method enables the debugger in an IDE to print the value of a variable in a comfortable way. This method does not need to be tested. However, he does note this as an exception.

I use the term *technical methods* for such methods.[8] In the context of this dissertation, I have assumed the creation and modification of technical methods to be safe. Technical methods play an important role in the calculation of some process compliance indices as discussed in Section 6.2.

2.4 Summary

This chapter has presented Test-Driven Development as a style of agile software development. TDD is an evolutionary process of applying agile software changes that are driven by tests. The developers have to be very disciplined in performing given TDD development practices. The next chapter introduces assessments that help keep up a high process discipline.

[8] Some developers prefer the term “tool method”.

Chapter 3

Software Assessment

Weinberg [Wei93] describes a cybernetic model of a software development project. For Weinberg, cybernetics is the science of aiming, i.e. of how to best follow a running target. As shown in Figure 3.1, a software development system produces the software and other outputs (e.g. greater competence with a specific technology, etc.). The controller can change the incoming requirements and resources. This input can range from physical resources such as computers to training courses. The feedback mechanism allows observation of the actual state of the software development system.

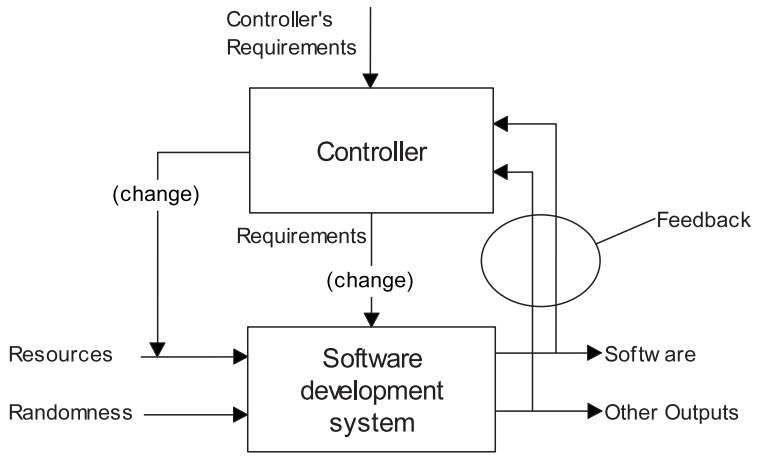

Figure 3.1: Cybernetic model of a software development project. Feedback allows to control the software development system [Wei93].

The importance of feedback in order to allow control and thus to make orderly progress is widely accepted [HSW91] [Bro95, p. 121] [Dem82b]. This chapter explains a number of techniques that have emerged over time that enable the retrieval of

feedback from an actual software development system on the performed process and the produced artifacts.

Software process assessment (see Section 3.1) is a structured approach for studying the methods, tools, and practices used in a software development system. Software metrics (see Section 3.2) express process and product attributes in numbers. Methods and techniques for software inspection (see Section 3.3) help understand and improve existing source code and development practices. All these disciplines find their application in TDD process assessment (see Section 3.4).

3.1 Software Process Assessment

No matter how much a programmer is aware of the development process — he/she always executes a process to produce artifacts [McC93]. Software process assessments study the methods, tools, and practices used by human beings for creating software [Jon00, p. 21]. These assessments are normally carried out on-site by independent or in-house process assessors. Process assessors normally gather the assessment data by means of structured interviews with managers and technical personnel. Gathering assessment data can also involve code inspections (see Section 3.3).

One goal of a process assessment is to identify an organization's capability of producing high-quality software or simply the current state of a development project.

The Capability Maturity Model for Software (SW-CMM) is one such approach at the organizational level. It was defined by the Software Engineering Institute (SEI) to evaluate software organizations with respect to their capability to produce high-quality software[1] [Hum89] [Hum98a] [PM90]. It defines five stages of process maturity: initial, repeatable, defined, managed, and optimizing [PWC95]. A SW-CMM assessment by an external assessor can lead to the certification of an organization's SW-CMM level.

The Capability Maturity Model Integration (CMMI) extends the SW-CMM concept for engineering processes in general [CMM02]. It has added several process areas that encourage agility and customer collaboration [BT03].

Similar to SW-CMM is Software Process Improvement and Capability Determination [URL:SPICE]. It is a major international initiative to support the development of an International Standard for Software Process Assessment [ISO98] [ISO03].

Process assessments for individual development projects often involve project reviews or retrospectives [Ker01]. Such retrospectives can be part of a continuous learning experience [Ker04]. A different form of process assessment is applied by in-process mentors that accompany a development team for some time. Their goal is to identify

[1] "The US Air Force asked the SEI to devise an improved method to select software vendors. [...] A careful examination of failed projects shows that they often fail for non-technical reasons. The most common problems concern poor scheduling and planning or uncontrolled requirements." [Hum98a]

weaknesses in the development practices and to improve the process.[2]

The objective of process assessments is to assess the quality of a development process. The reason for performing process quality ensuring activities is the belief that a high-quality process leads to a high-quality product [Som92, p. 592]. Assessing the compliance to defined development practices is an important part of the quality ensuring activities for TDD.

A process assessment is a diagnostic study and is therefore no therapy in itself. It can identify and rank problems that need to be solved and hence can lead to a successful therapy program. Assessing an organization's software process before launching an improvement effort is crucial [MPHN02, p. 185]. A process assessment is typically embedded into a software process improvement effort as depicted in Figure 3.2. For example, the Software Engineering Institute provides a handbook for software process improvement with SW-CMM as assessment method [McF96]. The first step of any process improvement effort is to diagnose the problems in the current set of practices [Jon00, p. 38] [MPHN02, p. 185].

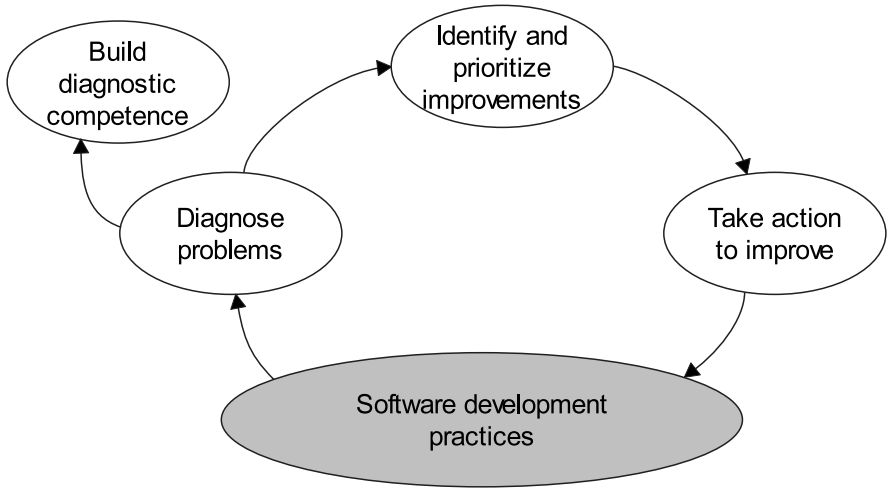

Figure 3.2: *Basic Software Process Improvement cycle.* Software process assessment is typically embedded in a broader process improvement effort. Software Process Improvement should focus on problems that are perceived in the current process. (Adapted from [MPHN02])

The diagnosed problems can either be driven by actual needs of the project's stakeholders or they can be driven by weaknesses within the performance of the development practices.

[2]Companies such as ObjectMentor have built a business model on providing development teams with Extreme Programming transition mentors and coaches.

TDD defines a set of practices. Failure to comply with these practices can at least limit the advantages of this development style. The approach proposed in this dissertation has concentrated on this second type of problem — the identification of problems in the performance of given practices.

Tool Support

Software process measurement tools support software process assessment. Most current tools concentrate on process attributes that are not measurable automatically. Thus, to a large extent, process measurement tools offer entry, calculation, storage and visualization of process assessment data [Sim01, p. 38]. Even worse, most development organizations do not even bother to collect quantitative data during software development [Jon00, p. 34]. This dissertation proposes a tool that measures, automatically, some of the process attributes from past source code changes in order to support a process assessment (see Chapter 7).

3.2 Software Metrics

DeMarco [DeM82a, p. 3] observed that "You can't manage what you can't control, and you can't control what you don't measure."

The term software metric refers to any measurement related to software development [McC93]. Metrics are defined for measuring the software development process and the software product [Bal98] [Pow98]. They are defined to map certain software properties to numbers [DD99]. However, as Balzert [Bal98, p. 232] notes, software metrics can only hint at possible anomalies (in a negative or positive sense). If used wrongly, metrics can do more harm than good, such as when developers start to optimize the numbers instead of following the goal for which the metrics provide the numbers.

Control and Predictor Metrics

Sommerville [Som92, p. 598] sees two kinds of metrics. Control metrics measure process attributes to control the development process (e.g. effort expended and disk usage). Predictor metrics measure product attributes in order to predict associated product quality (e.g. complexity metrics for estimating the maintainability of a program). Figure 3.3 shows those two metrics in the context of a development project.

Metrics Based on Kind of Entity

Another possible software metrics classification is the kind of measured entity. This comprises conceptional entities (e.g. processes), material entities (e.g. persons), and

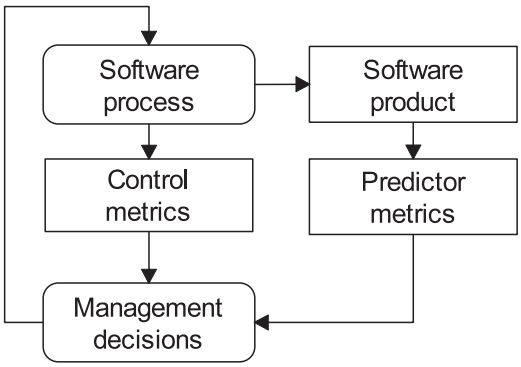

Figure 3.3: *Predictor and control metrics.* Control metrics measure process attributes to control the development process. Predictor metrics measure product attributes to predict associated product quality. [Som92]

ideal entities (e.g. programs) [FP96]. Accordingly, there are three kinds of software metrics:

- process metrics — for software changing activities
- resource metrics — for process-supporting resources
- product metrics — for artifacts created by the development process

Connecting Measured to Controlled Entities

The two types of metric classifications suggest two dimensions of software metrics. The vertical axis identifies the entities that are *measured*. The horizontal axis identifies the entities that are *controlled*. The metrics of Fenton and Pfleeger cover the vertical axis. The metrics of Sommerville connect entities on the vertical axis to entities on the horizontal axis as shown in Table 3.1.

	Control process	*Control product*	*Control resources*
Measure Process	Control metrics	"Conway's law"	Control metrics
Measure Product	Process compliance indices	Predictor metrics	Predictor metrics
Measure Resources	Control metrics		Control metrics

Table 3.1: *Measured versus controlled entities.* Different metrics cover different aspects in connecting measured entities to controlled entities, as explained below.

Control metrics measure the development process and the resources needed to perform the process. Their goal is to optimize both the process and the resource consumption. Predictor metrics measure the produced artifacts in order to predict their behavioural attributes, possibly leading to product changes. One goal of such a product optimization is to minimize future maintenance efforts; hence to control the resources at a future point in time. Another goal of such product measures could be to quantify developer expertise [MH02] and hence optimize the selection of development resources. Conway's Law states that a system's architecture will mimic the organization that built it. In other words — if you have three teams building a compiler, you will get a three-pass-compiler [Con68] [Bro95]. Coplien [Cop99] observed that "[a]rchitecture is not so much about the software, but about the people who write the software."

This matrix that connects measured entities to controlled entities has two gaps. Process compliance indices, as proposed in Chapter 6, fill one of these gaps. They provide the means to measure the product for the purpose of controlling the development process.

In order for a metric to be useful, it must be derived from the development method. For example, inappropriate use of software complexity measures (see Section 3.2.2) can have large damaging effects by rewarding poor programming practices and demoralizing good programmers. Software complexity measures must be critically evaluated to determine the ways in which they can best be used [KST+86]. The Goal-Question-Metric paradigm (see Section 3.2.3) is a way to derive concrete metrics from the project goals. The Orthogonal Defect Classification (ODC) paradigm [CBC+92] [BKS98] [BMK02] provides a schema to capture the semantics of software defects. It aims at diagnosing various aspects of software development such as design, development process, test and service.[3]

The remainder of this section reviews various software metrics and discusses their relation to TDD.

3.2.1 Software Process Metrics

Every activity during software development can be seen as a process. The measurement of such processes aims to provide a better understanding of the process efficiency [Sim01, p. 21]. Typical process metrics include defect discovery rate, defect backlog, test progress [BBS02], and communication metrics [DB98].

Most software process metrics (or software engineering metrics) were designed for analyses on an organization-wide rather than an individual level [JKA+03]. The measurement approaches focus on satisfying the information needs of project managers [MC02] [EC02].

[3] For a current description of ODC see [URL:ODC].

SW-CMM sees 17 key process areas (KPAs) [PWC95]. During a SW-CMM certification, feedback is provided for each KPA, that evaluates the extent to which the organization meets pre-established criteria in the areas of activities and infrastructure, institutionalization, and goals. SW-CMM does not measure the development process itself but rather provides a measure of the capability of a development organization to perform a process.

Compared to Agile Software Development, SW-CMM puts more focus on organizational maturity than on producing running and high-quality software [Orr02]. Where SW-CMM sees quality as "following defined practices and plans", agile methods see quality as "essential customer satisfaction" [BT03, p. 5].

With the Personal Software Process (PSP)[4], Humphrey [Hum95] [Hum98b] introduced metrics on an individual level. The primary goals of the PSP are to improve project estimation and quality assurance. The collected metrics include size, time and defect data. However the collection of these metrics has to be performed by the developer in parallel with the regular development activities. The commonly observed "natural resistance" [Wel93] of software developers compromises many measurement efforts. The Team Software Process [Hum98c] and Collaborative Software Process [Wil00] extend this approach to development teams.

Johnson *et al.* [JKA+03] estimate that the adoption of continuous data gathering by software developers, as suggested by PSP, is very low in practice. They ascribe this low acceptance to the overhead and context switch associated with the data gathering, which would require a very high level of personal discipline. The authors propose the use of the Hackystat system which collects the required data automatically. In summary, their message is that metrics collection on an individual level can only work properly if it is not associated with overhead for the individual developers.

Weinberg [Wei93, p. 30] proposed an observational model to make sense of the observed artifacts. This model guides the observation during the feedback process. In detail, an observation model must tell:

- what to measure
- how to interpret this data
- how to find the significant observations
- which control actions to take

An effective observation model should be able to identify the state of a system. Knowing this state allows a controller to take appropriate control actions.

TDD, by itself, does not define the software process metrics directly. If it is embedded in a development method such as Extreme Programming (see Section 2.1.1), the

[4]Personal Software Process and PSP are registered service marks of Carneggie Mellon University.

number of implemented user stories or task cards measure the progress of a development team. “Team velocity” measures the productivity of a development team to implement user stories. Elssamadisy and Schalliol [ES02] report a set of “process smells” for Extreme Programming, ranging from organizational observations to technical issues. Their smell of large refactorings is clearly due to a TDD process violation. Section 3.4.2 lists some TDD-specific measurements.

3.2.2 Software Product Metrics

Software product metrics are calculated from the produced artifacts of a software development process. Typically, their calculation can be performed completely automatically. Within a development project, their goal is to predict attributes of the produced system, such as maintainability and understandability (e.g. complexity metrics). Some metrics try to rate the design quality (e.g. coupling metrics). Not only the production code, but also the test harness can be the subject of software product metrics (e.g. test coverage metrics).

Another use of metrics is in reengineering and research about software evolution [DD99], where they support an understanding of existing systems. For example, change metrics can be used to find refactorings [DDN00] (see also Section 5.4).

Complexity metrics

Complexity metrics try to rate the maintainability and understandability of a system [HS96].

Lines Of Code (LOC) is perhaps the most basic metric. The size of software entities can be seen as an indicator of their complexity. In practice, however, the calculation of LOC bears many ambiguities. For example, LOC could be defined either as a physical line of code or as a logical statement [Jon00, p. 69]. Humphrey [Hum97] provides a comprehensive discussion of measurements based on LOC.

Weighted Method Count (WMC) is another complexity metric. This metric sums up the complexity indices for all methods of a class. These complexity measures of the individual methods are usually given by code complexity metrics such as LOC or McCabe cyclomatic complexity [CK94] [CS95a].

The validity of such simple measurements, as indicators for actual program complexity, is questionable. Therefore, some authors suggest complexity vectors which combine several attributes [Ebe95].

Belady and Lehman [BL76] derived a “law of increasing software complexity” by observing a number of software changes in an evolving system (see also Section 2.3.1). Mattsson [Mat00] estimates the amount of change in future releases of a software system based on the system’s change history.

Fenton [Fen94] shows that a search for general software complexity metrics is doomed to fail. However, measurement theory helps to define and validate measures for some specific complexity attributes.

One of the goals of TDD is to keep a program maintainable and understandable. New features are added incrementally. The costs for adding new features must not become prohibitively expensive. Therefore a TDD team must keep the complexity of the application under development low. XP expresses this desire in its practices of "Simple Design" and "You Aren't Going to Need It". This goal is achieved by driving the code with tests and continuous refactoring. However, as of today, there is no comprehensive empirical data available to support this claim.

Coupling and cohesion

Coupling and cohesion metrics try to rate the design quality of an application.

The focus of coupling models is to see how strongly components are tied together [SMC74] [YC79]. Coupling models exist on different levels of granularity, such as objects [CK94] and packages [Mar03]. It is seen as a good design if software entities that implement separate concepts have few dependencies between each other.

Cohesion models describe how closely the elements of one component (typically a class) are related to each other [CK94] [BK95]. It is seen as good design if software entities that add to the implementation of the same concept have strong relationships among each other.

TDD nearly automatically leads to a decoupled design [Mar03, p. 24]. In order for the production code to be testable, it must be callable in isolation and decoupled from its surroundings. The act of driving the production code using tests forces the developer to decouple the application design. While TDD reduces the coupling of production code, it introduces a coupling between test and production code. This coupling might impact upon development and production. For example, in order to run the tests against some test data, it must be possible to provide this data programmatically.

TDD also has an important effect on cohesion. The Single Responsibility Principle (see Section B.1) directs a programmer to the separation of coupled responsibilities. Robert Martin paraphrases this principle with: "A class should have only one reason to change." Each responsibility is an axis of change. Mixing responsibilities means mixing axes of change. If the implementation of a new feature touches one such responsibility, then — in theory — it will be extracted into a new abstraction by removing duplication in the application (see Section 2.2.1). Separated responsibilities would be decoupled — thus increasing the cohesion of the separated entities.

As of today, the literature does not provide comprehensive empirical evidence concerning these two positive effects of TDD on coupling and cohesion.

Test coverage

Test coverage[5] is the percentage of elements required by a test strategy that have been exercised by a given test harness [Bin99, p. 52]. Many different test coverage models have been proposed [KFN93] [Bei93] [ZHM97] [Mil88]. Test coverage is either specification-based or code-based. Some of the most well-known code-based coverage models are as follows:

Statement Coverage is the percentage of all source code statements that were touched at least once during a run of the test harness. A statement coverage of 100% does not guarantee the absence of errors.

Branch Coverage is the precentage of all prossible branches in a program that were touched at least once. It is a stronger criterion than statement coverage but still does not guarantee error free code.

Path Coverage is the percentage of all possible paths — combinations of branches — that were touched at least once. Still a 100% path coverage does not guarantee error-free code.

Commercial tools typically calculate statement coverage [LF03]. The capabilities of more sophisticated test coverage tools are limited by polymorphism and dynamic class loading. The use of test coverage tools is vigorously discussed in the TDD community. On the one hand, a program emerging from a TDD process should have a very high test coverage. On the other hand, if controlled by a tool, people tend to optimize the metric rather than the goal [LF03]. And even worse, a high test coverage does not guarantee that the unit tests really test the core of an application and express the intent of the user stories appropriately. Low test coverage is, however, an indicator that the developers did not work properly.

Code-based test coverage tools can either work dynamically or statically. Dynamic tools observe a program while executing its test harness (e.g. Clover [URL:Clover], Hansel [URL:Hansel], JXCL [URL:JXCL]). Some tools (such as Clover) instrument the source code by weaving in coverage measuring aspects. Other possibilities (at least in Java) are instrumentation of byte code or the use of the Java Virtual Machine debugging interface. Static tools (e.g. NoUnit [URL:NoUnit]) calculate the test coverage by static analysis of the source code only. They use models such as control flow graphs [FOW87], program dependence graphs [BH93], or program slicing [Bin98].

[5] Test coverage is sometimes called test adequacy criteria since it expresses how adequately a test harness tests entities of a program.

Other Software Product Metrics

Halstead [Hal77] was one of the first to describe source code metrics. His metrics work on a very low level. They take into consideration operands and operations to calculate some numbers which are meant to indicate the complexity of a software module. Besides the critique they received from [HF82] they do not seem to be of great value for finding potential process compliance indicators.

3.2.3 Goal-Question-Metric Approach

Organizational assessments such as SW-CMM (see Section 3.2.1) start with a set of practices and issues that are considered to be critical in the production of software. They then evaluate the extent to which interview and documentation evidence supports the presence or absence of required criteria elements. Experiences with such criteria-oriented assessments are mixed. For example, some software organizations aimed at optimizing the assessment scores but did not achieve meaningful performance improvements [WBP+02].

Basili and Weiss [BW84] [WB85] proposed the Goal-Question-Metric (GQM) approach for stepwise refinement of a quality model. "GQM is based upon the assumption that for an organization to measure in a purposeful way, it must first specify goals for itself and its projects; then it must trace these goals to the data that are intended to define those goals operationally; and finally it must provide a framework for interpreting the data with respect to the stated goals." [BCR02]

The measurement model has the following three levels (see Figure 3.4):

Goal Goals are defined in terms of purpose, perspective and environment. The objects of measurement are products, processes, and resources (see classification of metrics above).

Question For the objects of measurements, questions try to characterize these objects in terms of selected quality issues.

Metric Each question is associated with a set of metrics which should answer the question in a quantitative way. Metrics can be objective or subjective and can be shared among questions.

GQM has found its application in practice and research. For example, Demeyer *et al.* derive change metrics from the goal to detect refactorings [DDN00]. The Avaya Software Assessment Process [WBP+02] is a goal-oriented process that borrows many concepts from GQM. It combines structured interview with the analysis of change data, combining aspects of SW-CMM assessments with quantitative analyses.

Over time, GQM has evolved into the Quality Improvement Paradigm (QIP). This consists of a mechanism for feedback in a software development organization in order

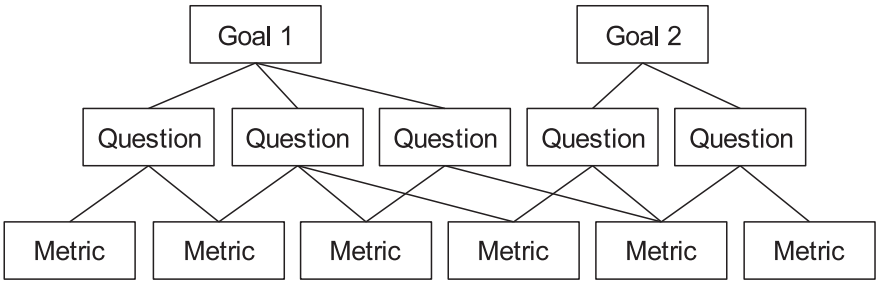

Figure 3.4: *Goal-Question-Metric Hierarchy.* For every goal, a number of questions are derived. These questions are answered using a combination of software metrics. [BCR02]

to ensure that results learnt from one project could be applied to future projects. The next evolutionary step was the Experience Factory (EF). EF is a learning organization model used in increasing the understanding of development problems, characterizing the current environment, and developing and testing proposed alternatives to solve these problems [BMPZ02].

Section 6.1 provides the GQM deduction for the metrics in this dissertation.

3.2.4 Measurement over Time

Some metrics only make sense if they are observed over time. For example, the plain number of touched modules in a new release of a software system contains little valuable information. However, if the fraction of touched modules continuously grows for successive releases, then this observation can be taken as a sign for increasing complexity. Based on this observation, Belady and Lehman [BL76] stated the "law" of increasing complexity for evolving software systems (see also Section 2.3.1).

Other metrics change their meaning if they are depicted over time. For example, Weinberg [Wei93, p. 67] observed that "One measurement of user satisfaction is, of course, a product measure, but plotting the trend of several measurements of user satisfaction over time or across different products creates a process measure."

Biemann *et al.* [BSW+03] analyzed a large number of versions of some systems. Their interest was to explore the relationship of software changes to design structures. They provide insight into the actual use of design patterns over the course of a system's evolution.

Weiss *et al.* [WBP+02] extract data from modification requests (MRs) that describe software changes to a system. The MRs are held in a database, parallel to the source code version control system. Examples of extracted data include time, effort, MR extent, developer experience, and history of files being changed. Some information is also

extracted from the source code version control system, however the authors do not perform any source code measurements. They perform statistical analyses on the raw data, possibly over a time span of several years. The authors observed that it is critical to discuss the interpretations of the data with those familiar with the project's history. In parallel to the analysis, the data are stored in a "data mart" for future comparisons. This data mart is comparable to the Experience Factory paradigm (see Section 3.2.3).

Mockus and Herbsleb [MH02] analyzed change histories to quantify developer expertise. Similar to Weiss *et al.* they extracted data from a modification request (MR) database and the source code version control system. Data from the version control system was only used to correlate source code deltas to the author of that delta.

Gall, Jazayeri *et al.* [GJKT97] [GHJ98] applied several kinds of analyses of change data of a Telecommunications Switch System to find potential architectural inadequacies.

Kan *et al.* [KPM01] collected in-process metrics over time which helped to decide when a product could be shipped. However, they only collected process metrics that could not be extracted from the source repository.

Axes of Aggregation

Eick *et al.* [EGK+01] pointed out the importance of positioning an empirical study on the two axes of aggregation: the axis of module decomposition and the temporal axis of change.

On the axis of module decomposition, you find entities such as line, type, file, package, subsystem, system. Most approaches to software product measurement work on this axis for the most recent version of the system (see also [Sim01]).

Software evolution studies are positioned on the temporal axis. The entities on the temporal axis range from simple file deltas, modification requests, initial modification requests, feature requests [EGK+01] to product releases [BL76].

For this research on TDD, the interesting units of aggregation on the temporal axis are the integration deltas. They contain the changes from one basic development cycle. On the axis of module decomposition, this research deals with individual methods and classes. At the time of writing, no higher-level abstractions were taken into account, and no further distinction between regular feature/modification requests and production releases was made.

Tool Support

A first set of generally available tools has appeared that allow the ongoing collection of metrics over time.

For example, JMetra [URL:JMetra] can collect some simple metrics over time. It can even be configured for metric continuity over package and class name changes.

However, there are no heuristics implemented for an automatic mapping of changed names and this has to be configured manually.

The tool support proposed by this dissertation allows the reconstruction of past integration versions and deltas from a CVS source code repository [URL:CVS]. Identification of the integration deltas is the basis for the proposed process compliance index calculations (see Chapter 6). TddMentor applies a heuristic, to connect renamed or moved entities, across such changes. Furthermore, the reconstruction of integration deltas can be used to calculate the completion rate of "todo pragmas" in the source code.

3.3 Software Inspection

Software inspections are peer reviews of a programmer's work to find problems and improve quality [Hum95, p. 171]. They are based on Weinberg's [Wei71] concept of egoless programming. Weinberg refers to cognitive dissonance as the human tendency to self-justify actions. This tendency limits our ability to find errors in our own work. Software inspections, by peers, help overcome this limitation.

Software inspections were originally introduced by Fagan [Fag76] as a method of finding errors in design and code. In 1976, Fagan assessed the beneficial influence of software inspections for a "constant process improvement" through removal of systemic defects [Fag86]. This lead to the dual objectives of the inspection process [Fag01]:

- Find and fix all defects in the product, and
- Find and fix all defects in the development process that gives rise to defects in the product (namely, remove the causes of defects in the product).

Fagan emphasised the importance of tool support for software inspections (like syntax error compiler flags and LINT-type compiler tools) so that the human inspectors can focus their efforts on the more intellectually challenging aspects.

Despite the dual objectives of the inspection process, recent empirical research has focused mainly on detection of software defects of the product [PV94].

The types of software artifacts submitted to an inspection process typically includes requirements documents, design, code, test cases and more [GG93] [Won02].

The discipline of code reading has established a number of techniques that help with the understanding of large amounts of source code. See, for example Spinelli [Spi03], who compares code reading with the way an engineer examines a machine. Code reading not only enforces programming standards, but facilitates a quicker understanding of the existing system. Understanding Open Source software is just one possible area of application.

Basili *et al.* [BGL+96] propose to read artifacts from a specific perspective (e.g. tester, developer, user). They evaluated their approach of perspective-based reading for

defect detection in requirements documents. The authors see strong similarities to that of reading source code.

Dunsmore *et al.* [DRW02] extend reading techniques to object-oriented software. Object-oriented software is inherently less accessible for reading techniques due to its delocalized nature. The Famoos project [DD99] collected some patterns for object-oriented software reverse engineering.

For a TDD developer, software inspections and reading techniques are tremendously important, because in TDD the source code is the only reliable source of reference. In the context of Extreme Programming, programming is performed by pairs of developers. This so-called pair-programming [WK02] is a kind of continuous software inspection or peer review. Another important XP practice is "collective code ownership", which leads to a continuous inspection of other team members' code.

For a TDD process assessor, inspection techniques are even more important, because he or she is confronted with all the source code as a whole, whereas a TDD developer can build up knowledge of the source code incrementally. The tool proposed by this dissertation helps a TDD process assessor focus the assessment effort on those areas in the source code that are likely to be troublesome.

3.4 TDD Process Assessment

A team of developers might state that they work in a TDD manner while the process actually performed proves to be different. As cited in [Pau02], Robert Martin reported an incident of such method misuse at the 2001 XP Universe conference. Someone from a software organization claimed they were doing Extreme Programming with the argument that they did not document anything. Process assessment cannot only help uncover such massive method misuse but can also help uncover more minor forms of misunderstanding of a development method.

Maximilien and Williams [MW03] document the assessment of a TDD project. The developers were inexperienced in TDD and needed an introduction to this style of development. The project was closely monitored because it was the first TDD project in this development's organization. One experience of the authors was that the developers needed continuous encouragement to add unit tests to the system, for new features and bug fixes. Therefore, the process assessors set up a monitoring system that reported the relative number of unit tests for every subsystem. The developers needed a third of the overall project time until they appreciated the test harness.

The remainder of this section discusses retrospectives and measurements as concrete techniques for performing a TDD process assessment.

3.4.1 Retrospectives

One of the major contributions of agile methodologies is the emphasis on the team to organize itself. This contrasts to most traditional development methodologies. Cockburn [Coc01] argues that the people involved in the development process are the most important factor in guaranteeing the success of a project. Boehm *et al.* [BT03] emphasise that "agile retrospectives and reflections enable continuous process and technical improvement."

The self-organization can be driven by interviews, post-increment workshops, post-morta, post-parta, project reviews, and study groups [Coc98] [Ker01] [Ker04] [Hig00]. They aim at harvesting the experiences made by the team members throughout the project. Some agile methodologies advocate *pair programming* as an extreme form of such reviews where two programmers constantly review each other [WK02] and *collective code ownership* where the developers review the other team members' code continuously.

Such retrospectives typically concentrate on the established tracking mechanisms and personal experiences of the team members [Ker01]. Usually, there are only a few documents available describing the project's evolution because agile methods, especially TDD, value running software over comprehensive documentation. The collected assessment data are more qualitative than quantitative.

In this spirit, TDD retrospectives are comparable to traditional process assessments which collect assessment data by interviewing project members. The agile manifesto [All01] adds the principle of self-organization, where teams reflect at regular intervals on how to become more effective.

3.4.2 TDD-Specific Measurements

The retrospective activities outlined above provide more qualitative observations about the state of a development process. TDD-specific measurements allow the derivation of more quantitative observations from the produced source code. Here is a list of some TDD-specific measurements:

Test Coverage For Integration Deltas: Typically, current test coverage tools concentrate on determining the test coverage of the most current integration version. Such tools neglect the process aspect of building a highly covering test harness, piece by piece.

Every change in the production code is driven by tests. Hence every production code change in every integration delta is covered comprehensively by the tests that were added or changed in that integration delta. The only exception are refactorings and other safe software changes (see Section 2.3.4). They do not have to be covered by tests.

Some advanced TDD developers regularly generate test coverage reports during development [Ast03, p. 419], for example as part of the nightly build process. Such regular reports would identify process-violating integration deltas very quickly. However, such practice cannot be expected commonly. A retrospective analysis of past integration deltas reveals similar insight as a continuous observation.

This metric is actually one of the PCIs discussed in Chapter 6.

Test Code Versus Production Code Ratio: Beck [Bec03, p. 78] observed that "You will likely end up with about the same number of lines of test code as model code when implementing TDD." Astels [Ast03, p. 418] reports a typical ratio of test code to production code between 1:1 to 2:1.

This test code must be available during daily development. Actually, one of the regular requests in TDD process assessments is to show the tests.[6] Occasionally, a team fails to show the tests. Some modern project management tools like Maven [URL:Maven] even have a separate tag for identifying test code.

Continuous Integration: The rule to check in only clean code leads to the attitude that developers try to integrate and commit as soon as possible. They do not want their tests to get broken by other programmers' changes. In practice, this means that a developer commits rather small amounts of changes. In TDD, the smallest amount of cohesive changes is the implementation of a new feature because in order to allow for a commit all tests must run and all duplications must be removed by refactoring. Thus, an analysis of the commit log should show that the integration deltas are neither too small nor too big.

Changes per refactoring: Beck [Bec03, p. 85] expects the number of changes per refactoring to follow a "fat tail" or leptocurtotic profile [Man97]. Such a profile is similar to a standard bell curve but the distribution is different because more extreme changes are predicted by this profile than would be predicted by a bell curve. Beck does not provide proof for this statement but points to a possibly fruitful area of research. This dissertation supports such a kind of research by providing a way to identify the integration versions in real world TDD projects and detect refactorings in past software changes.

Large refactorings stink: Elssamadisy and Schalliol [ES02] report a set of process "smells" for Extreme Programming ranging from organizational observations to technical issues. Relevant to this dissertation is the smell of large refactorings. Elssamadisy and Schalliol describe this smell as follows: "Basically, the premise here is that if you find yourself having to do large refactorings then you should

[6] Personal communication with Peter Roßbach, a professional TDD assessor.

have done many smaller refactorings earlier and you got lazy." There are, however, exceptions to this rule. Sometimes a big refactoring might be necessary due to a customer request or a technical issue.

This dissertation covers that "smell" as a process compliance index in Section 6.3.

Low Coupling, High Cohesion: Low coupling and high cohesion (see Section 3.2.2) is generally regarded as good object-oriented design [Boo94, p. 136]. It is, however, a goal which is hard to achieve. Writing tests first helps to reach that goal. Being able to perform each test in isolation as good as possible requires the feature under test to be decoupled from the rest of the system. Also the desire to minimize fixture code in the automated tests requires a high degree of cohesion of a feature in production code [Bec03, p. 125] [Mar03, p. 24]. Elssamadisy and Schalliol [ES02] report a similar smell of extensive setup and teardown functions in test code.

Typically, high coupling and low cohesion indicates that the refactoring step was not performed thoroughly enough.

Use Of Advanced Testing Techniques: Test-driving every single change to a system requires advanced testing techniques in many cases. The literature contains a number of advanced testing techniques such as Mock Objects, Crash Test Dummies, Self Shunt, etc. [Bec03] [LF03].

The absence of advanced testing techniques might indicate that not all changes to a system were actually driven by tests.

3.5 Summary

This chapter has presented a number of software assessment approaches.

Software process assessments are diagnostic studies that identify the current state of the development process. While such assessments can be supported by software process metrics, they tend to give a high-level qualitative picture of the development process rather than providing concrete suggestions for improving the development practices.

Software product metrics try to quantify some product properties. However, they are mainly used to predict product and resources attributes. They are not used for reflection about the development process that has produced the product.

Section 3.4.2 lists some TDD-specific measurements. They have in common that they (1) can be performed on the actually produced source code and (2) to a great extent leverage the distinction between test and production code. That way an application's source code can become subject to TDD process assessment in addition to rather qualitative retrospectives.

Some of the measurements become process metrics by measuring product attributes over time. This means that, in order to be feasible, any analysis must identify the individual integration deltas from a project's past.

The analysis of past source code changes for process assessment has not yet been subject to extensive research. Yet in TDD, the source code repository is one of the best resources to find out what has happened in the past of a project because a TDD team focuses on running software as the most important artifact that emerges from a development process.

Such analyses require tool support to focus the assessment effort. Especially the source code repository withstands a comprehensive manual inspection due to the volume of information.

This dissertation proposes a tool to support such assessments as outlined in the next chapter and detailed in the remainder of this document.

Chapter 4

Thesis

This chapter states the research hypothesis, explains the keywords, and lists the constraints of this dissertation. The subsequent chapters will detail the proposed approach and validate the research hypothesis.

4.1 Thesis Statement

> *In a Test-Driven Development context, a software process assessment can benefit from a static analysis of past source code changes. It is possible to provide automated support for this analysis.*

4.1.1 Explanation of Keywords

Test-Driven Development is a style of development that has recently become popular. Like other agile software development methods, it values "working software as a primary measure of progress" [All01]. Test-Driven Development (see Section 2.2) also demands the creation of automated tests as the primary driver for this working software and continuous refactoring [Bec03].

Software Process Assessments are diagnostic studies that can identify and rank problems in a software development project that need to be cured. They are normally carried out by on-site independent assessors that gather qualitative information about the practices, methods, tool suites, and organizational structures used for software development (see Section 3.1).

Past Source Code Changes are software changes to an application that have happened in the past of a project. They are stored in a source code repository. An integration delta consists of a number of source code changes (in test and production code),

bundling a cohesive piece of functionality. Each integration delta is the result of a basic development cycle (see Section 2.2.1). An integration version is the sum of all integration deltas that had been integrated into the application over time.

Static Analysis calculates some software product metrics on the integration deltas (see Chapter 5). This static analysis does not require an executable application for the individual integration versions. The calculated metrics are composed to process compliance indices (see Chapter 6). The process compliance indices help find process violations and focus the assessment effort during process assessment.

Automated Support is provided by the tool TddMentor (see Chapter 7) that was developed as part of this research. TddMentor is the vehicle with which a software process assessor would perform the outlined analyses. In its prototypical state, TddMentor helped to validate the research hypothesis using the analysis of a number of case studies (see Chapter 8).

4.2 Constraints

This section discusses constraints on the problem and the proposed solution.

4.2.1 Constraints on the Problem

Lack of formalization. There are numerous publications about TDD, but there is currently no rigorously formal description of its techniques in detail, or their interaction and impact on the created source code. Jackowski [Jac03] proposed a process description meta-model; however, it does not allow the required level of detail. Recent efforts in the area of software development patterns [GMR+02] [Cop97] [HG04] are promising. The literature also contains approaches to describe a software development process as interacting agents [PB02], finite state machines [BFL+95], or in terms of differential equations [CDM02].

A process assessment supported by analysis of past source code changes will always require expert judgement. A completely automated assessment is not possible. The calibration and interpretation steps are intrinsically manual steps. This dissertation proposes an approach that should help process assessors focus their analytical effort on the integration deltas that are potentially process-violating.

Experts' dispute. Elements of TDD have been well-known for some time now (e.g. unit tests, principles of good design, etc.). However, the comprehensive description of TDD as a development style is relatively young. The expert debate concerning all the

details of TDD has not yet come to an end. For example, there is still no generally accepted guideline for when tests can be omitted. Beck [Bec03, p. 194] discusses one guideline that demands to "write tests until fear is transformed into boredom". In the discussion he becomes more specific about the issue but still leaves some questions open. The QDox case study discusses one such incident (see Section 8.2.2).

Programming tasks that are hard to test. TDD generally assumes that new features have to be driven by tests into the application under development. Then the tests can demonstrate the availability of the new feature. This might be prohibitively expensive. Examples are security software and concurrency, where such a demonstration can be very hard to achieve within reasonable work effort. Some of the most important algorithms in this dissertation rely on the fact that new features are actually test-driven. This might restrict the applicability of the proposed approach or at least might impact on the accuracy of the reported results.

4.2.2 Constraints on the Solution

Concentration on Test-Driven Development. This research explicitly concentrates on TDD. Other agile software development methods might benefit from similar analytical approaches as those described here. TDD concentrates specifically on tests and on how they drive new features into the system. This is very accessible for the proposed analytical approach. Other development methods have a much broader spectrum of activities that might not be as accessible.

Statistical bandwidth. This research does not carry out a broad statistical evaluation of a large number of TDD projects. It concentrates on a few case studies to show the validity of the research hypothesis. A broader evaluation might add to the knowledge about TDD and other agile methods in general. TddMentor would enable future projects to attempt such an undertaking.

Identification of integration versions. The analysis requires the identification of the individual integration versions. The explicit declaration of such integration versions, by the developers, is the exception rather than the rule. Some authors list examples of stored change management data [EGK+01] [MEGK99]. I did not find any publicly accessible TDD project that had stored change management information in sufficiently fine detail required for this research. Therefore, TddMentor reengineers this data from the information in the source code repository (see Section 5.1). The assumptions of the reengineering algorithm for finding the integration versions are met by the documented case studies and by a few more projects that have been screened during this research. However, there is no guarantee that these assumptions apply in general.

Concentration on source code. The proposed solution concentrates on Java source code and neglects all other produced artifacts (except from the version information of the Java source files). For example, it ignores specification files from which source code could be generated. It also ignores build information which would control the processing of such specification files. These omissions lead to the introduction of new concepts such as span claims (see Section 6.2). The QDox case study discusses one such incident (see Section 8.2.2).

Static program analysis of source code only. TddMentor performs only static program analysis. It does not have the capability to build and run the individual integration versions. This prohibits assessment tools that hook into the program execution, such as some test coverage calculation tools, and therefore it restricts the capability to calculate the process compliance indices in terms of a theoretical respect. Also, many analysis tools work on compiled binaries that TddMentor can not produce. This restricts the analytical capabilities for the calculation of process compliance indices in a practical respect.

Detection of safe software changes. The proposed algorithm for calculating test coverage requires the detection of refactorings and other safe software changes in source code (see Section 5.4). A detection algorithm had to be developed as part of this research. It works well for the evaluated case studies but is not tested on a broader range of safe software changes. The proposed algorithm can, however, facilitate further research on the topic of detecting refactorings.

Chapter 5

Detection Techniques

This chapter describes a number of detection techniques that help find and understand changes in source code on various levels of abstraction. This understanding of source code changes is required for the calculation of the PCIs in Chapter 6.

At the most basic level, individual integration deltas are reconstructed from the source repository (see Section 5.1). The PCI calculations require the distinction between test and production code (see Section 5.2). PCIs are calculated for added and changed methods; they are not directly managed by the source repository (see Section 5.3). Safe software changes, such as refactorings, have to be detected in past software changes (see Section 5.4).

5.1 Reconstructing Integration Deltas

The most basic level is to reconstruct the individual integration deltas and versions (see Section 2.3.3) of a program. An integration delta is the sum of the increments that are added together to the system in order to implement a new feature. An integration version is the sum of all integration deltas from the beginning of the project. All the projects in the scope of this research use CVS as their version management system. CVS stores every individual revision of each file. Tags typically combine specified file revisions of a baseline. Individual integration versions or deltas, however, are not tagged. Given the agility of the development method, a record of integration deltas is very unlikely to exist (for example, in Extreme Programming, task cards are typically physically destroyed after their implementation).

This dissertation assumes that all commits of one integration delta

- use the same commit comment and
- are commited in close temporal proximity.

This way it is possible to reconstruct each individual integration delta from the revision log of the underlying CVS. This assumption is supported by the documented case studies (see Chapter 8).

Another proof for this assumption is provided by the version management system Subversion [URL:SVC]. It aims at being a "a compelling replacement for CVS". Commits of a number of files are atomic, and revision numbers are per commit, not per file. Subversion contains a tool that migrates CVS source code repositories to Subversion. This migration needs to identify those CVS file revisions that were committed together — operationalizing exactly the assumptions from above. Also, some TDD developers use Subversion in combination with an XP project planning tool and link user stories to the Subversion revision that contains their implementation.[1]

The proposed approach for reconstructing the integration deltas from the revision log was inspired by StatCvs [URL:StatCvs]. StatCvs analyzes a given revision log and constructs, what it calls, a commit log. Yet, this basic approach needed to be extended in several ways.

In some cases, it is necessary to deal with several CVS modules at one time. An application could consist of several Eclipse plug-ins (where each plug-in resides in a separate CVS module). Also, a separate test plug-in resides in its own CVS module (see Section 5.2). Hence, the first extension over StatCvs is to deal with multiple modules at the same time and correlate the entries of multiple revision logs. The criteria for analyzing a single module apply to this case as well. A different approach might be the use of CVS aliases [Ced93, p. 129]. A CVS alias allows the combination of several CVS modules to one combined module. The only remaining issue then might be the identification of the test code (see Section 5.2).

The analysis also needs to access the actual source code of the individual integration versions — it is not sufficient to use the names and revisions of the individual files only. More specifically, it needs to have access to multiple integration versions because it analyzes source code changes over time. In a way, this means running CVS backwards stepwise and storing each of the steps at a specific location (see Section 5.1.1).

A program not only consists of source code but also of artifacts of other types. Examples are images, libraries, grammar specifications for a compiler compiler, etc. TddMentor concentrates solely on Java source code. It has to filter out the undesired artifacts. However, the build process of the application under inspection might apply some code generation techniques to create necessary source files. It does not have the means to generate those source files because it does not run the program's build process. This policy leads to problems in the construction of the call graph (see Section 6.2).

[1] Personal communication with Jens Uwe Pipka, a professional TDD developer.

5.1.1 Fetching from Source Code Repository

TddMentor contains a component called Integration Reducer which is responsible for fetching the integration versions from a source code repository. It saves network bandwidth by reducing the accesses to CVS to a minimum, as explained below.

The Integration Reducer computes the commit log from the CVS revision log. Each commit log entry contains the file names that are different between the two integration versions — which is basically the integration delta. To be able to fetch the whole integration version, the Integration Reducer needs to know about all files and their revisions — not only the changed ones.

The commit time stamp identifies each integration version as stated in its integration delta. The Integration Reducer computes a data structure that contains all file names and their revisions for a specified integration version. It starts with an empty set of files. It then applies every integration delta up to the specified integration version. At this stage, the integration deltas are only applied to the internal data structure — no files are fetched from the source repository yet. The Integration Reducer scrolls forward in the project history, up to the so called analysis head.

Definition 5.1

Analysis head: The most recent integration version in the analysis scope. It is declared by the project-specific analysis descriptor.

This analysis head can be checked out from CVS using its standard API. The Integration Reducer creates a separate Eclipse project for every integration version (called integration project). If the application under inspection consists of a number of modules, then every integration version will consist of the same number of integration projects.

Each integration project is named uniquely by appending the commit time stamp to the name of the application under inspection.

Now, the integration projects that precede the analysis head must be created. The difference between two integration versions is located in a few files, as stated by the integration delta. To save network bandwith, the Integration Reducer copies the current integration project's contents and only fetches the changed files from CVS. It scrolls backwards in the project's history. Files that are added in the integration delta have to be removed. Files that are deleted in the integration delta have to be added from CVS. Changed files have to be reduced to their previous revision. These activities give this TddMentor component the name Integration Reducer.

The motivation for the scroll forward — scroll backwards approach is that, typically, a process assessor is most interested in the analysis head and its closest predecessors. They are the most current integration versions and are therefore the most accurate artifacts of the current project reality. It is always possible to dig deeper into the project history when needed.

5.2 Differentiate Between Production and Test Code

Test code has a very important role in test-driven development. It is not part of the production code[2], but it drives modifications of the production code (see Section 2.2.2). Thus, the capability to differentiate between production and test code is at the core of the proposed approach. All the systems within the scope of this dissertation use JUnit [URL:JUnit] for their regression unit test suite. JUnit demands the tests to be written in Java and tests are encapsulated in individual methods [GB99].

TddMentor starts building the call hierarchy graph from the modified test methods (see Section 6.2). It helps projects that are already running and therefore must be able to deal with a variety of test code organization methods. The remainder of this section discusses the different methods used to organize test code and its identification.

In the most basic and unstructured case, the test code is co-located with the production code. With a bit more structure, the test code is encapsulated in separate Java packages, closely related to the production code. In these cases, finding the test code might rely on conventions for naming test code classes and extracting them from the common code base. A more exact approach is to find all the classes that extend the JUnit-provided test case class. However, this can still leave some test code undetected, e.g. test suite, test setup, and test resource classes that are not structurally marked as test code.

JUnit itself has its means to identify test code to enable it to run the test cases. Ordinarily, the developer has to provide a test suite that compiles together a set of test cases. Following this approach, TddMentor would need to know the starting test suite for this specific integration version and then it would mimic the JUnit mechanism to find all test cases (i.e. test methods).

A very common approach to organize the test code is to put it into a separate source directory. This facilitates the test code identification dramatically because all code in the specific directory is test code. Newer project management tools such as Maven [URL:Maven] even introduced a JUnit test directory construct into their project descriptor to organize test code more easily.

Eclipse knows yet another approach to organize test code. All test code for an Eclipse plug-in is typically located in a separate test plug-in. PDE JUnit (which is an extension to JUnit) starts a new Eclipse instance and invokes such test plug-ins. Since such a separate test plug-in requires a separate Eclipse project, test code is again easily identified. Given CVS as version control system, this test code resides in a CVS module separate from the production code module. This requires dealing with two distinct revision logs (see Section 5.1 for a discussion of this case).

Other test frameworks (such as Parasoft's Jtest or SilverMark's TestMentor) have other mechanisms to organize unit tests. The documented and scanned case studies

[2] Test code might be used for the production environment for integration tests, though.

did not use any test frameworks other than JUnit but as long as the individual tests are located in Java methods that call production code, TddMentor is able to process it.

5.3 Finding Modified Methods

Most of the analysis of TddMentor is based on individual methods of test and production code. More specifically, it is based on modifications of individual methods between two integration versions (namely the integration deltas). For example, the test coverage is defined in terms of method reachability within the call graph (see Section 6.2) from modified test methods. This section explains how to find these modified methods. In other words, it deals with how to get from modifications in the byte stream world on disc to the corresponding modifications in the static structural analysis world of the Eclipse Java model [SDF+03] [GB04]. In this model, every method is clearly identified by its signature, class and project. This means, while a method has the same signature and is located in the same class as its predecessor, it is still located in a different project and hence can be uniquely identified.

Each modified method is either changed, added, deleted, or moved according to the following definitions:

Definition 5.2

Changed Method: The method has changed, however, leaving the method signature the same.

Added Method: There was no previous method.

Deleted Method: There is no successor method.

Moved Method: The method has moved to a different class or the class has moved to a different package, leaving the method otherwise unchanged.

The algorithm described here will not find any members of the Moved category. Rather, it will find a lot of false positives for the Added and Deleted categories and it will find far too few for the Changed category. Section 5.4 deals with how to adjust this mismatch.

The integration delta tells which files were added, deleted or just changed. File changes occur far more often than additions or deletions. However, the recorded file changes refer to textual changes in the managed files. CVS has no means for recording structural file changes although newer version control systems can deal with structural changes to some extent (see Section 5.4.5).

The TddMentor component called Predecessor Finder marks all methods in added or deleted files as added or deleted methods accordingly, because for now, no further information is available. It uses the method abstraction of the Eclipse Java model.

A major drawback is that CVS records classes that are renamed or moved to a different Java package as deleted and added files. Thus, many of the marked methods should be in the moved category; however, this correction must be left to later stages in the analysis (see Section 5.4.1).

In practice, the majority of files are changed and recorded as such by the version control system. These files are eligible for calculating their structural changes at the level of changed, added or deleted methods (see definition 5.2) between two given files. Miller and Myers [MM85] describe an algorithm for exactly this kind of structural comparision of two given files. Eclipse already implements this algorithm and provides structure creators for Java.[3] For the sake of simplicity, TddMentor ignores all modifications in inner classes and non-public top level classes. This is an area for further improvement of future versions of the tool.

Another improvement would be to find, rename and move modifications on class level. In [WL01] we showed that this is possible, heuristically. The algorithm described below finds rename and move modifications on method level.

5.4 Finding Safe Software Changes

Finding refactorings in existing code seems to be almost a blind spot in current research. See Section 5.4.5 for the few existing approaches. TddMentor, however, needs to find the refactorings and some other safe software changes for the PCI calculations in Chapter 6.

This chapter explains a pragmatic approach to finding refactorings, refactoring participants and change participants. The approach has three steps:

1. Select a predecessor for a given method.
2. Find the raw differences between the two methods (called method deltas).
3. Reason about those differences in order to identify refactorings, change participants, and refactoring participants.

5.4.1 Selecting a Predecessor Method

TddMentor needs to find the refactorings in the methods of a given integration delta. A refactoring changes a given method — and sometimes creates a new method as in *Extract Method* (see Section A.1.6). Because CVS does not record changes for methods, it cannot return a method's predecessor. So the first challenge is to find a predecessor for a given method before the next step can start to calculate the exact differences between a method and its predecessor. The two steps to select a predecessor are:

[3] Whitespace modifications are ignored.

- Select the exact predecessor if possible.
- Select a predecessor heuristically.

Select Exact Predecessor

Some refactorings leave a method's signature and location[4] unchanged — like *Rename Parameter* (see Section A.1.10). In the preceding integration version, such a method is found easily, at exactly the same location with exactly the same signature. That method is called the exact predecessor — assuming the developer did not change the intention of that method while leaving its signature and location unchanged.

Having found a predecessor, the next algorithm step can begin to calculate the differences between a method and its predecessor (see Section 5.4.2). If such an exact predecessor is not found, TddMentor applies the heuristic below.

Select Predecessor Heuristically

Some refactorings change a method's signature — like *Rename Method* (see Section A.1.9) — or its location — like *Move Method* (see Section A.1.6). Predecessors for such methods are not easily found. A simple heuristic helps to find a good-enough predecessor, given the following observations.

- Whenever a method signature changes between two integration versions, the algorithm in Section 5.3 reports the preceding method as deleted and the successive method as added.
- Whenever a method moves to a different class, the same algorithm reports the preceding method as deleted and the successive method as added.
- Whenever a classes moves to a different package, CVS reports the preceding class as deleted and the successive class as added. The above algorithm reports methods in deleted classes as deleted and methods in added classes as added.

Thus, all methods that change signature or location are added and their predecessors are deleted. This is the search space for the heuristic to find a good-enough predecessor.

The heuristic iterates through this search space and tries to match two methods by calculating the raw differences (see Section 5.4.2) and then the refactorings (see Section 5.4.3). If it finds a match (i.e. the difference between two methods is identified as refactoring), the heuristic stops and reports the matched method as predecessor. If it does not find a match, no predecessor is reported.

[4]The location of a method is its declaring class and the location of that class. The location of a class is its package. Therefore moving a method to a different class or moving its declaring class to a different package would change its location.

This heuristic has its limitations, though. For example, if a method is duplicated from one integration version to the next, only one of these methods is found. Also, the heuristic does not respect local proximity, if it has several choices — a method in a different class of the same package would be more likely to be the predecessor than a method in a package "far away".

5.4.2 Finding Method Deltas

The changes of interest for finding refactorings are structural differences. An abstract syntax tree (AST) models the structure of a programming language construct like a method. To find the structural differences between two methods, TddMentor has to find the differences of the ASTs of the two methods at hand. Those differences are called method deltas.

Definition 5.3

Method delta: A structural difference between two given ASTs. It is an instance of a number of known method delta types.

The Eclipse Java model has built-in support for Java ASTs. Moreover its AST matcher can compute whether two given AST subtrees are structurally isomorphic or not. It does this by visiting the two trees from their roots.[5] Type-specific match methods (like `boolean match(Block node, Object other)`) compare the nodes of the two ASTs in each step. Every two pair of nodes has to match in order for the whole subtree to match.

This AST matcher is an excellent basis for computing the method deltas. TDD demands to go in small steps; doing one piece of change at a time. Method deltas cannot change the structure dramatically. Thus the ASTs of interest are similar except for incremental differences.

TddMentor provides an AST Differencer component, which is a specialized AST matcher. It reports two ASTs to be structurally isomorphic if they have well-known differences only and returns a list of these method deltas. The delta mapping algorithm below only detects method delta types for the match methods which have been implemented already. See later for some match method implementations in TddMentor.

The Delta Mapping Algorithm

The AST Differencer visits the AST subtree just like an AST matcher. In every match method, it tries to match the elements of the AST node at hand (the elements are AST subtrees by themselves). Whenever it finds a difference, it tries to match it to one of the known method delta types.

[5] The AST matcher is an implementation of the Visitor design pattern as described in [GHJV95]. However it visits two trees at the same time, not only one tree.

This match might involve a direct comparison of attributes of the given nodes. In many cases, it requires to walk deeper into the subtrees and collect the method deltas found on the way. If such a subtree walk does not result in a match, the method deltas found up to that point are invalid. They must be discarded. The next match attempts provide another chance until the match method has attempted all possible method delta types — in which case the match method returns `false`. The following example hopefully clarifies this algorithm.

Listing 5.1 (match method for a Java block)

```
public boolean match(Block node, Object other) {
    int saveResultSize = fDeltas.size();
    boolean superMatch = super.match(node, other);
    if (superMatch) return true;
    restoreSavedResult(saveResultSize);
    if (checkForDelegatingCtorDelta(node, other)) return true;
    restoreSavedResult(saveResultSize);
    if (checkForExtractMethodDelta(node, other)) return true;
    restoreSavedResult(saveResultSize);
    IDelta delta = new OtherBodyDelta(fSuccMethod);
    fDeltas.add(delta);
    return true; // Typically returns false. See text!
}
```

Listing 5.1 shows the match method for a Java `Block`. Before any match attempt starts, `saveResultSize` points to the current end of the results list. `super.match(node, other)` invokes the regular match method of the AST matcher. Be aware that this might still indirectly invoke some other AST Differencer-specific match methods. If the regular match method did not find a match, all results collected so far are invalid. `restoreSavedResult(saveResultSize)` discards all the invalid results.

This specific match method always returns `true`, because it reports an *Other Body* method delta, if all other match attempts fail. This method delta type only makes sense for the calculation of change participants, as detailed in Section 5.4.4.

Note the observations about how to implement a match method:

- No match method exists in isolation.
- The order of the attempted delta matches is important.
- An introduction of a new method delta type might impact the detection mechanism for other method delta types.
- Clean up after a match attempt has failed.

- Sometimes a match method higher in the subtree needs to prepare some information for a match method deeper in the subtree. For an example see *Change Return Type* delta below.

In the following section, details of how to detect some specific method delta types are shown. Only a selection of method deltas types that help explain the concepts of this chapter are shown. Appendix A contains a comprehensive list of all method deltas that are detected by TddMentor.

Rename Parameter Delta

Parameter names are found easily in the `MethodDeclaration` AST node. The match method finds the corresponding parameters via their position in the parameter list. This way, it is easy to detect if a parameter name has changed. The match method records these changes as *Rename Parameter* deltas in the results list.

If parameter names change, then references to those parameters change as well. This means the match method for `SimpleName` nodes will run into name changes. A normal AST matcher would report such a name change as non-isomorphic. The AST Differencer however will not report it as mismatch, if it finds a corresponding *Rename Parameter* delta in the list of results that have been found so far.

Extract Method Delta

An *Extract Method* refactoring (see Section A.1.4) replaces a subsequence of `Statement` nodes with a `MethodInvocation` node and moves the replaced `Statements` to a new method.

The match method looks for a subsequence in the predecessor `Block`'s `Statements` that corresponds to the `MethodInvocation` in the successor `Block`. The `Statements` before and after the subsequence have to match the `Statements` before and after the `MethodInvocation`.

The `MethodInvocation` refers to the `MethodDeclaration` of the newly created method. Its `Statements` have to match the identified subsequence in the predecessor `Block`. Then the predecessor and the successor `Block` are reported as structurally isomorphic modulo the *Extract Method* delta (see Section A.4.10).

Change Return Type Delta

The return type of a method is found in the `MethodDeclaration`. Only `SimpleTypes` are allowed as return types.[6] Thus a comparison of the identifiers of the two `SimpleType` nodes detects a change.

[6] A `SimpleType` is a `Type` node for a named class or interface type.

However, to allow for a good interplay with other detection algorithms, the detection is left to the match method for `SimpleTypes`. The `MethodDeclaration` match method simply saves the return type `SimpleType` node. Then its match method knows when to check for a *Change Return Type* delta.

Other Body Delta

The AST Differencer returns *Other Body* delta (see Section A.4.12), if all other match attempts for a `Block` fail; as seen in listing 5.1. The purpose of this method delta is to enable the AST Differencer to return a list of method deltas if it can match the two method signatures but not the two method bodies. As seen in Section 5.4.3, this method delta will never be reported as refactoring. However, to calculate change participants (see Section 5.4.4) the detection mechanism needs a list of method deltas in the signature, even if the two method bodies do not match. *Other Body* delta somehow resembles the *Null Object* design pattern [MRB97].

5.4.3 Identifying Refactorings

The TddMentor component Refactoring Finder gets the method deltas from the AST Differencer. It analyses the method deltas to identify refactorings. As the raw method deltas are already at hand, it is an easy procedure; sometimes as easy as directly accepting a method delta as refactoring such as *Rename Parameter*. In other cases it has to search the type hierarchy of affected types as for *Reconcile Differences*, or do some other checks.

The Refactoring Finder gets a list of method deltas. It only accepts a method as refactored if it can resolve the whole list of method deltas to refactorings. Otherwise, it will not accept the whole method. There are two reasons for rejecting a method delta as refactoring. Firstly, some precondition has not been met (e.g. the new type cannot be found in the type hierarchy of the old type). Secondly, some method deltas will never resolve to refactorings. For example, the *Other Body* will always be rejected as refactoring.

The details of when to accept some specific method deltas as refactoring have been outlined below.

Rename Parameter Refactoring

Detecting the *Rename Parameter* refactoring (see Section A.1.10) is easy. The Refactoring Finder simply accepts every *Rename Parameter* delta as refactoring. All reference checks were already performed by the AST Differencer. There is nothing else to reason about.

Extract Method Refactoring

The Refactoring Finder accepts an *Extract Method* delta (see Section A.4.10) as *Extract Method* refactoring (see Section A.1.4), if the called method was added in this integration version. The AST Differencer already has ensured that the extracted `Statements` have been moved to the newly created method. The Predecessor Finder reports new methods as added.

Reconcile Differences Refactoring

The *Reconcile Differences* refactoring (see Section A.1.7) tries to unify pieces of code that look similar [Bec03, p. 181]. One type of this refactoring is to replace a type with its supertype if the accessed functionality has moved to the supertype. For some methods this means "return an object of a supertype of the type that it had declared in its predecessor". The following example is taken from [Bec03, p. 35].

Listing 5.2 (Before refactoring: Method returns Franc)

```
public Franc times(int multiplier) {
   return new Franc(amount * multiplier);
}
```

As a small step in the process of unifying the types `Franc` and `Dollar`, the `times` method returns the supertype `Money` instead of `Franc`.

Listing 5.3 (After refactoring: Method returns supertype Money)

```
public Money times(int multiplier) {
   return new Franc(amount * multiplier);
}
```

The Refactoring Finder receives such changes as *Change Return Type* delta. It has to check whether the new type is a supertype of the old one. In the example, `Money` is an ancestor of `Franc`. This means the method delta is reported as refactoring.

5.4.4 Finding Change and Refactoring Participants

Change and refactoring participants (see Section 2.3.4) are method changes that also need to be detected for the correct calculation of the test coverage PCI (see Section 6.2). Again, the AST Differencer has to detect a raw method delta before the Refactoring Finder can decide on the type of change.

For an example see listings 2.5 and 2.6. There the AST Differencer detects a *Change Argument* delta. To see if this method delta was induced by a change in the called method, TddMentor has to check the list of method deltas for `setLocation`. If it finds

a *Change Parameter Type* delta for this parameter position, then it reports `getCompile-Classpath` as change participant. This check is the raison d'être for the *Other Body* delta. It allows checking for deltas in the method signature even if the two method bodies do not match. The TddMentor component Coverage Finder will report such a change participant as covered if the method that induced the change is in the test span (see definition 6.3).

A refactoring participant is some kind of change participant. It differs in the fact that the inducing method itself was undergoing a refactoring — not only a change. Thus the inducing method does not have to be spanned by tests. The method at hand is still within the test coverage.

5.4.5 Discussion

The described approach for finding refactorings has some shortcomings, but they are more than compensated for by its benefits.

Firstly, the proposed approach is not as stable against simple changes as one might hope for. Malpohl *et al.* [MHT00] describe an algorithm for detecting renamed identifiers in existing code. Their intent is to support the merge of different branches in a version control system. The algorithm is stable against some simple statement reorderings. Yet they only detect renamed identifiers. The approach in this dissertation clearly needed to detect more kinds of refactorings.

Secondly, the proposed approach may not scale well for big changes between two compared versions. Demeyer *et al.* [DDN00] describe an algorithm that finds a small number of refactorings via change metrics. They compare major releases of three OO systems as case studies. However, most of their refactorings are too big in the sense that such refactorings would not be possible without a test in TDD. Also, they mainly find class reorganizing refactorings such as *Extract Superclass* (see Section A.1.5) and *Collapse Hierarchy* (see Section A.1.2). The PCI calculation in Chapter 6 on the other hand needs to find a large number of small refactorings — such as *Move Method* (see Section A.1.6) and *Rename Parameter* (see Section A.1.10).

Steiger [Ste01] takes the idea of Demeyer *et al.* [DDN00] a bit further by providing a flexible query engine. This query engine helps to detect more refactorings. However, the author still has the problem of identifying predecessors. He proposed to calculate a fingerprint of software entities to identify predecessors but did not carry out this idea.

Antoniol *et al.* [ACL99] compare two versions of classes in an intermediary abstract language. They present metrics to express the distance between the two versions of the same class. Such metrics might help to find refactorings similar to the approach described above.

A completely different approach records refactorings as meta information in the source code repository. Existing version control systems traditionally support operations such as add, delete, change on a file or method level. They are not designed for

huge amounts of structural changes such as refactorings. A new idea is to treat refactorings as basic operations. This extension facilitates more advanced operations such as merging different program branches.[7] Collard [Col03] describes several approaches for attaching meta information to source code. For example, he plans to attach difference and refactoring information to changed code. However, such meta information is currently not available for real world projects.

The proposed approach finds a large number of refactorings on a level of detail that fits into the PCI calculations. It is stable against some identifier renamings and some method movements, and it scales well for TDD code, which is the scope of this research. The case studies support the adequacy of this approch.

I presume that the topic of finding refactorings will attract more research interest in the near future. The software engineering research community starts to understand the importance of refactorings [MDB+03]. When design patterns were first proposed, they were seen as tools to help design computer programs [GHJV95]. Nowadays they are reengineered from existing code to better understand program design [Bro96] [SS03]. It is my conviction that detection of refactorings and other safe software changes will see a similar career to enable better understanding of source code evolution.

5.5 Summary

This chapter describes techniques for detecting software entities on several levels of abstraction. The detection of those entities is required for the PCI calculations in the next chapter.

Appendix A lists and describes the method deltas, refactorings, refactorings participants and change participants that are found by TddMentor, which prototypically implements the detection techniques of this chapter.

[7] For exposées that envision this research direction see [URL:CmEclipse] and [URL:Freese].

Chapter 6

Process Compliance Indices

This chapter defines a number of process compliance indices (PCIs).[1] PCIs quantify symptoms of TDD process violations based on an analysis of past source code changes. Chapter 5 describes detection techniques for a number of software change entities that are used for the PCI calculations. PCIs are calculated for a sequence of integration deltas (see definition 2.1). They help focus the manual assessment effort in order to find integration deltas that document a process violation.

These integration deltas that were found can be subject to a review with the development team members to discuss the reasons and find ways to prevent such process violations in the future. A PCI can only be an indicator for a process violation — it cannot serve as proof. The process assessor has to judge based on his or her TDD process expertise and when in doubt discuss a software change with its developers. Or, as Armour [Arm04] has put it, "Simply collecting one metric rarely gives us the answer — on the contrary, it usually gives us the question."

Section 6.1 below deduces the research assumptions and metrics criteria based on the GQM paradigm. The test coverage PCI (see Section 6.2) is used in Chapter 8 to validate the research hypothesis. Section 6.3 defines another possible PCI. More PCIs could be derived from TDD-specific measurements easily (see Section 3.4.2).

6.1 Goal-Question-Metric Deduction

This chapter defines several process compliance indices that are based on software product change metrics. The decision which metrics to apply is made using the Goal-Question-Metric paradigm (see Section 3.2.3). The goal and the question are defined first in Table 6.1.

[1] The notion of PCIs is inspired by Code Decay Indices of Eick *et al.* [EGK+01], who also analyze past source code changes to estimate maintenance effort.

Goal	Identify those integration deltas that suggest a TDD process compliance violation in order to focus the manual assessment effort.
Question	When analyzing the software changes of one TDD integration delta, what characteristics can be used as symptoms for a TDD process violation?

Table 6.1: Goal and question for the definition of the process compliance indices (PCI). Each PCI is a specific answer to the question that was deduced from the goal.

The question "How to find symptoms of process violation in software changes" leads to the following research assumptions:

- *Source code reflects process violations.* TDD is a very source code centered style of development (see Section 2.2.1). All development activities are directly reflected in the sources. Process violations also directly impact the source code (e.g. not having a test for some production code change would be a process violation). Some process violations are detectable — like a missing test. Clearly, not all process violations are detectable (like creating the right design closure for the problem domain).

- *Integration deltas can be identified and integration version can be recreated.* Since TDD developers typically do not tag individual integration deltas, this research assumes that all changes of one integration delta are submitted to the source code repository as a whole. TddMentor allows to reconstruct the integration deltas and hence the integration versions based on this assumption (see Section 5.1).

- *Test code is distinguishable from production code.* Test and production code have different but very important roles. This dissertation describes a number of possibilities to distinguish between the two types of source code (see Section 5.2). It is becoming standard development practice to mark test code as such.

A PCI is a heuristic for indicating process violations. It is the combination of one or more software product change metrics. Each PCI is one possible answer to the GQM question from Table 6.1. The metrics used in a PCI have to satisfy the following criteria:

- The metrics must be derivable from the source code and/or the version control information.

- The metrics should be cheap to collect because the process compliance indices should be calculated for several integration deltas as part of two successive integration versions. Each of which can contain a lot of source code.

- The metrics must be computable via static source code analysis. Every integration version should have been executed by running the automated tests. But it is very hard or might even be impossible to recreate the test execution context for every single integration version.

Each PCI specifies a recipe that explains how to interpret whether the result indicates a process violation or not. In other words, the interpretation gives a positive or negative response to the question, whether a given integration delta contains a process violation or not. This interpretation can also lead to false positives and false negatives. A *false positive* for a PCI is a case where the PCI indicates a process violation but should not have reported one. A *false negative* for a PCI is a case where the PCI does not indicate a process violation where it should have reported one; thus where TddMentor would fail to report a process violation.

A PCI fulfils the first three requirements of Weinberg's observational model as introduced in Section 3.2.1. Weinberg [Wei93, p. 30] also suggests to define a set of top tier metrics for a quick overview of a development process and a set of second tier metrics for more detailed root cause analysis. The PCIs can give an overview of what has happened in the past of a development project. Finding the root cause requires a detailed inspection of the identified integration deltas and its discussion with the developer.

Each section below defines one PCI, containing a detailed description, a recipe for its interpretation, and a discussion of false positives and false negatives.

6.2 Test Coverage PCI

As in Figure 2.1, an integration delta consists of both test code and production code. As outlined in Section 3.4.2, all the software changes of an integration delta have to be covered by the tests of the same integration delta. This PCI calculates the test coverage for individual integration deltas. The PCI calculation is modelled along the basic TDD development cycle, as outlined in Table 6.2.

Step	**Development**	**Analysis**
Red	Write a test	Start the call graph at the modified test methods
Green	Implement a new feature	Mark all spanned methods as covered
Refactor	Clean the code structure	Mark all safe method changes as covered

Table 6.2: The Test Coverage PCI calculation is modeled along the TDD mantra Red/Green/Refactor.

The test coverage is calculated in terms of a call graph that starts at the test methods of the integration delta. The algorithm marks all the production methods spanned by this call graph as covered. This call graph might need to be extended artificially as detailed below. Safe software changes (see Section 2.3.4) do not have to be tested explicitly and therefore are also marked in the last step as covered.

The literature knows a lot of different test coverage types and calculation algorithms (see Section 3.2.2). I chose the approach presented here, because it models the way a developer test-drives new functionalities. The case studies (see Chapter 8) provide empirical evidence that this simple algorithm works well.

Details of this test coverage calculation are described below.

6.2.1 Details

The test coverage has to be calculated for the changes in each integration delta within the analysis time frame. This notion differs from the traditional view on test coverage (see Section 3.2.2) where it is typically calculated for the whole system.

Definition 6.1

Test coverage denotes to which degree methods of an integration delta are covered by tests in the TDD sense.

$$\textbf{Test coverage rate} = \frac{\text{\# covered production methods in integration delta}}{\text{\# all production methods in integration delta}}$$

TddMentor applies a very simple test coverage algorithm based on the method call graph. Informally, a method call graph represents the calls between methods in a given program. The quality of the call graphs computed by available static call graph extractors varies widely [MNGL98]. TddMentor uses the static call graph extractor found in Eclipse.[2]

Definition 6.2

Method call graph: A directed graph with nodes representing individual methods and edges from method `m1` to `m2` if `m1` calls `m2`.

The call graph extractor simply looks for all methods that are called somewhere from within the method at hand. For the test coverage calculations, it starts at the test methods of an integration delta. This includes methods that are not specifically marked by JUnit as test cases (like `setup()`). The algorithm marks all methods in the integration delta that it reaches via the call graph as spanned.

[2] It is found in the package `org.eclipse.jdt.internal.corext.callhierarchy` in Elipse 3.0.

Definition 6.3

Test span: The set of all methods reached from a test method via the call graph.

Spanned method: A method contained in the test span.

Test coverage is basically this test span of the call graph that starts in the test methods of the integration delta.[3] As described below, this algorithm needs some extensions to provide realistic results. Chapter 8 documents some case studies where these extensions were needed.

Implementors and Subclasses

In TDD code it is common to modify existing abstractions such as interfaces and superclasses. Test-driving such modifications means to write a test case or to modify an existing test case that executes the modified methods. Interfaces, however, have implementors, and superclasses have subclasses that implement or override methods. They have to change if their ancestors change. Because the tests drive against the ancestor types, the simple algorithm would leave modified descendants as not spanned.

Thus, the algorithm marks modified methods as spanned if they changed because of a signature change of their ancestor method and the ancestor method is spanned.

Span Claims

TddMentor only analyzes Java source files (see Section 5.1). It leaves out other artifacts (like grammar specifications) that a build process might use to generate Java source code. TddMentor is missing those would-be-generated Java source files and thus it is missing some links in the call graph if the generated Java source code calls manually created code.

For an example see listing 6.1 from the QDox case study (see Section 8.2).

Listing 6.1 (Calling generated code)

```
public class JavaClass {
   ...
   public JavaSource addSource(Reader reader) {
     ModelBuilder builder =
       new ModelBuilder(classLibrary, docletTagFactory);
     Lexer lexer = new JFlexLexer(reader);
     Parser parser = new Parser(lexer, builder);
     parser.parse();
     ...
```

[3] This algorithm does not take into account more advanced test coverage techniques such as branch reachability and the like.

```
    }
}
```

The classes `JFlexLexer` and `Parser` would be generated from some specification files. The `parser` object would call the `addPackage(p)` method in the `builder` object as seen in Figure 6.1.

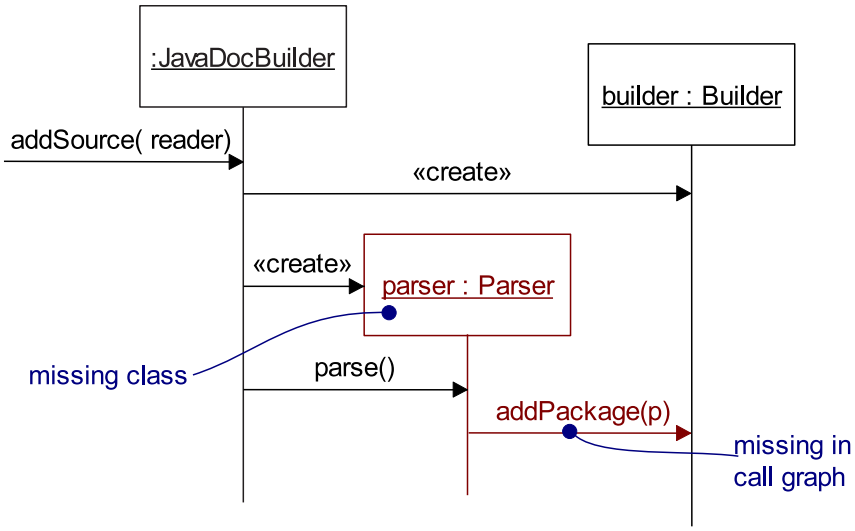

Figure 6.1: *Rationale for span claims.* The class `Parser` cannot be generated during the static analysis, thus it breaks the call graph. Therefore the call to `Builder.addPackage(p)` has to be declared manually be a span claim.

Span claims make up for those missing links.

Definition 6.4

Span claim: A user-provided extension of the call graph. A span claim consists of

- the method signature of a method that the call graph reaches from a test method and
- a set of method signatures of the methods that would be called by the generated code.

The tool user declares this information in a project-specific analysis descriptor. A simple call graph detection algorithm stops when it does not find a method that is called from within an existing call graph. In this case, the call graph has to continue at the target methods declared by the span claim.

The QDox case study documents the span claim for the example above (see listing 8.3).

Safe Software Changes

In TDD, every production code modification should be test-driven, except for safe software changes. A refactoring is such a safe software change (see Section 2.3.4). Assuming the code before the refactoring was developed in a test-driven way, there should be tests that cover this piece of code. The semantic-preserving code modifications neither change the coverage nor invalidate the the old test, so there will not be any new tests.

Modified code, due to a refactoring, is not spanned by tests in the current integration version. It is, however, spanned by tests in an earlier integration version and as such regarded as covered for the current integration version.

Refactoring participants, change participants, and accessor method are similar kinds of safe software changes (see Section 2.3.4). TddMentor can detect some of those safe software changes automatically (see Section 5.4). All automatically detected safe changes are marked as covered.

Coverage Claims

TddMentor cannot detect all kinds of safe software changes, e.g. some accessors and technical methods (see Section 2.3.4).

For an automatic analysis this means that TddMentor will report some methods as uncovered because it does not find a test, even though the developer of this method did not define a test on purpose. Reporting such methods would disrupt an analysis report. To prevent them from reappearing, a TddMentor user can declare coverage claims.

Definition 6.5

Coverage claim: A user-provided extension of the test coverage. A coverage claim consists of

- a method signature of the method that should be added to the test coverage.

The methods declared by a coverage claim are not reported as uncovered in any case. Coverage claims can be applicable globally or specifically to an individual integration version.

The Money Example case study shows an example of a coverage claim for the `toString()` method (see listing 8.1).

Test Span Fix Point

A simple call graph algorithm does not compute the test span adequately, as described above. The above extensions add to a test span that has been calculated so far. Data from TDD projects show that the call graph algorithm has to continue after these additions. Then again, the algorithm extensions need to determine whether they can add even

more to the test span. This whole procedure needs to be performed until a fix point is reached, where the call graph algorithm and its extensions find no further addition to the test coverage.

6.2.2 Recipe for Interpretation

The source of interpretation is a time series diagram of the test coverage rates for the individual integration deltas. This diagram also contains the number of detected safe software changes, as exemplified in Chapter 8.

The analysis of fully TDD compliant projects should result in a test coverage rate of 1.0 for every integration delta (e.g. see the Money Example in Section 8.1). The analysis of a project that does not have any tests at all results in a test coverage rate of 0.0 for every integration delta. These two cases mark the extreme ends of the spectrum for real world projects.

This PCI has two different recipes for interpretation according to the two ends of the spectrum. One recipe is for cases that are close to an ideal TDD project (see QDox case study is Section 8.2 for an example). For such projects, an interpretation of the numbers can help find individual process violations. A review of such process violations supports a TDD process optimization on an advanced level. The other recipe is for cases that have hardly any TDD process elements (see Ant case study in Section 8.3 for an example). For such projects, an interpretation can help the team members recognize first steps into the right direction and encourage to build on initial success.

Nearly Ideal TDD Process

1. Find the integration deltas that have a poor test coverage rate and order those candidates accordingly.

2. Manually review those integration deltas to find the reason for the poor test coverage. Possible reasons are: no tests defined in this integration delta, refactorings and other safe changes were not detected, changes were not driven by tests, or any other process violation that reflects in the source code.

3. In the case of missing coverage claims or span claims, declare them in the analysis descriptor and rerun the analysis. Coverage claims can also make up for refactorings that were not detected by the tool.

4. Discuss the identified integration deltas with the developers to prevent misinterpretations.

Hardly any TDD Process Elements

1. Find the integration deltas that have some reasonable test coverage rate.

2. Check the number of detected safe software changes to see if this non-zero test coverage rate is the sole result of detecting safe changes. In this case, the integration delta does not provide insight into initial TDD steps.

3. Manually assess the remaining integration deltas to identify the TDD process elements.

4. Discuss the results with the developers to learn more about their comprehension of these TDD process elements.

6.2.3 False Positives and Negatives

False positives. A false positive scores a low test coverage rate while it is process-compliant. Probably the most important reason for a low test coverage rate is the refactoring detection mechanism. It is limited in the number of refactoring types that it can detect in software changes. A tool user would need to detect such cases manually. He/she can then declare a coverage claim for that integration delta to prevent the refactored method from reappearing again in the analysis report.

Occasionally, the regular build process generates source code from some specification. The presented approach does not run the build process, thus some source code might be missing. Such missing source code might break the call graph and therefore is another source for false positives. The declaration of span claims can alleviate such problems.

Another reason for false positives is the static extraction of the call graph. This extraction relies on the unit tests to invoke the production code via static method calls. If those calls are made dynamically, the test coverage calculation would fail. TddMentor does not offer a quick fix for such cases but they are rather rare.

False negatives. Some integration deltas could score a high test coverage rate but in reality they are not TDD process compliant. They would be seen as false negatives. The simplest example of such a case is an integration delta with a test coverage rate that is explained only in terms of detected safe software changes. The Ant case study (see Section 8.3) documents such integration deltas. For this reason, the results diagram always shows the number of detected safe software changes so that the process assessor gets an idea for such false negatives.

Another possibility to obtain a false negative is a wrong coverage claim declaration. The user has to exercise the tool calibration with great care.

6.3 Large Refactorings PCI

As outlined in Section 3.4.2, Elssamadisy and Schalliol [ES02] reported a process "smell" for Extreme Programming named "Large refactorings stink". They recommend to put more effort into the refactoring step during every basic development cycle if the refactoring activities become too large. They assess that, in this case, the developers had become lazy with the continuous refactoring activities. As a result, the required refactorings accumulate to a large amount of work at several discrete points in time as opposed to a continuous refactoring effort during each integration delta.

6.3.1 Details

During the refactorings development step, not only refactorings happen, but in general, all kinds of safe software changes can happen (see Section 2.3.4). Besides refactorings, there can be changes in refactoring participants or change participants.

It is possible to detect a number of safe software changes in past integration deltas (see Section 5.4.3). For the calculation of the test coverage PCI (see Section 6.2), only the safe software changes that had not been covered by tests already were of interest. The Large Refactorings PCI needs to find all safe software changes, irrespective of their coverage by a test.

This PCI simply counts the number of safe software changes for every integration delta within the analysis time frame. These numbers are depicted as time series diagram.

6.3.2 Recipe for Interpretation

1. Identify the integration deltas that have the highest numbers of safe software changes and mark them as candidates for the manual assessment. As a first guess for the control limit you might choose three times the standard deviation from the arithmetic mean.

2. Manually assess the marked integration deltas in order to see if they indicate some refactoring activities that should have been performed in an earlier integration delta. Possibly adjust the control limit.

3. Discuss the identified integration deltas with the developers to prevent misinterpretations.

6.3.3 False Positives and Negatives

False positives. A false positive integration delta scores a PCI above the control limit but does not contain any safe software changes that have been deferred from earlier integration deltas due to programmer laziness.

One reason might be that the new tests in this integration delta really require such a massive refactoring activity. Another reason might be the size of the integration delta. A big integration delta contains more safe software changes than a small one.

A more technical reason might be a wrong value for the control limit. It requires some experience to find the right adjustment. Another technical reason could be the safe software change detection mechanism itself. The detection algorithm is limited in the number of different types of safe software changes that it can detect. This shortcoming equally applies to all integration deltas. If, however, one integration delta has a much higher rate of detectable safe software changes than all other integration deltas, it is reported as a false positive.

False negatives. A false negative integration delta scores a PCI below the control limit but shows an accumulation of many deferred safe software changes.

The control limit might just be too high for the detection of a process violating integration delta.

Another technical reason is the limitation of the safe software change detection mechanism to detect a large number of safe software change types. This would result in a PCI that is too low.

6.4 Summary

This chapter introduces PCIs that indicate process violations for individual integration deltas. It lists two specific PCIs. The test coverage PCI was implemented by TddMentor to validate the research hypothesis. The TddMentor implementation and usage is discussed in the next chapter.

Chapter 7

TddMentor Architecture and Usage

TddMentor is a prototypical implementation of the detection techniques of Chapter 5 and the test coverage PCI of Chapter 6. It supports the assessment approach proposed by this dissertation. In the taxonomy of Demeyer *et al.* [DMW01], TddMentor provides change-based retrospective evolution support. It is a semi-automated tool because it still requires human assistance. In fact, it helps a process assessor focus the inspection effort during TDD process assessment.

This chapter outlines the tool architecture and briefly describes its components (see Section 7.1). It lists the configuration options and discusses the calibration process (see Section 7.2). While TddMentor is mainly used for TDD process assessment, it could be applied in other areas as well (see Section 7.3).

7.1 Architecture

TddMentor plugs into the Eclipse IDE. An integration of an analysis tool into the development environment has many advantages. Separate tools would construct separate models of the application under inspection and often need to omit details. "Omitting those details often has the effect of producing spurious error reports." [YTFB89]

Eclipse provides several types of tool integration: invocation, data integration, control integration, and GUI integration [WSP+02] [Ams01].

7.1.1 TddMentor as Eclipse Plug-in

Eclipse is a tool integration platform that has a built-in Java IDE [SDF+03] [GB04]. TddMentor uses several Eclipse components, namely the plug-in mechanism, the resource model, the Java model, and the CVS tooling as described below (see also Figure 7.1). It is integrated with Eclipse on the control and GUI levels.

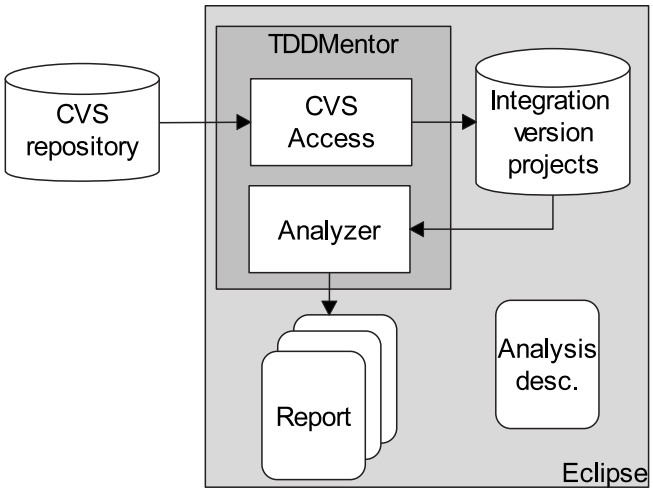

Figure 7.1: *Architecture of TddMentor.* TddMentor plugs into the Eclipse IDE. It accesses the CVS repository to reconstruct the integration deltas and versions. The analyzer produces PCI reports from those integration deltas. The analysis descriptor scopes this analysis.

The Eclipse plug-in mechanism provides an easy integration into existing tools in the platform. Extension points allow the reuse of core functionality or integration into the GUI. TddMentor is one such external plug-in.

The Eclipse resource model organizes resources such as files, directories and projects. TddMentor reads the analysis descriptor for an application under inspection. It generates PCI report files. TddMentor creates an Eclipse project for every integration version extracted from the source code repository.

The Eclipse Java model parses Java files and provides handles to program elements such as classes and methods. It provides a comprehensive library for program analysis. TddMentor does not define its own program model or model database but rather completely integrates with the model in Eclipse. It uses the static analysis capabilities to calculate the PCIs.

The Eclipse CVS tooling allows easy access to CVS source repositories from the Eclipse GUI. TddMentor uses its internal core functionality to fetch individual source files for reconstructing the integration versions.

In the current implementation, TddMentor requires the user to create the CVS log file manually via the CVS command line. The location of this log file is declared in the analysis descriptor. TddMentor uses the StatCvs library to parse this log file and identify the individual integration deltas.

7.2 Usage

TddMentor seamlessly integrates with the Eclipse GUI as seen in Figure 7.2. It is triggered via the context menu of the analysis descriptor of an application under inspection.

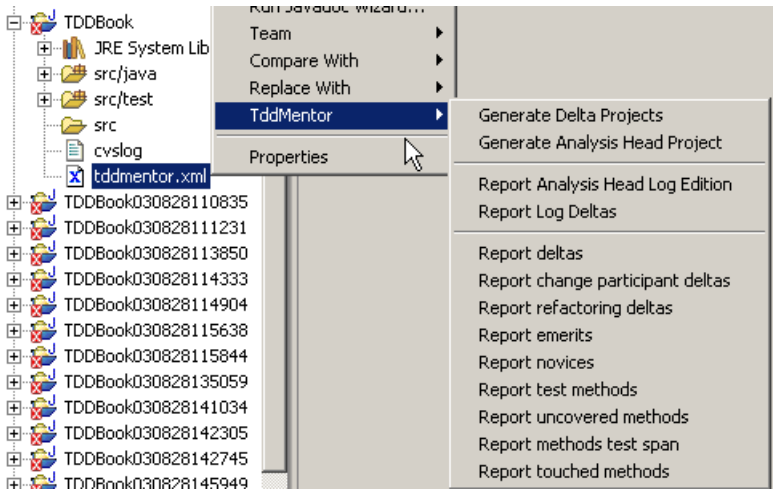

Figure 7.2: *Seamless GUI Integration.* TddMentor plugs into the context menu of the analysis descriptor file. This is located in the project directory of the application under inspection. The projects below contain recreated integration versions.

The steps to perform an analysis are as follows:

1. Identify the application that should be inspected in the CVS browsing perspective and check it out as project.

2. Open a command line on the project directory and create a CVS log file (e.g. `cvs log > cvslog` produced the log file in Figure 7.2).

3. Create an analysis descriptor file as described below in Section 7.2.1.

4. Fill out the `cvslog` and the `project` entries.

5. Generate the integration deltas report (TddMentor≫Report Log Deltas). This report shows the identified integration deltas. Each integration delta has a unique identifier.

6. Identify the analysis head and tail of the integration deltas and fill out the `analysis-scope` entry of the analysis descriptor.

7. Generate the analysis head (TddMentor≫Generate Analysis Head Project). This creates a new Eclipse project and fetches all revisions of the files in the latest integration version.

8. Generate all remaining delta projects within the analysis scope (TddMentor≫ Generate Deltas Projects). Figure 7.2 shows the result of this operation. For every integration version within the analysis scope you see a separate Eclipse project.

9. Generate the desired report (e.g. TddMentor≫Report uncovered methods reports the Test Coverage PCI and the methods that are not covered in this integration delta).

This first report might contain some inappropriate results because of some inappropriate configuration of span claims or coverage claims. TddMentor needs to be calibrated, as described below in Section 7.2.2.

7.2.1 Analysis Descriptor

The analysis descriptor scopes the integration deltas and provides some additional information for the analysis. Table 7.1 explains the analysis descriptor for the Money Example case study (see Section 8.1).

In addition to the configuration elements in Table 7.1, there are some more configuration options as follows.

Coverage claims can also be specified for individual integration deltas. Such a coverage claim would look like:

```
<edition name="030828154711">
   <method type="tddbook.Money" name="currency" paramsig="[]"/>
</edition>
```

It would declare the specified method as covered only in one single integration delta, whereas the coverage claims in Table 7.1 apply to all integration deltas within the analysis scope.

Span claims (see Section 6.2.1) are another possible extension to the above analysis descriptor. Listing 8.3 shows the span claims from the QDox case study.

The `origin-method` entry is the starting point of a span claim. It denotes a method that tries to call some source code that is not available (for example because it would have been generated during a regular build). It contains `method` entries. They denote methods that would have been called by the missing code. Declaring these call targets fixes the broken call graph.

tddmentor.xml	Description
`<cvslog file="cvslog"/>`	Path to CVS log file.
`<project name="TDDBook"/>`	Name of the analyzed project.

`<analysis-scope`
`head="030828154711"`
`tail="030828110835"/>`
	Scope of integration deltas.
	Latest in scope.
	Last in scope.

`<sources>`	Production sources list.
`<dir path="src/java"/>`	Source directory path.
`</sources>`	
`<tests>`	Test sources list.
`<dir path="src/test"/>`	Test directory path.
`</tests>`	
`<coverage-claims>`	List of coverage claims.
`<edition name="global">`	`global` applies for every integration delta in scope.

`<method name="toString" paramsig="[]"/>`
`<method name="equals" paramsig="[QObject;]"/>`
`<method name="hashCode" paramsig="[]"/>`
	Parameter signatures as used by Eclipse Java model.

`</edition>`	
`</coverage-claims>`	

Table 7.1: Analysis descriptor of the Money Example.

7.2.2 Calibration

Calibration is the process of tailoring the current analysis descriptor to the current application under inspection.[1] This step is necessary due to the diversity of projects that are assessed.

After the generation of the first report, a tool user would interpret the results according to the PCI descriptions (see Chapter 6). He or she would then manually inspect the integration versions to check the validity of the suggested process violations. This inspection either confirms a process violation or disproves it.

In case of disproval, the tool user calibrates TddMentor to prevent this method from being reported again. This process repeats until the process assessor has enough confidence in the reported PCIs.

TddMentor calibration uses two different approaches: configuration evolution and tool extension.

[1] Porter and Selby [PS90] describe the calibration of classification trees to identify high-risk components in evolving software systems.

Configuration Evolution

Configuration evolution is the simplest case. It simply adds an entry to the analysis descriptor (such as a span claim or coverage claim).

For example, technical methods (see Section 2.3.4) do not need to be covered by tests but would be reported as uncovered by the test coverage PCI. A coverage claim prevents such a method from reappearing. Similarly, span claims are added to the analysis descriptor.

Tool Extension

The prototypical implementation of TddMentor still has some deficiencies in the automatic detection of refactorings and other safe software changes. In order to produce adequate results, the detection mechanism might need to be extended to further types of refactorings, refactoring participants and change participants.

Such extensions change the Java source code of the existing tool. There is no pluggable extension mechanism available. It is unclear if it is feasible to create such an extension mechanism for refactoring detection.

7.3 Other Application Areas

TddMentor was developed to support TDD process assessment and validate the research hypothesis of this dissertation. Beside this use, TddMentor might be beneficial in a number of further application areas which have been outlined below.

7.3.1 Teaching TDD

Extreme Programming and TDD have been taught at universities [Wil01] [WG01]. Mugridge [Mug03] identified two major challenges in teaching TDD to students:

- fast feedback during the learning process on how to incrementally evolve the design and
- building interdependent technical skills (like writing a test and refactoring source code).

To answer such challenges, Edwards [Edw03b] [Edw03a] proposed a web-based system for automatic grading in teaching TDD. Students submit their test and production code to a grading server that performs a results-based assessment in terms of test coverage and functional completeness.

TddMentor supports a change-based assessment and could also be used in a classroom setting. Students would integrate their test and production code after each basic

development cycle with the central source code repository. At the end of the day, the trainer would assess the repository with the help of TddMentor. This assessment would reveal good and bad performance of TDD practices. The next day of the course would start with a discussion of the findings. The major advantage of this approach is that the learning environment is closer to a real world development project and the students learn to work as part of a team.

7.3.2 Refactoring Mining

Software system mining is the activity of finding valuable information in the source code of a software system in order to make this information explicitly available to developers [Moo02]. Refactoring mining is the activity of finding new types of refactorings in existing source code changes in order to make them explicitly available.

TDD developers continuously refactor their application and possibly apply refactorings that have not been documented. TddMentor could be used to assess source code from advanced TDD teams. It is expected that an advanced TDD team should achieve high scores for the test coverage PCI. Part of its calculation is the detection of refactorings. A low test coverage for an individual integration delta can mean a shortcoming of the refactoring detection mechanism or a new type of refactoring that has not been documented (and implemented in TddMentor) before.

I applied this process for calibrating TddMentor (see Section 7.2.2) in the documented case studies. During that process I found some refactorings that were not documented in the literature. Appendix A lists those refactorings among others for which TddMentor contains detection code.

7.3.3 Test Pattern Mining

Test pattern mining is the activity of finding test patterns in existing source code in order to make them explicitely available.

Similar to refactoring mining, test pattern mining would work by assessing the source code from advanced TDD teams. Again, a low test coverage score would be the starting point. The test coverage algorithm calculates the call graph from the changed unit tests. This call graph calculation requires the tests to call production code statically.

While scanning several projects as potential case studies, I found unit tests that performed dynamic calls only, by providing an invocation target as a string. Further case studies might reveal more types of test patterns.

7.3.4 Empirical Research

Abrahamsson *et al.* [AWSR03] assess that empirical evidence, based on rigorous research, is scarce in the area of agile software development methods. They call for more

empirical, situation-specific research and the publication of the results. A general problem of this endeavour is that TDD produces hardly any artifacts except source code, while it puts a very strong focus on source code and design evolution.

To some extent TddMentor can support such empirical work by:

- reconstructing individual integration versions and making them available inside an IDE,
- identifying all changes within an integration delta and making them accessible for static program analysis,
- heuristically identifying renamed and moved methods to some extent which allows to make observations across such changes, and
- detecting a number of refactorings and other safe software changes.

An analysis across several projects would try to find commonalities within several projects that all use TDD as their style of development. These commonalities would add to the body of knowledge about TDD.

7.4 Future Implementation Improvements

TddMentor exists only as a prototypical implementation. It still has some areas that require improvement.

These limitations do not impact on the validation of the research hypothesis. However, before applying TddMentor in a broader context, these limiatations need to be removed. Here is a list of such limitations:

- TddMentor ignores all modifications in inner classes and non-public top level classes.
- It ignores instance variables completely. As seen in the Money Example case study, this can result in missing data points.
- TddMentor needs to detect more types of refactorings and other safe software changes than it does at the time of writing. The current set of detected safe software changes is biased towards the documented case studies.
- TddMentor does not deal with refactorings in test code. Improving test code requires some specific refactorings [DMBK01].
- TddMentor does not take into account the more advanced test structuring capabilities of JUnit or of its extensions (like MockObjects).

- Maven is a Java project management and project comprehension tool. Its project object model (POM) contains an entry that specifies the locations of the unit tests of a project. TddMentor could read that information to simplify the configuration step.
- The current implementation supports CVS as version control system only. Subversion aims at being a "a compelling replacement for CVS". Commits of a number of files are atomic and revision numbers are per commit, not per file. The reconstruction of integration deltas would be a lot easier in Subversion. Also, some TDD developers link user stories to their Subversion revision.

7.5 Summary

This chapter describes the architecture and usage of TddMentor — the prototypical implementation to support the approach proposed by this dissertation. It is used to validate the research hypothesis as documented in the next chapter.

Chapter 8

Case Studies and Hypothesis Validation

This chapter lists several case studies, each documenting the TddMentor-supported analysis of a project. The goal of those case studies is "theory testing and experimentation" [Tic98]. The case studies serve theory testing by providing real world data for validating the research hypothesis. They were also experiments to refine the implemented concepts; e.g. the original test coverage algorithm did not consider span claims (see Section 6.2).

Zelkowitz and Wallace [ZW98] list several important aspects for experimental data collection. For this research, these aspects are as follows:

Replication The chosen systems are open source systems and freely available (QDox, Ant) or even published in a book (Money Example). This means the raw data are available for repeated analysis. This work describes all relevant algorithms to replicate such an analysis.

Local control The goal of this research is to provide automated support for process assessment. For the sake of validating the research hypothesis, no control over the project's developers was necessary. Only in a future step of a process improvement effort should such a local control be relevant.

Influence Experimental designs have to take their influence on the studied subjects into consideration. I.e. active methods may suffer from the Hawthorne effect.[1] The data collection of this research is passive in the sense that it analyzed the source

[1] One problem of actually observing a project team is the Hawthorne effect [Boe81, p. 672] [May45]. In a classical experiment, Elton Mayo examined the effect of lighting and illumination on employee productivity. In essence, the mere fact of observing and including the observed persons in the observation process had an impact on the results.

code repositories of already existing software. Therefore, the analysis itself did not have any influence on the project's developers.

Temporal properties The data collection for the hypothesis validation is retrospective (or archeological). A more current data collection might apply to a later application of TddMentor (see Section 7.2) in a process mentoring or process improvement context.

Given the retrospective nature of the data collection, Zelkowitz and Wallace might call this a legacy analysis or historical lessons-learned study. The term "case study" is justified by the fact that (1) the analysis has all necessary information available in the source code repository, (2) it can calculate the necessary quantitative data, and (3) the results can directly help an assessed team.

The scope of this thesis is to provide tool support for TDD process assessment. It is outside the scope of this thesis to collect data from a large number of projects for statistical evaluation. This would be a separate research project. TddMentor can support such an endeavour. Therefore I selected two case studies that serve as typical examples for TDD and one case study as a counter-example for TDD.

Each case study is documented in a separate section. The structure of those sections is as follows:

1. Introduce the case study.
2. Outline the scope of the analysis. This might be the whole project history or just a selected time frame. Document the performed calibration steps. The listed refactorings, refactoring participants, and change participants, for which detection mechanisms had to be implemented, are described in appendix A.
3. Document the raw results as reported by TddMentor. Chapter 6 introduces a number of process compliance indices (PCI). The case studies focus on the Test Coverage Rate PCI as reported by TddMentor.
4. Interpret the results and derive recommendations for the authors of the case studies where applicable. Their interpretations are examples for the use of TddMentor in a process assessment context (see Section 7.2).
5. Discuss consequences of the case study for TddMentor and the underlying concepts.

8.1 The Money Example

In his book *Test-Driven Development by Example*, Beck [Bec03] introduces and discusses TDD. In Part I he demonstrates the development of typical production code

completely driven by tests. His goal is to illustrate the principles of TDD by working through the Money Example in the book.

Every chapter introduces a small increment of functionality. For every increment, Beck goes through the three basic TDD steps Red/Green/Refactor (see Section 2.2.1). At the end of every chapter, he has an integration version (see definition 2.1) that is working, fully driven by tests, and consisting of clean code that had undergone the necessary refactorings. All code changes in one chapter form an integration delta. In a real project, a developer would check-in such an integration delta into the revision control system. For the purposes of this case study, I checked-in every such integration delta to CVS in order to be able to use TddMentor for the analysis.

8.1.1 Analysis Setup

Scope

The Money Example consists of 16 individual integration versions. The number of classes varies between 2 and 6. All tests had been merged into one test class for the sake of simplicity of this scholarly example. The author restricted the example to show how tests drive functionality into the system. It was out of the scope of the study to show good testing techniques:

> "One of the ironies of TDD is that it isn't a testing technique. It's an analysis technique, a design technique, really a technique for structuring all activities of development." [Bec03, p. 204]

It is only a small example but it is ideal to study code changes over time in TDD. This analysis covered the whole CVS history of the Money Example.

Calibration

The Money Example was the first case study used to calibrate TddMentor. TddMentor needed to show perfect results for it because with this example Beck demonstrated the concepts of TDD. Making TddMentor show perfect results here helped refine many of the concepts discussed earlier in this work.

TddMentor needed to implement detection code for the refactorings *Reconcile Differences, Replace Constructor with Factory Method* and detection code for the refactoring participant *Add Argument*.

Some methods, however, were still reported as uncovered. A manual inspection revealed that these methods had not been covered by tests intentionally because they had been considered too trivial (see Section 6.2 for a discussion of those exceptions). The project-specific analysis descriptor therefore declares coverage claims for the methods as shown in listing 8.1. Section 7.2.1 explains the syntax of coverage claims.

Listing 8.1 (Coverage claims for the Money Example)

```
<coverage-claims>
   <edition name="global">
     <method name="toString" paramsig="[]"/>
     <method name="equals" paramsig="[QObject;]"/>
     <method name="hashCode" paramsig="[]"/>
   </edition>
</coverage-claims>
```

Span claims (see Section 6.2) were not needed for the Money Example.

8.1.2 Analysis Results

Figure 8.1 shows the test coverage rate over time. One can see complete test coverage for all integration deltas except for one missing data point in the third integration delta.

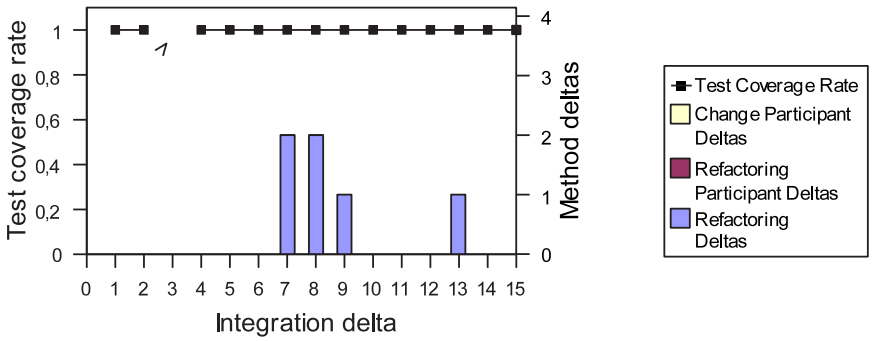

Figure 8.1: *Test coverage rate and method deltas of the Money Example.* All integration deltas except number three had a test coverage rate of one. This exception was due to a missing data point that revealed a limitation of TddMentor as discussed below. Some integration deltas required the calculation of refactoring method deltas.

A manual inspection of the integration delta that missed the data point revealed that (1) in the test code, a reference to an instance variable had been removed and that (2) in the production code, the visibility of this instance variable had been changed to `private`. TddMentor takes the number of changed production code methods as divisor for calculating the test coverage rate. The set of production code methods in this integration delta being empty lead to the missing data point.

TddMentor detected a number of method deltas that were not spanned by tests. All those method deltas were identified as refactorings. TddMentor accepted all those method deltas as covered (see Section 6.2). Figure 8.1 also shows the number of those method deltas over time.

The figure only shows method deltas that were needed to calculate the test coverage rate. TddMentor might have identified other method deltas between two integration versions. They are not shown because they were irrelevant for the calculation of the test coverage rate. The objective behind the method delta detection for this PCI is the correct calculation of the test coverage rate.

8.1.3 Discussion of Results

The test coverage rate history in Figure 8.1 is not very surprising, as the Money Example is a scholarly example for demonstrating TDD. Therefore, a complete test coverage for every individual integration delta could have been expected.

The noted exception reveals a limitation of TddMentor. At the time of writing it has only analyzed changes in methods. Therefore the change of an instance variable went unnoticed. The manual inspection showed a full test coverage for this integration delta.

Even though the Money Example starts from scratch and in small steps, very soon the code needs to be cleaned up by applying some refactorings. The detected refactoring method deltas also shown in Figure 8.1 were essential for reaching a test coverage rate of one for those integration deltas. Without the detection of those method deltas, the test coverage rate would have been below one.

Clearly the production code had been refactored while the developer could rely on the tests that had been written for the preceding integration deltas. A small amount of production code changes could be explained by the application of refactorings. New functionality had been driven into the system by new tests. This introduction of new functionality had triggered the need to refactor the already existing code. Without this new functionality, the refactorings would not have been appropriate in the TDD sense. In TDD, a redesign is always the third of the three steps of a basic development cycle.

Summary and Recommendations

Unsurprisingly, I do not have any recommendations for Beck on how to improve his TDD process compliance.

8.1.4 Discussion of Case Study

TddMentor produced results as expected. The example illustrates the basic TDD development cycle. Therefore, it was expected that it would comply to this process.

The test coverage rate could only be calculated correctly because TddMentor detected some refactorings in the source code. This shows the necessity of the refactoring detection mechanism.

The missing data point in Figure 8.1 reveals a limitation of TddMentor; it ignores instance variables completely. Dealing with instance variables will be left as future work.

Another PCI is the distribution of all refactorings across a period of time as described in Section 6.3. However, the test coverage algorithm only needs the method deltas of those methods that were not spanned by tests already. In order to make a more extensive analysis of refactorings, TddMentor needed to detect more types of refactorings than it does at the time of writing.

8.2 QDox

QDox [URL:QDox] is a high speed, small footprint parser for extracting class, interface, and method definitions from source files complemented with JavaDoc @tags. It is designed to be used by active code generators or documentation tools. QDox implements a custom built parser using JFlex [URL:JFlex] and BYacc/J [URL:BYacc]. It is open source.

According to its developers it is one of the best examples of a completely test-driven application. Many of the developers of QDox are affiliated with ThoughtWorks [URL:ThoughtWorks], a company well known for its leadership in agile methodologies and TDD. At the time of writing, QDox had 5 developers and listed 6 contributors that "have contributed to this project through the way of suggestions, patches or documentation."

8.2.1 Analysis Setup

Scope

From September 2002 to year end 2003, QDox summed up to over 120 individual integration deltas. At that time the system contained around 20 manually created production code classes and around the same number of test code classes.

The classes that would have been generated from the lexer and parser specifications were not taken into account for the analysis. TddMentor could not run build processes for the individual integration versions, therefore it had no chance to generate classes from the specifications.

This analysis covered 29 integration deltas from September 2003. The selected integration deltas contained mainly changes in Java code. Other integration deltas showed changes in the lexer or parser specifications or in other non-Java artifacts.

Calibration

QDox is used in practice. It is not an example project, and its developers claim that QDox is a good example for applying TDD. Therefore, similar to the Money Example, this case study was expected to show good results and help validate and refine the concepts of TddMentor. Moreover, this case study provided numbers that are closer to actual project reality than the Money Example.

TddMentor needed to implement detection code for the refactorings

- *Extract Method*,
- *Expose Method*,
- *Move Method*,
- *Remove Parameter*,
- *Replace Exception with Error Code*,

and detection code for the refactoring participants

- *Add Argument*,
- *Remove Argument*,
- *Add Leftmost Invocation*,
- *Remove Leftmost Invocation*,
- *Delegate Constructor*,
- *Delegate Method*,

and the change participant *Change Argument*.

Within the given scope, the QDox analysis did not require coverage Claims; however, it did require span claims (see Section 6.2).

Listing 8.2 (A method that called generated code.)

```
public JavaSource addSource(Reader reader) {
   ModelBuilder builder =
     new ModelBuilder(classLibrary, docletTagFactory);
   Lexer lexer = new JFlexLexer(reader);
   Parser parser = new Parser(lexer, builder);
   parser.parse();
   JavaSource source = builder.getSource();
   sources.add(source);
```

```
   addClasses(source);
   return source;
}
```

For an example of span claims, see the method in listing 8.2 (section 7.2.1 explains the syntax of span claims). In a regular build, the classes `JFlexLexer` and `Parser` would have been generated from the lexer and parser specifications. Calls to instances of those classes would have been part of the method call graph. Their absence resulted in an incomplete method call graph and thus in an incomplete test span. Span claims as listed in listing 8.3 filled this gap.

Listing 8.3 (Span claims for the QDox analysis)

```
<span-claims>
   <origin-method type="com.thoughtworks.qdox.JavaDocBuilder"
                  name="addSource" paramtype0="Reader">
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="addPackage" paramtype0="String"/>
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="addImport" paramtype0="String"/>
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="addJavaDoc" paramtype0="String"/>
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="addJavaDocTag" paramtype0="String"
             paramtype1="String" paramtype2="int"/>
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="beginClass" paramtype0="ClassDef"/>
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="endClass"/>
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="addMethod" paramtype0="MethodDef"/>
     <method type="com.thoughtworks.qdox.parser.Builder"
             name="addField" paramtype0="FieldDef"/>
   </origin-method>
</span-claims>
```

8.2.2 Analysis Results

Figure 8.1 shows the test coverage rate over time and the method deltas that were detected during the test coverage calculation.

The diagram shows complete test coverage except for four integration deltas.

A manual inspection revealed that at data point A, the integration delta contained a new convenience method. This method had been introduced without a test and it was

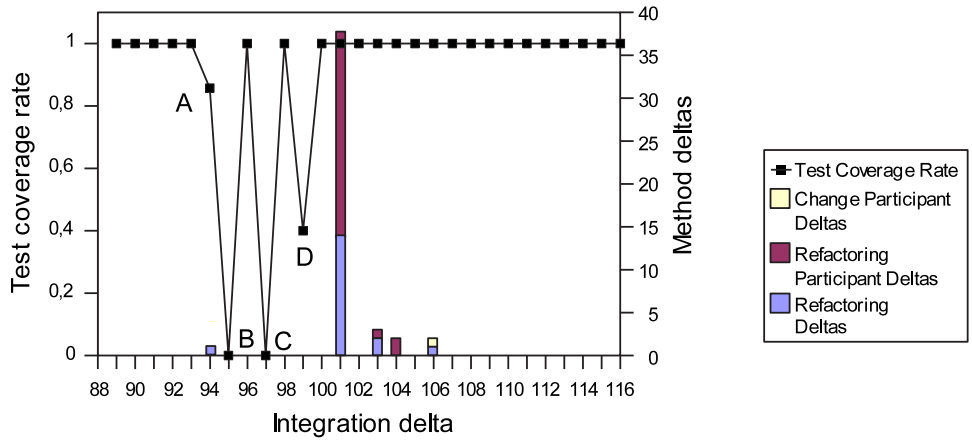

Figure 8.2: *Test coverage rate and method deltas of QDox.* The test coverage rate was one, except for a few integration deltas. Those exceptions are discussed in the text. A number of integration deltas required the calculation of method deltas.

not referenced from anywhere within QDox. This resulted in a test coverage rate below one.

At data point B a new JUnit test class had been introduced. This class had been stored, however, in the production code directory.

A new method in a public interface was responsible for the non-existing test coverage in data point C. No test and no implementation were found in the system during manual inspection.

For data point D the integration delta contained three new methods that had no tests. One method contained some initialization code that relied on a new helper method. These two methods called accessor methods of an other class and as such they were not reported as uncovered (see Section 5.4.4). The third method had introduced a new `catch` statement in the method's body to discard some exception.

For the calculation of the test coverage, TddMentor needed to detect method deltas of all three kinds (refactoring method deltas, refactoring participant method deltas and change participant method deltas). The number of detected method deltas was mostly similar to the money example. In integration delta 101, a *Move Method* refactoring for several methods had resulted in some signature and many method body changes due to the new context of the moved methods.

8.2.3 Discussion of Results

For nearly every new functionality, the repository contained a test as requested by the TDD process. This is a strong indication that new functionality was really driven into the system by tests.

The exceptional data points (A to D) in Figure 8.2 do not invalidate this result. For a real world project a complete process compliance could not be expected — as opposed to the scholarly Money Example.

The convenience method in data point A is a public method and as such part of the interface to the user. One could argue that it does not require a separate test case because it only combines existing functionality. Yet the method is part of the public API and its logic is complex enough that I would dedicate a separate test case to it.

Initialization logic in data point D was important enough for requiring a dedicated test. The additional `catch` statement in the third uncovered method in data point D revealed a general problem of TDD. Exception-handling code deals with situations where something goes wrong. Some of those race conditions never occur during development and occur only in a production environment. In such cases, a developer should write a test that reproduces the race condition before changing the production code. That means that in the integration delta for data point D, there should have been a test case that would have led to the exception. Such a test case might be very hard to write with reasonable effort. For the case at hand, this means that a human process assessor would have had to decide whether the missing test for the `catch` statement was a TDD process violation or not. A coverage claim (see Section 6.2) could have prevented the method to be reported in subsequent runs of TddMentor.

The test code in the production code directory for data point B seemed like a major error. A final judgement however should only be made after a review with the developer of that code.

The integration delta of data point C contained a new method in a Java interface without having either a test or an implementation. This was a TDD process violation in several ways. First, it is clear that this new method had not been driven into the application by tests. It also was not the result of a refactoring which would not have needed a separate test — this new method is the only change in the whole integration delta. Second, the new interface method extends an abstraction without providing an implementation for that extension. This strongly smells of “Needless Complexity”.

Summary and Recommendations

The high reputation of the QDox developers had promised a good TDD process compliance. The source code history of QDox was holding this promise.

However, the developers should review the discovered exceptions in order to avoid such process violations in the future.

8.2.4 Discussion of Case Study

TddMentor produced results that met the expectations for the known TDD system. Still it found some areas that require improvement and thus demonstrated the value of such an analysis.

Method delta detection was necessary to produce correct results for the test coverage. The case study demonstrates that it is possible to detect those method deltas in non-scholarly code.

The case study also demonstrated how to find potential TDD process violations. It recommended that the developers should review those process violations.

The uncovered convenience method in the integration delta of data point A could fuel an expert discussion about when not to test (see Section 6.2).

8.3 Ant

Ant [URL:Ant] is a Java build tool hosted by the Apache [URL:Apache] software foundation and was meant to replace the well-known "make" tool for building the Tomcat servlet engine [URL:Tomcat]. The Ant project has around three dozens of "committers". Within a few short years it has become the de facto standard for building open source Java projects. It is also used in a large number of commercial projects and with other languages like C++ and C# [HL03].

Ant's source code is written in Java and is freely available as open source. It uses CVS for keeping its revisions. The Ant developers use JUnit for writing unit tests. The development process is incremental and most increments are directly integrated into the system. Ant applies a nightly build process with fast feedback from its users by their nightly builds for other open source Java projects.

Within the context of this research, Ant serves as a counter-example. The Ant project has some aspects that would qualify for TDD (like incremental development and fast integration). However, it is not developed in a test-driven way (at least not within the scope of the analysis presented here). See the following quote from the Ant developer mailing list.

> "Testcases tend to be added to ant when problems have appeared, for instance to demonstrate that a change fixes a bug report from Bugzilla. In some instances, test cases are added proactively, to ensure that functionality will be preserved after a change.
>
> One of the problems with testcases is that a lot of tasks require specific external resources (databases, application servers, version control systems, SMTP mail server, ...) which do not always exist and do not exist under the same name everywhere. These tasks are often very badly covered by testcases."[2]

This is clearly not a TDD process where new functionality would be driven by tests. Also the testing techniques are not very elaborate in Ant. TDD typically applies a lot

[2] Posting from September, 21st 2003 in the ant-dev mailing list [URL:AntDev].

of highly sophisticated testing techniques (like mock objects) to get around limitations such as mentioned in the posting. Link and Fröhlich [LF03] list a number of sophisticated testing techniques for TDD.

At the time of the analysis scope, TDD had not been described as a method. Elements of it had been available and were partly described for example in [Bec00]. At that time, the approach was named Test-First Programming.

8.3.1 Analysis Setup

Scope

This analysis comprised 34 individual integration deltas from July/August 2000. The number of production classes was around 110. The number of test classes was around 7. It was not a big application but of reasonable size for a system in daily production use.

During the chosen time frame, the use of JUnit was in the beginnings. First JUnit test classes had appeared in the repository. Many integration deltas did not contain any tests. The existing tests had a very simple structure that TddMentor could analyze easily.

At a later stage of the Ant development, the source repository contained unit tests that extended the JUnit testing framework. These tests were still not very elaborate, as witnessed in the posting above, but they were less accessible to analysis by TddMentor than the tests at the early stage of JUnit usage within Ant. Therefore, this analysis was scoped in the same way.

Calibration

Within the analysis scope, there was no need to apply span claims (see Section 6.2). Also, I have not seen any possibility to apply coverage claims. TddMentor did not need to implement any new detection code.

8.3.2 Analysis Results

Figure 8.3 shows the test coverage rate over time. Most integration deltas were not covered by tests at all. The integration deltas simply did not contain any test class.

The integration delta for data point A had a moderate test coverage rate. A manual inspection revealed that this was the only integration delta within the analysis scope that comprised changes in production code and in test code. That test code covered some of the production code. The change participants of the same integration delta (also depicted in the diagram) extended the coverage. The resulting test coverage rate, however, was still below one.

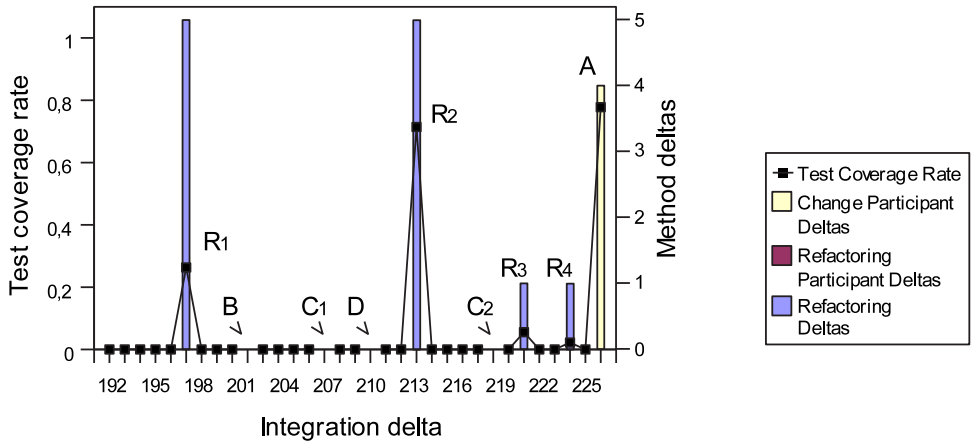

Figure 8.3: *Test coverage rate and method deltas of Ant.* Most integration deltas were not covered by tests at all. Some peaks show moderate or weak test coverage. Note the missing data points. All marked data points are discussed in the text.

The integration deltas R_1 to R_4 showed a test coverage rate above zero. This was exclusively due to the detected refactorings that are also depicted in the diagram. All those integration deltas did not have any new or changed test.

B denotes a missing data point. A manual inspection showed that some existing test cases had been refactored. No production code had been touched. This caused TddMentor to divide by zero and thus resulted in the missing data point.

Integration delta D also had no test coverage rate. A manual inspection showed that this integration delta only contained the deletion of two files. The algorithm to calculate the test coverage rate counted the touched methods that continued to stay in the system. Methods of deleted classes did not add to the divisor which resulted in the missing data point.

The missing data points denoted by C_1 and C_2 were caused by changes in inner classes. As seen earlier, TddMentor could not deal with inner classes which resulted in the two missing data points.

8.3.3 Discussion of Results

Even though Ant had a JUnit test suite during the analyzed period, new tests were only added occasionally. New functionality had not been driven by tests. Even in the case of data point A, where new tests had been introduced, the test coverage was not complete. This indicates that the production code changes were not driven by tests but rather accompanied by tests. This observation matches the description of the development process in the posting from the Ant developers' mailing list above.

The test coverage rate peaks R_1 to R_4 give a false picture of the actual development process. The test coverage algorithm (see Section 6.2) assumes that refactored methods do not have to be spanned by tests, because they had been spanned in earlier integration deltas. The test coverage rates of the surrounding integration deltas in Figure 8.3 clearly indicate that this assumption does not hold here. The objective of the refactoring detection is to calculate a correct test coverage rate in the case of a TDD process as seen in Figures 8.1 and 8.2. For the non-TDD process of Ant, the data points R_1 to R_4 should have been zero. A human process mentor would identify such false test coverage rates by the fact that they are solely induced by detected refactorings and that surrounding integration deltas have a poor or not existing test coverage (as exemplified in Figure 8.3).

The refactoring of test code for data point B in Figure 8.3 also shows that the developers were still working on their testing techniques. In TDD, refactoring is one step of the basic development cycle. This applies to production code as well as to test code. A refactoring of test code would not occur without the need to introduce new functionality into the system.

The deletion of the production code for data point D also reveals an understanding of refactoring that is not compliant with TDD. In TDD, every development cycle ends with a refactoring step. This is the time when dispensable code would have gone out. It also would have required the deletion of the tests that had triggered the production code to be in the system. No test code had been deleted in D. Hence the deleted production code had obviously not been driven into the system by tests.

The missing data points C_1 and C_2 clearly show a limitation of TddMentor to deal with inner Java classes at the time of the analysis.

Summary and Recommendations

The development process of Ant during the analyzed period was not TDD. Especially

- new functionality was occasionally accompanied by tests — not test-driven into the system,
- the testing techniques were not elaborate enough, and
- refactorings were not performed as a mandatory step of the basic development cycle.

The existence of a JUnit test suite and its occasional use as exemplified in A and B can be a start for evolving a development process that is more test-driven. The developers who had already worked with the test suite can be an example for the other team members. However, even those developers still needed to refine their testing techniques.

8.3.4 Discussion of Case Study

This case study shows how a process that did not conform to TDD could be discovered. A manual inspection of the code confirmed the automatically calculated results. The case study also demonstrates the deduction of concrete recommendations for the project's developers in getting closer to a TDD process.

The case study is a counter-example for a TDD process. It supports the relevance of the data reported by TddMentor.

At the same time, it reveals some limitations of TddMentor. E.g., it does not deal with refactorings in test code. The case study also shows some limitations of the underlying concepts. E.g., they do not cover inner Java classes at the time of writing. Such cases can result in missing data points. Their solution, however, is left as future work because those cases have only a minor impact on the results reported by TddMentor.

8.4 Hypothesis Validation

In their preliminary guidelines for empirical research in software engineering, Kitchenham *et al.* [KPP+02] see two types of studies for validating a research hypothesis: observational studies and formal experiments. The combination of the case studies of this chapter takes the form of an observational study. More specifically, it resembles a qualitative effects analysis which "provides a subjective assessment of the qualitative effect of methods and tools" [Kit96]: TddMentor extracts quantitative information from a source code history. However, a human subject still has to interpret the numbers for translating them into concrete recommendations.

8.4.1 Case Study Selection

For a validation to be significant, the selected case studies should be representative for the field of study [KPP95].

Sim *et al.* [SEH03] propose the use of benchmarks to advance software engineering research. They define benchmark as "a test or set of tests used to compare the performance of alternative tools or techniques". A traditional example of technical benchmarks is TPC-A — the Transaction Processing Council (TPC) Benchmark™A for Online Transaction Processing including a LAN or WAN network [Gre91]. Demeyer *et al.* [DMW01] propose a benchmark for software evolution. For TDD there is no benchmark available. It would consist of a set of typical test-driven programs.

In the absence of some form of benchmark for TDD, I had to select some case studies. Table 8.1 summarizes the state variables of the three case studies. All three projects use Java as programming language and JUnit as unit testing framework. The Money Example (see Section 8.1) is a scholarly example of how to apply TDD. Therefore, we can

have high confidence in its TDD process compliance. Also narrative evidence suggests that QDox can be seen as exemplary demonstration of TDD. QDox (see Section 8.2) and Ant (see Section 8.3) can be regarded as good examples of real world projects applied in industry and with experienced software developers. They are not industrial projects. As open source projects they are much more open than a closed industrial project and a high commitment of the individual developers can be assumed.

State variable	**Money Example**	**QDox**	**Ant**
TDD process compliance	highest	high	accidential
Number of classes (prod/test) at time of analysis scope	6(5/1)	40(20/20)	117(110/7)
Number of analyzed integration deltas	16	29	34
Number of developers at time of analysis scope	1	5	14
TDD experience of developers	highest	high	poor

Table 8.1: *State variables of the studied projects.* The Money Example and QDox are representative TDD projects. Ant is a representative counter example.

8.4.2 Confounding Factors

When the effect of one factor cannot be properly distinguished from the effects of another factor, the two factors are *confounded* [KPP95]. For avoiding a negative effect on the internal validity, the case studies minimized the confounding factors.

TDD is not the only method that uses automated unit tests. Unit tests are probably still more often written after the implementation of some production code. The application of JUnit in Ant can be seen as a typical example for this. Even Beck [BG98] had been working in that order before he developed TDD. Thus the mere presence of unit tests does not suffice to identify a TDD process. It is rather the relentlessness of unit testing and the very high test coverage rates that are a strong indication, together with the iterative and incremental design changes if they are driven by tests. TddMentor can only provide the raw numbers; they still have to be interpreted. The Ant case study gives an example of the application of JUnit tests in a non-TDD way. As discussed in the case study, a manual inspection can minimize the effect of this confounding factor.

Every basic development cycle of a TDD process has to leave with the design being as good as possible by iteratively refactoring the new feature into the system design. Some process compliance indices (see Chapter 6) incorporate those design changes. However, TDD does not prevent non-TDD developers from creating good designs and effectively applying refactorings. A TddMentor-based assessment takes the

test-coverage as the most important process compliance index. Not only new functionality but also design changes have to be driven by tests. A rejection of design changes without tests minimizes this confounding effect.

Another possibly confounding factor is that the selected case studies are at both ends of the spectrum. Two case studies are exemplary TDD projects. One case study has some TDD elements more by accident rather than on purpose. Projects somewhere in between might be more difficult to analyse correctly and therefore might produce inaccurate results. The objective of TddMentor, however, is to support a TDD process assessor. It is not meant to quantify the performance of a development team. The TddMentor results give hints where the effort of a manual inspection might be invested best. Therefore the analysis of a project within the spectrum might result in a higher assessment effort but does not affect the internal validity of this research.

One might argue that the selection of the case studies was biased towards the capabilities of TddMentor, which is still a prototype. This argument might hold in the sense that other analytical contexts such as new types of refactorings, other types of source code organization might have required more development effort for TddMentor. However, all detection mechanisms cover only well-known and well-documented concepts (like agile design changes, refactorings, etc.). Those concepts are very likely to appear in other case studies that could have been selected. Therefore such a selection bias as a confounding factor can be neglected.

A similar confounding factor might be the limitations of TddMentor at the time of the analyses as documented above. Some unhandled changes of inner classes caused TddMentor to omit some data points. However, for the demonstration of a good TDD process in the case of the Money Example and QDox, this limitation was not an obstacle. In the case of Ant, only two data points were lost due to changes in inner classes and the remaining data points provided enough evidence as to neglect the confounding influence of TddMentor's limitations.

The calibration phase of a TddMentor-supported assessment is another possibly confounding factor. A false calibration could potentially invert the results reported by TddMentor. For the three presented case studies, the calibration was manually checked with great care.

8.4.3 Conclusion of Validity

Chapters 2 and 3 state the theory of Test-Driven Development, agile design changes and process assessment. Chapter 4 describes the research hypothesis. Chapter 6 derives the process compliance indices (PCIs) from the theory. Chapter 5 describes the algorithms that allow the PCI calculations. Chapter 7 outlines TddMentor that implements the described approach. This chapter discusses several case studies of the application of TddMentor. The reported results match the expectations for the selected projects.

In summary, this discussion provides strong evidence for the validity of the research hypothesis.

Chapter 9

Summary and Conclusions

9.1 Summary of Contributions

This dissertation makes the following important contributions:

1. It shows how a better understanding of past software changes can support a TDD process assessment.

2. It shows how the interdependence of test and production code can be leveraged for better understanding of past software changes.

3. It introduces the concept of Process Compliance Indices (PCIs) as indicators for TDD process violations. PCIs help focus a manual assessment effort. This dissertation lists a number of specific PCIs, shows how to calculate them via a static program analysis of source code changes, and how accurately they indicate process violations in some case studies.

4. It explores the automatic detection of a number of safe software changes via delta mapping of abstract syntax trees; namely refactorings, refactoring participants, change participants and accessor methods. The detection of safe software changes is required by some PCI calculations. A prototypical detection algorithm was developed as part of this research.

5. It provides TddMentor, which is a prototypical implementation of the presented concepts. TddMentor makes the past source code changes, that are buried in a source code repository, accessible for static program analysis. Without this extraction of integration deltas, an analysis of the software changes in the source repository would not be manageable. Such analysis enables this dissertation's research but can also enable further research about source code evolution in TDD.

9.2 Limitations of Approach

There are several limitations to the proposed approach:

1. TDD combines well-known software engineering practices (e.g. unit tests, principles of good design, etc.) in a new way. The comprehensive description of the interaction of those elements in TDD as a development style is relatively young. The expert debate about all details in this respect has not come to an end yet.

2. The explicit declaration of integration deltas by the developers is rather the exception than the rule. The approach relies on a heuristic for the identification of integration deltas. While this heuristic provides no proof for being correct, it shows good results in the documented case studies.

3. This research concentrates on Java source code. It neglects all other source artifacts (like grammar specifications).

4. TddMentor performs only static program analysis. It has no capability to build and run the individual integration versions. This prevents the use of some dynamic analysis techniques.

5. The proposed algorithm for calculating test coverage requires to detect refactorings and other safe software changes in source code. A detection of such safe software changes might be complicated if developers mix refactoring activities with corrective and extending changes (and thus violate the “two hats” principle).

9.3 Application Considerations

“Overreliance on tools and procedures, and underreliance on intelligence and experience are recipes for disaster.” [Mar03, p. 202] The proposed approach only supports human judgement, it does not replace it. TDD process assessors apply their human judgement and process expertise during TDD process assessments. The proposed tool — TddMentor — calculates some heuristics that indicate TDD process violations. These indications still have to be judged by an individual. However, the use of TddMentor helps focus the assessment effort and thus allows a process assessor spend his or her time more effectively.

This dissertation does not impose a specific assessment approach on the expert. The inspection of source code as output of a development process is part of process assessment practice. TddMentor supports a process assessor during this activity by making past source code changes accessible and manageable, that are normally buried in the source code repository.

Today, a TDD process assessor still has to work in process mentoring sessions to see how individual developers change software. By facilitating the assessment of past source code changes, TddMentor allows to decouple this assessment activity from mentoring actvities and thus gain a more comprehensive understanding of what happened in the past of a development project. In that spirit, it is comparable to a refactoring tooling inside an integrated development environment (IDE). The availability of such tools enabled refactorings to become common practice. Similarly the availability of tools like TddMentor could increase the amount of inspections of source code changes.

Software developers might be tempted to learn TDD alone by applying TddMentor. While TddMentor implements some TDD expertise as PCIs, it cannot replace human TDD expertise. To a certain extent this expertise can be acquired by literature study, insight and discussion with peer developers. TddMentor can facilitate the study of own source code. Also it can facilitate the study of source code that was produced by others. The study of source code from respected TDD teams (as in the case of QDox), can give a wealth of insight into how source code is changed in a TDD world.

9.4 Areas of Future Research

"Empiricists in software engineering often complain about the lack of opportunities to study software development and maintenance in real settings." [Sea99] A TDD project continuously creates a huge amount of data in the form of source code changes, that wait for their exploration. A broader evaluation of a large number of TDD projects would add to the knowledge about TDD and other agile methods in general. TddMentor could enable such an undertaking.

New process compliance indices should be explored. For example, Beck [Bec03, p. 85] expects the number of changes per refactoring to follow a "fat tail" or leptocurtotic profile. Such a profile resembles a standard bell curve but with more extreme changes than predicted by a standard bell curve [Man97]. Beck does not provide proof for this statement but points to a possibly fruitful area of research. This dissertation facilitates such a kind of research by providing a way to identify the integration versions in real world TDD projects and detect refactorings in past software changes.

I expect that the topic of finding refactorings will attract more research interest in the near future. The software engineering research community starts to understand the importance of refactorings [MDB+03]. When design patterns were first proposed, they were seen as tools to help design computer programs [GHJV95]. Nowadays they are reverse engineered from existing code to better understand program design [Bro96] [SS03]. It is my conviction that refactoring detection will see a similar career to better understand source code evolution.

9.5 Conclusions

The agile community stresses the importance of software as the most important *output* of a software development team, leading to a continuous flow of source code changes. The view on past source code changes as *input* for a better understanding of how a team produces the software is a topic that deserves much more attention than it has received thus far.

While there are still some problems to solve before the exploitation of source code changes can become day to day practice, these problems need to be solved to gain a better understanding of how TDD teams change software, where they had been lax in following the process discipline, and to help them improve their development practices.

This thesis is a step towards their solution.

Appendix A

List of Detected Software Changes

This chapter lists and describes the refactorings, refactorings participants, change participants, and method deltas that TddMentor can detect at the time of writing.

A.1 Refactorings

Most of the refactorings are taken from the literature. For each refactoring, the reference is given after its name. Refactorings without reference have, to my knowledge, not been documented elsewhere.

All refactorings in this chapter have the same format. They have four parts which are as follows:

- The **name** identifies the refactoring. If available, followed by a **reference** to where it is described in more detail.
- The summary of the **situation** in which you need the refactoring is rendered as regular text.
- The summary of the **action** you take to perform this refactoring is *emphasised*.
- A short **description** containing just enough information to understand the refactoring in the context of this dissertation.

The refactorings in this list do not contain comprehensive motivations, mechanical procedures or examples. Please refer to the existing literature for these details.

A.1.1 Add Parameter [Fow99]

A method needs more information from its caller.

Add a parameter for an object that can pass on this information.

A very common refactoring. Authors regularly advise against this refactoring because it is so obvious. Several alternatives to this refactoring are possible (such as deriving the needed information from the already available parameters). However, due to the incremental nature of TDD, it is sometimes inevitable.

A.1.2 Collapse Hierarchy [Fow99]

A superclass and subclass are not very different.

Merge them together.

With all the refactorings during agile development, a class hierarchy can easily become too tangled for its own good. It is often the result of pushing methods and fields up and down the hierarchy.

A.1.3 Expose Method

A private method could reasonably be used by another class.

Make the method public.

Wake [Wak03] lists this refactoring as missing in Fowler's catalog. Fowler [Fow99] only documents how to restrict the visibility of a method because, as he argues, it is easy to spot cases in which one needs to make a method more visible.

This refactoring is listed here because TddMentor needed to implement detection code for this refactoring. It was observed in the QDox case study together with *Move Method*.

A.1.4 Extract Method [Fow99] [Bec03]

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

It is one of the most common refactorings. This refactoring makes the code more readable and saves comments in the source code. In TDD, it is often used to remove duplication — left-overs of the second basic development step.

A.1.5 Extract Superclass [Fow99]

You have two classes with similar features.

Create a superclass and move the common features to the superclass.

Another way to remove duplication. It creates a new abstraction and generalizes the common functionality. The refactored object-oriented code uses the built-in mechanism to simplify this situation with inheritance.

A.1.6 Move Method [Fow99] [Bec03]

A method is, or will be, using or used by more features of another class than the class on which it is defined.

Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation or remove it altogether.

Fowler sees moving methods as the “bread and butter of refactoring”. It is good at uncovering unwarranted preconceptions and helps to disentangle a design. Sometimes only a part of a method needs to move. This part can be extracted by *Extract Method*.

A.1.7 Reconcile Differences [Bec03]

Two methods contain similar looking pieces of code.

Gradually bring them closer. Unify them only when they are absolutely identical.

The second basic development step in TDD often leaves duplication in the source code. This refactoring is one of the most common activities to remove duplication — as described by Beck. It is not a classical refactoring since it relies strongly on the test suite rather than being semantic-preserving under all circumstances. It also is not described with as much rigor as, for example, the refactorings from Fowler. It works on several levels of scale: similar loop structure, branches of a conditional, methods, and classes.

A.1.8 Remove Parameter [Fow99]

A parameter is no longer used by the method body.

Remove it.

Programmers often add parameters but are reluctant to remove them. Fowler advices to remove not required parameters because they indicate a meaning which does not exist and hence make the method harder to use.

A.1.9 Rename Method [Fow99]

The name of a method does not reveal its purpose.

Change the name of the method.

Methods should be named in a way that communicates their intention. Due to the evolutionary nature of the development style, a method might change its intention or it may have been inappropriately named initially.

A.1.10 Rename Parameter

The name of a formal parameter does not reveal its purpose.

Change the name of the formal parameter.

This refactoring is so simple and basic that it is not described elsewhere. However, for a refactoring-aware IDE like Eclipse, it is one of the most important to provide support for. In the documented case studies, it is one of the most commonly detected refactorings.

A.1.11 Replace Constructor with Factory Method [Fow99]

You want to do more than simple construction when you create an object.

Replace the constructor with a factory method.

The most obvious motivation comes from replacing a type code with subclassing. You have an object that often has been created with a type code but now needs subclassing. A constructor can only return an instance of its own type and thus the constructor is replaced with a factory method [GHJV95].

A.1.12 Replace Exception with Error Code

A method throws an exception to indicate an error that should never happen and the exception disrupts the method's interface.

Return a special error code instead.

Fowler [Fow99] argues for the inverse refactoring because exceptions are a better way to deal with race conditions. However, there are rare cases when this refactoring might apply, such as that illustrated in the example below. The exception should never occur as, after making the method public, it would disrupt the method's interface.

The refactoring is applied at one's sole discretion. While Fowler argues for the inverse refactoring and it is arguable whether this refactoring makes sense or not, it exists in practice (e.g. in the QDox case study). Therefore, TddMentor needs to be able to detect this refactoring.

Example

```
// This method will fail if the method isn't an accessor or mutator,
//  but it will only be called with methods that are, so we're safe.
private Type getPropertyType(JavaMethod method) {
     Type result = null;
     if (isPropertyAccessor(method)){
         result = method.getReturns();
     } else if(isPropertyMutator(method)){
         result = method.getParameters()[0].getType();
     } else {
         throw new IllegalStateException("Shouldn't happen");
     }
     return result;
}
```

```
/**
  * @return the type of the property this method represents,
  * or null if this method
  * is not a property mutator or property accessor.
  * @since 1.3
  */
public Type getPropertyType() {
     Type result = null;
     if (isPropertyAccessor()){
         result = getReturns();
     } else if(isPropertyMutator()){
         result = getParameters()[0].getType();
     } else {
         result = null;
     }
     return result;
}
```

A.2 Refactoring Participants

The format of refactoring participants resembles the format of refactorings (see Section A.1). Instead of a description you find a reference to the refactoring in which it participates.

A.2.1 Add Argument

A method call misses an argument to the refactored method.

Add required argument to referencing call.

Participant to: *Add Parameter*

A.2.2 Add Leftmost Invocation

A referenced method is moved to a different class.

Prepend an instance of the target class to the referencing method call.

Participant to: *Move Method*

A.2.3 Delegate Constructor

A new parameter is added to a constructor but the old constructor interface is still needed.

In the old constructor delegate to the new constructor, using a default argument for the new parameter.

Participant to: *Add Parameter*

A.2.4 Delegate Method

A new parameter is added to a method but the old method interface is still needed.

In the old method delegate to the new method, using a default argument for the new parameter.

Participant to: *Add Parameter*

A.2.5 Remove Argument

A method call has too many arguments for the refactored method.

Remove the argument corresponding to the removed target method parameter.

Participant to: *Remove Parameter*

A.2.6 Remove Leftmost Invocation

A referenced method is moved to the same class from another class.

Remove the instance to the originating class from the referencing method call.

Participant to: *Move Method*

A.3 Change Participants

The format is the same as for refactorings (see Section A.1).

A.3.1 Change Argument

A parameter type of a referenced method changes.

Change the argument in the referencing method call to match the new parameter type.

Listings 2.5 and 2.6 give an example of this change participant. Such a change can be assumed to be safe. In general, it is arguable to what extent change participants are safe software changes. This change participant was the only one that was found during the assessment of the case studies. The discovery and discussion of further types of change participants might be an interesting subject for future research.

A.4 Method Deltas

A method delta is a difference between two methods' abstract syntax trees. This section lists and describes method deltas that TddMentor can detect. Section 5.4.2 documents the detection conditions in detail for some selected method deltas.

Many of the method deltas listed here have the same name as the safe software changes for which they are used. Within the text, the difference between method deltas and safe software changes is made clear by the context in which they are used and they are rendered differently, as declared in Section 1.3.1.

All method delta descriptions have the same format. They have four parts which are outlined as follows:

- The **name** identifies the method delta.
- The summary of the **difference** between two AST nodes for which this method delta is identified.

- A **Used for** clause that lists the safe software changes for which this method delta is used.
- If necessary, some further **description** of the method delta or the context in which it is observed.

A.4.1 Add Argument

The argument list of a `MethodInvocation` node contains an additional entry.

Used for: *Add Argument* refactoring participant.

A.4.2 Add Leftmost Invocation

A `MethodInvocation` node has an additional expression to qualify the method invocation target.

Used for: *Move Method* refactoring participant.

This method delta is typically observed when the calling method had invoked a method of its own class and that called method was moved away.

A.4.3 Add Parameter

The parameter list of a `MethodDeclaration` node contains an additional parameter declaration.

Used for: *Add Parameter* refactoring.

A.4.4 Change Argument

An entry of a `MethodInvocation` node's argument list changes.

Used for: *Change Argument* change participant.

At the time of writing, this is the only method delta that can result in the identification of a change participant. The way the argument changes is not further specified for the detection of this method delta.

A.4.5 Change Parameter Type

A `SingleVariableDeclaration` of a `MethodDeclaration` node's parameter list changes its type.

Used for: *Reconcile Differences* refactoring.

A.4.6 Change Return Type

The return type of a `MethodDeclaration` node changes.

Used for: *Reconcile Differences* refactoring.

A.4.7 Change Visibility

The visibility modifier of a `MethodDeclaration` node changes.

Used for: *Expose Method* refactoring.

This method delta could also be used for *Hide Method*, which does not appear in this dissertation.

A.4.8 Delegate Constructor

The statements of a `Block` node that represents a constructor's body are replaced by a `ConstructorInvocation`.

Used for: *Delegate Constructor* refactoring participant.

A.4.9 Delegate Method

The statements in a `Block` node that represents a method's body are replaced by a `MethodInvocation`.

Used for: *Delegate Method* refactoring participant.

A.4.10 Extract Method

Some statements in a method's body are replaced by a `MethodInvocation`.

Used for: *Extract Method* refactoring.

To identify this method delta, the delta mapping algorithm also checks if the replaced statements are found in the called method.

A.4.11 Move Method

A `MethodDeclaration` is moved to a different location.

Used for: *Move Method* refactoring.

This method delta applies if the method is moved to a different class or if its defining class is moved to another package. For detecting this delta, the method's body is not touched.

A.4.12 Other Body

Any change in a method's body that cannot be identified as one of the other method deltas.

Used for: *Change Argument* change participant.

The Refactoring Finder component must be able to see if a called method changes in a way that allows the identification of a change participant. For such a called method, the *Other Body* delta is reported together with other identified method deltas. This method delta implements the *Null Object* design pattern [MRB97].

A.4.13 Remove Argument

The argument list of a `MethodInvocation` node misses an former entry.

Used for: *Remove Argument* refactoring participant.

A.4.14 Remove Leftmost Invocation

A `MethodInvocation` node misses a former expression to qualify the method invocation target.

Used for: *Move Method* refactoring participant.

Typically, this method delta is observed when the calling method has invoked a method of its own class and that called method has moved to here from a different class.

A.4.15 Remove Parameter

The parameter list of a `MethodDeclaration` node misses a former parameter declaration.

Used for: *Remove Parameter* refactoring.

A.4.16 Rename Parameter

A `SingleVariableDeclaration` of a `MethodDeclaration` node's parameter list changes its name.

Used for: *Rename Parameter* refactoring.

To identify this method delta, the delta mapping algorithm also checks if the parameter name is also changed throughout the method body.

A.4.17 Replace Constructor with Method Call

A `ConstructorInvocation` is replaced by a `MethodInvocation`.

Used for: *Replace Constructor with Factory Method* refactoring.

A.4.18 Replace Throw with Assignment

A `ThrowStatement` is replaced by an `Assignment`.

Used for: *Replace Exception with Error Code* refactoring.

A.4.19 Replace Throw with Return

A `ThrowStatement` is replaced by a `ReturnStatement`.

Used for: *Replace Exception with Error Code* refactoring.

Appendix B

List of Design Principles

This chapter lists some principles of object-oriented design that are used in the text. These principles "help developers eliminate design smells and build the best designs for the current set of features." [Mar03, p. 86]

B.1 Single–Responsibilty Principle (SRP)

Also known as *cohesion*, which goes back to [DeM79] and [PJ88]. It is defined as "the degree to which something models a single abstraction, localizing only features and responsibilities related to that abstraction" [FE95]. Martin [Mar03] defines it as follows: "A class should have only one reason to change."

Despite its simplicity, the SRP is hard to get right. Martin's view on responsibility as "a reason for change" helps in the practical application of the principle. If a new test drives a change in a class for which it was not intended then this might be a violation of the SRP. The reason for the change might be a new responsibility that needs a new abstraction (e.g. a class or interface) in the application design.

B.2 Open Closed Principle (OCP)

Bertrand Meyer [Mey97] coined the well-known Open-Closed principle. It says that software entities (classes, modules, function, etc.) should be open for extension but closed for modification.

A single change in production code might induce a cascade of changes in dependent modules, which is a smell of bad design [Mar03, p. 99]. The OCP advices to redesign the application so that future changes of the same kind do not have the same effect. To achieve this goal, a programmer typically introduces a new abstraction.

The design closure determines against which changes a design is closed for modification but open for extension. In agile development, design evolves over time; it

is not planned up-front. In other words, the design closure is extended each time the implementation of a new feature does not fit into the existing design closure.

The remainder of this section discusses how software changes impact upon the design closure.

B.2.1 Closure-Preserving Changes

A closure-preserving change lies within the design closure of the existing system. This means in some earlier point in time, a software change of the same kind was implemented, which resulted in the creation of the appropriate abstractions.

Figure B.1 shows an example of a closure-preserving change. Interface IX is an abstraction for some functionality in class V. Now the addition in class W can rely completely on the already existing abstraction in interface IX.

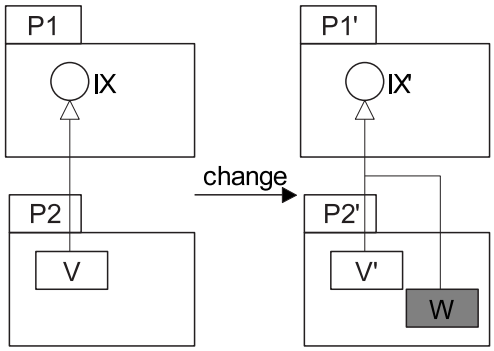

Figure B.1: *Example of a closure-preserving change.* The new class W fits into the already existing abstraction that is implemented by interface IX.

Closure-preserving changes are commonly observed in TDD. In fact the application under development evolves to its own framework. Such closure-preserving changes demonstrate the beneficial use of that framework.

B.2.2 Closure-Violating Changes

Closure-violating changes do not fit into the set of abstractions offered by the existing application. The design closure consists of all the abstractions that were introduced to implement new features in the application. Whenever a change of a new kind is implemented, the programmer needs to add a new abstraction and thus extend the existing design closure. Closure-violating changes implement a new feature but fail to introduce a new abstraction by hacking a “solution” which does not fit the existing design.

Figure B.2 shows an example of a closure-violating change. Interface IX is an abstraction for the functionality in class V. Now class W is part of some new functionality for which no existing abstraction can be used. To implement the required functionality, changes in existing classes are performed without creating an appropriate abstraction.

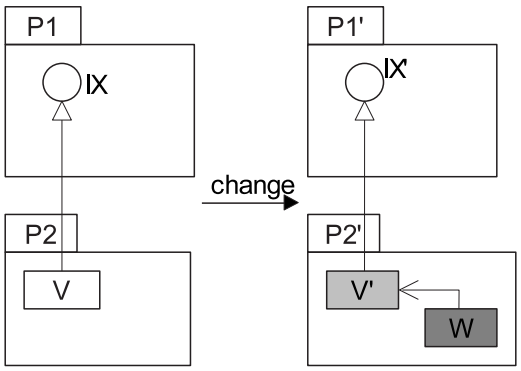

Figure B.2: *Example of a closure-violating change.* New functionality in class W doesn't fit into the existing abstractions. In order to integrate with the application, some existing class V has to be modified.

Agile methodologies request that whenever a programmer introduces a change of a new kind into the system, he/she has to create the appropriate abstractions at the same time. Failing to do so introduces design smells in the design. The purpose of applying design principles is to remove design smells. Deferring the removal of design smells to a later point in time is not valid because it reduces the ability to be agile.

B.2.3 Closure-Extending Changes

In the same fashion as closure-violating changes, this kind of change does not fit into the existing design closure. However, instead of just violating the existing design closure, this change extends it.

Figure B.3 shows an example of a closure-extending change. Interface IX is an abstraction for the functionality in class V. Class W is part of some new functionality for which no existing abstraction can be used. Thus the new interface IY is the abstraction necessary to implement the new functionality. Existing classes have to be changed as well but now the new abstraction extends the design closure.

In a strict Test-Driven Design (TDD) sense, a developer would first implement the new feature using the existing design closure. At first this looks like a closure-violating change. As discussed above, this results in design smells because the additions do not fit into the existing design closure. After detecting these smells, the developer applies the design principles in order to return to a smell-free state.

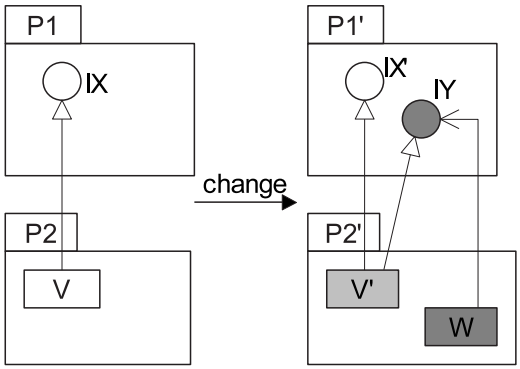

Figure B.3: *Example of a closure-extending change.* New functionality in class W fits into a new abstraction implemented by interface IY. Some existing class V might need to be adapted.

A developer which is not following TDD in the strict sense could apply the principles first to prepare the system for the new feature. He simply has to ensure that there are no smells left after having finished the software change.

Whether or not the software change goes through the closure-violating state, the application of the design principles introduces new abstractions into the system that are used by the newly implemented feature. It is important to note that the newly introduced abstractions have to be used right away unless they are a smell for “Needless Complexity”.

Closure-extending changes are commonly observed in TDD. In fact, as seen above, an application under development evolves to its own framework. Such closure-extending changes are the means for this framework evolution.

Appendix C

Glossary

This chapter describes some commonly used terms that are not defined elsewhere in the document.

Abstract class A class whose primary purpose is to define an interface. An abstract class defers some or all of its implementation to subclasses. An abstract class cannot be instantiated. [GHJV95]

Ancestor (type of a given type) Any type from which the given type is directly or indirectly derived via inheritance. [FE95]

Application Any collection of classes that work together to provide related functionality to the end user. *Synonym:* Program. [FE95]

Committer A developer who has commit rights to a source repository (typically of an Open Source project).

Delegation An implementation mechanism in which an object forwards or *delegates* a request to another object. The delegate carries out the request on behalf of the original object. [GHJV95]

Descendant (type of a given type) Any type that inherits, either directly or indirectly, from the given type. [FE95]

Framework A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. [GHJV95]

Implementor A concrete realization of the contract declared by an interface. [JBR99]

Interface A collection of operations that are used to specify a service of a class or component. [JBR99]

Method call graph A graph representing calls between methods in a given program. [MNGL98]

Program *see* Application.

Relationship A relationship is any logical static connection between two or more things. [FE95]

Software System Any application consisting of software, hardware, documentation (a.k.a. paperware), and roles played by people (a.k.a. wetware). [FE95]

Type The declaration of the interface of any set of instances that conform to this common protocol. [FE95]

References

[ACL99] G. Antoniol, G. Canfora, and A. de Lucia. Maintaining Traceability During Object-Oriented Software Evolution: A Case Study. In *Proc. IEEE Intl. Conf. on Software Maintenance (ICSM)*, Oxford, England, August/September 1999.

[AGM87] K.H. An, D.A. Gustafson, and A.C Melton. A Model for Software Maintenance. In *Proc. IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 57–62, 1987.

[All01] Agile Alliance. Agile Manifesto. Online at http://www.agilemanifesto.org, 2001.

[AM03] Alain Abran and James W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - Trial Version (Version 1.0)*. IEEE Computer Society, 2003. Online at http://www.swebok.org.

[Ams01] Jim Amsden. Levels of Integration — Five ways you can integrate with the Eclipse Platform. Eclipse Corner Articles, March 2001. Online at http://www.eclipse.org/articles.

[Arm04] Phillip G. Armour. Beware of Counting LOC. *Communications of the ACM*, 47(3):21–24, March 2004.

[Ast03] Dave Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, Upper Saddle River, NJ, 2003.

[AWSR03] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New Directions on Agile Methods: A Comparative Analysis. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 244–254, Portland, OR, May 2003.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik — Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, Heidelberg, Germany, 1998.

[BB03] Kent Beck and Barry Boehm. Agility through Discipline: A Debate. *IEEE Computer*, 36(6):44–46, June 2003.

[BBC+96] Victor Basili, Lionel Briand, Steven Condon, Yong-Mi Kim, Walcélio L. Melo, and Jon D. Valett. Understanding and Predicting the Process of Software Maintenance Releases. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, Berlin, Germany, 1996.

[BBS02] K. Bassin, S. Biyani, and P. Santhanam. Metrics to Evaluate Vendor-Developed Software Based on Test Case Execution Results. *IBM Systems Journal*, 41(1):13–30, 2002.

[BC01] Piergiuliano Bossi and Francesco Cirillo. Repo Margining System: Applying XP in the Financial Industry. In *Proc. 2nd Intl. Conf. on eXtreme Programming and Flexible Processes in Software Engineering (XP2001)*, Cagliari, Italy, 2001.

[BCR02] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. In John J. Marciniak, editor, *Encyclopedia of Software Engineering, 2 Volume Set*. Wiley, 2002.

[Bec00] Kent Beck. *eXtreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[Bec03] Kent Beck. *Test-Driven Development by Example*. Addison-Wesley, Boston, MA, 2003.

[Bei93] Boris Beizer. *Software Testing Techniques*. Van Noshand, New York, 1993.

[BFL+95] Sergio Bandinelli, Alfonso Fuggetta, Luigi Lavazza, Maurizio Loi, and Gian Pietro Picco. Modeling and Improving an Industrial Software Process. *IEEE Transactions on Software Engineering*, 21(5):440–454, 1995.

[BG98] Kent Beck and Erich Gamma. Test-Infected: Programmers Love Writing Tests. *Java Report*, 3(7), 1998. Online at http://www.junit.org.

[BGL+96] Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sørumgård, and Marvin V. Zelkowitz. The Empirical Investigation of Perspective- Based Reading. *Empirical Software Engineering: An International Journal*, 1(2):133–164, 1996.

[BH93] Samuel Bates and Susan Horwitz. Incremental Program Testing Using Program Dependence Graphs. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 384–396, January 1993.

[Bin98] David Binkley. The Application of Program Slicing to Regression Testing. *Information and Software Technology*, 40(11–12):583–594, 1998.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[BK95] J. M. Bieman and B. K. Kang. Cohesion and Reuse in an Object-Oriented System. In *Proc. ACM Symposium on Software Reusability*, April 1995.

[BKS98] Kathryn A. Bassin, Theresa Kratschmer, and P. Santhanam. Evaluating Software Development Objectively. *IEEE Software*, 15(6):66–74, November/December 1998.

[BL76] Laszlo A. Belady and M. M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(1):225–252, 1976.

[BMK02] Mark Butcher, Hilora Munro, and Theresa Kratschmer. Improving Software Testing via ODC: Three Case Studies. *IBM Systems Journal*, 41(1):31–44, 2002.

[BMPZ02] Victor R. Basili, Frank E. McGarry, Rose Pajerski, and Marvin V. Zelkowitz. Lessons Learned from 25 Years of Process Improvement: The Rise and Fall of the NASA Software Engineering Laboratory. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, May 2002.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, West Sussex, England, 1996.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, Upper Saddle River, NJ, 1981.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, CA, second edition, 1994.

[BR00] Keith H. Bennett and Vaclav T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 73–87, 2000.

[Bro95] Fred Brooks. *The Mythical Man-Month*. Addison-Wesley, Boston, 20th anniversary edition, 1995.

[Bro96] Kyle Brown. Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. Master's thesis, University of Illinois, 1996.

[BSW+03] James M. Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In *Proc. Ninth International Software Metrics Symposium (METRICS'03)*, Sydney, Australia, September 2003. IEEE.

[BT03] Barry Boehm and Richard Turner. *Balancing Agility and Discipline — A Guide for the Perplexed*. Addison-Wesley, Boston, MA, 2003.

[BW84] Victor R. Basili and David Weiss. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, 10(3):728–738, November 1984.

[CBC+92] R. Chillarege, I. S. Bhandari, J. K. Chaar, D. S. Moebus M. J. Halliday, B. K. Ray, and M.-Y. Wong. Orthogonal Defect Classification: A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

[CDM02] João W. Cangussu, Raymond A. DeCarlo, and Aditya P. Mathur. A Formal Model of the Software Test Process. *IEEE Transactions on Software Engineering*, 28(8):782–796, August 2002.

[Ced93] Per Cederqvist. *Version Management with CVS*, 1993. Online at http://www.cvshome.org.

[CHK+97] Jim Coplien, Luke Hohmann, Norm Kerth, John Rae-Grant, and Eileen Strider. Panel: Changing the Engine of the Car? While Driving 60 Miles an Hour! In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 158–161. ACM Press, 1997.

[CK94] S. R. Chidamber and C.F. Kemerer. A metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[CMM02] CMMI Development Team. *Capability Maturity Model Integration V1.1*. Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, 2002.

[Coc98] Alistair Cockburn. *Surviving Object-Oriented Projects: A Manager's Guide*. Addison-Wesley, Reading, MA, 1998.

[Coc00] Alistair Cockburn. Characterizing people as non-linear, first-order components in software development. In *Proc. 4th Intl. Multi-Conference on Systems, Cybernetics and Informatics*, Orlando, Florida, 2000.

[Coc01] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2001.

[Coc02] Alistair Cockburn. Learning from Agile Software Development. *Crosstalk*, October 2002.

[Col03] Michael L. Collard. An Infrastructure to Support Meta-Differencing and Refactoring of Source Code. In *Proc. Intl. Conf. on Automated Software Engineering (ASE'03)*, Edinburgh, UK, September 2003. IEEE.

[Con68] Melvin E. Conway. How do Committees Invent. *Datamation*, 14, April 1968.

[Cop97] Jim O. Coplien. Idioms and Patterns as Architectural Literature. *IEEE Software*, 14(1):36–42, January 1997.

[Cop99] James O. Coplien. Reevaluating the Architectural Metaphor: Toward Piecemeal Growth. *IEEE Software*, 16(5):40–44, September/October 1999.

[Cox90] Brad Cox. There is a Silver Bullet. *BYTE Magazine*, Oktober 1990.

[CS95a] N. I. Churcher and M. J. Shepperd. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 21(3):263–265, March 1995.

[CS95b] James O. Coplien and Douglas C. Schmidt. *Patterns Languages of Program Design*. Addison-Wesley, 1995.

[DB98] Allen H. Dutoit and Bernd Bruegge. Communication Metrics for Software Development. *IEEE Transactions on Software Engineering*, 24(8), August 1998.

[DD99] Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. FAMOOS project, 1999.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding Refactorings via Change Metrics. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 166–177. ACM Press, 2000.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press, Prentice Hall, Englewood Cliff, NJ, 1979.

[DeM82a] Tom DeMarco. *Controlling Software Projects*. Yourdon Press, Prentice Hall, Englewood Cliff, NJ, 1982.

[Dem82b] W.E. Deming. *Out of the Crisis*. MIT Center Advanced Eng. Study, Cambridge, MA, 1982.

[DM02] Arie van Deursen and Leon Moonen. The Video Store Revisited — Thoughts on Refactoring and Testing. In *Proc. 3rd Intl. Conf. on eXtreme Programming and Agile Processes in Software Engineering (XP2002)*, pages 71–76, Alghero, Sardinia, Italy, 2002.

[DMBK01] Arie van Deursen, Leon Moonen, A. van den Bergh, and G. Kok. Refactoring Test Code. In *Proc. 2nd Intl. Conf. on eXtreme Programming and Flexible Processes in Software Engineering (XP2001)*, 2001.

[DMW01] Serge Demeyer, Tom Mens, and Michel Wermelinger. Towards a Software Evolution Benchmark. In *Proc. Intl. Workshop on Principles of Software Evolution*, Vienna, Austria, September 2001.

[DRW02] Alastair Dunsmore, Marc Roper, and Murray Wood. Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, Orlando, FL, May 2002.

[Dum00] Reiner Dumke. *Software Engineering*. Vieweg, Braunschweig, Germany, 2 edition, 2000.

[Ebe95] Christof Ebert. Complexity Traces: An Instrument for Software Project Management. In Norman Fenton, Robin Whitty, and Yoshinori Iizuka, editors, *Software Quality: Assurance and Measurement – A Worldwide Perspective*, pages 166–176. Internation Thomson Computer Press, 1995.

[EC02] Khaled el Emam and D. Card, editors. *ISO/IEC Standard 15939 — Software Measurement Process*. International Organization for Standardization, 2002.

[Edw03a] Stephen H. Edwards. Teaching Software Testing: Automatic Grading Meets Test-First Coding. In *Addendum to Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[Edw03b] Stephen H. Edwards. Using Test-Driven Development in the Classroom: Providing Students with Concrete Feedback on Performance. In *Proc. Intl. Conference on Education and Information Systems: Technologies and Applications (EISTA'03)*, August 2003.

[EGK+01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, and Audris Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27(1):1–12, January 2001.

[EGK+02] Stephen G. Eick, Todd L. Graves, Alan F. Karr, Audris Mockus, and P. Schuster. Visualizing Software Changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, April 2002.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3):210–224, March 2003.

[ER96] Nancy. S. Eickelmann and Debra. J. Richardson. What Makes One Software Architecture More Testable Than Another? In Alexander L. Wolf, editor, *Proc. Second Intl. Software Architecture Workshop (ISAW-2) at SIGSOFT'96*, San Francisco, CA, October 1996.

[ES02] Amr Elssamadisy and Gregory Schalliol. Recognizing and Responding to "Bad Smells" in Extreme Programming. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, Orlando, FL, May 2002.

[Fag76] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–210, 1976.

[Fag86] Michael E. Fagan. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, July 1986.

[Fag01] Michael E. Fagan. A History of Software Inspections. In M. Broy and E. Denert, editors, *Proc. sd&m Conference 2001, Software Pioneers*, pages 215–225. Springer, 2001.

[FE95] Donald G. Firesmith and Edward M. Eykholt. *Dictionary of Object Technology — The Definitive Desk Reference*. SIGS Books, New York, 1995.

[Fen94] Norman Fenton. Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FP96] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Publishing, second edition, 1996.

[GB99] Erich Gamma and Kent Beck. Junit: A Cook's Tour. *Java Report*, May 1999.

[GB04] Erich Gamma and Kent Beck. *Contributing to Eclipse – Principles, Patterns, and Plug-ins*. Addison-Wesley, 2004.

[GG93] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, Boston, MA, 1993.

[GHJ98] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proc. IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 190–198, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Gil96] Tom Gilb. Level 6: Why We Can't Get There From Here. *IEEE Software*, 13(1):97–98,103, January 1996.

[GJKT97] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. Software Evolution Observations Based on Product Release History. In *Proc. IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 160–166, 1997.

[GMR+02] Michael Gnatz, Frank Marschall, Gerhard Popp Andreas Rausch, Maura Rodenberg-Ruiz, and Wolfgang Schwerin. Proc. 1st Workshop on Software Development Patterns (sdpp'02) held at OOPSLA'02. Technical Report TUM-I0213, Technical University of Munich, December 2002.

[Gre91] Jim Grey. *The Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufman, San Mateo, CA, 1991.

[Hal77] Maurice Halstead. *Elements of Software Science*. Elsevier, Amsterdam, Netherlands, 1977.

[HF82] Peter G. Hamer and Gillian D. Frewin. M.H. Halstead's Software Science — A Critical Examination. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 197–206, 1982.

[HG04] M. Hagen and V. Gruhn. PROPEL — Eine Sprache zur Beschreibung von Process Patterns. In *Proc. of Modellierung'04*, LNI. Gesellschaft für Informatik, 2004.

[Hig00] Jim Highsmith. *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems*. Dorset House, 2000.

[Hig02] Jim Highsmith. *Agile Software Development Ecosystems*. Addison-Wesley, 2002.

[HL03] Erik Hatcher and Steve Loughran. *Java Development with Ant*. Manning, Greenwich, CT, 2003.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[HSW91] Watts S. Humphrey, Terry R. Snyder, and Ronald R. Willis. Software Process Improvement at Hughes Aircraft. *IEEE Software*, 8(4):11–23, July/August 1991.

[Hum89] Watts S. Humphrey. *Managing the Software Process*. SEI Series in Software Engineering. Addison-Wesley, Reading MA, 1989.

[Hum95] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Boston, MA, 1995.

[Hum97] Watts S. Humphrey. *Introduction to the Personal Software Process*. SEI Series in Software Engineering. Addison-Wesley, 1997.

[Hum98a] Watts S. Humphrey. Three Dimensions of Process Improvement Part 1: Process Maturity. *Crosstalk*, February 1998.

[Hum98b] Watts S. Humphrey. Three Dimensions of Process Improvement Part 2: The Personal Process. *Crosstalk*, March 1998.

[Hum98c] Watts S. Humphrey. Three Dimensions of Process Improvement Part 3: The Team Process. *Crosstalk*, April 1998.

[ISO98] Information Technology — Software Process Assessment, 1998. ISO/IEC TR 15504:1998.

[ISO03] Information Technology — Process Assessment — Part 2: Performing an Assessment, 2003. ISO/IEC 15504-2:2003.

[Jac03] Zygmunt Jackowski. Metamodel of a System Development Method. Agile Alliance Articles, September 2003. Online at http://www.agilealliance.com.

[JAH00] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JKA+03] Philip M. Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, and William E.J. Doane. Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 641–646, Portland, OR, May 2003.

[Jon00] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley, 2000.

[Ker01] Norman Kerth. *Project Retrospectives — A Handbook for Team Reviews*. Dorset House, New York, 2001.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004. To be published.

[KFN93] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing Computer Software*. Van Nostrand Reinhold, New York, second edition, 1993.

[Kit96] Barbara Ann Kitchenham. Evaluating Software Engineering Methods and Tool Part 1: The Evaluation Context and Evaluation Methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–15, January 1996.

[KPM01] S. H. Kan, J. Parrish, and D. Manlove. In-Process Metrics for Software Testing. *IBM Systems Journal*, 40(1):220–241, 2001.

[KPP95] Barbara Kitchenham, Lesley M. Pickard, and Shari Lawrence Pfleeger. Case Studies for Method and Tool Evaluation. *IEEE Software*, 12(4):52–62, July 1995.

[KPP+02] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[KS99] Chris F. Kemerer and Sandra Slaughter. An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July/August 1999.

[KST+86] Joseph K. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. Software Complexity Measurement. *Communications of the ACM*, 29(11):1044–1050, November 1986.

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

[LF03] Johannes Link and Peter Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann, 2003.

[LP80] M. M. Lehman and J. Patterson. Programs, Life Cycles and Laws of Software Evolution. In *Proc. IEEE Special Issue on Software Engineering*, volume 68, pages 1060–1076, September 1980.

[Man97] Benoit Mandelbrot, editor. *Fractals and Scaling in Finance*. Springer, Berlin, 1997.

[Mar03] Robert Cecil Martin. *Agile Software Development — Principles, Patterns, and Pracitces*. Prentice-Hall, Upper Saddle River, NJ, 2003.

[Mat00] Michael Mattsson. *Evolution and Composition of Object–Oriented Frameworks*. PhD thesis, University of Karlskrona/Ronneby, 2000.

[May45] Elton Mayo. *The Social Problems of an Industrial Civilization*. Harvard University Press, Cambridge, MA, 1945.

[MC02] J. McGarry and D. Card. *Practical Software Measurement*. Addison-Wesley, Boston, MA, 2002.

[McB03] Pete McBreen. *Questioning Extreme Programming*. Addison-Wesley, Boston, MA, 2003.

[McC93] Steve McConnell. *Code Complete – A Practical Handbook of Software Construction*. Microsoft Press, 1993.

[McF96] Bob McFeeley. IDEAL: A User´s Guide for Software Process Improvement. Technical Report CMU/SEI-96-HB-001, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, 1996.

[MDB+03] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current Research and Future Trends. In *Proc. 3rd Workshop on Language Descriptions, Tools and Applications (LDTA'03)*, Warsaw, Poland, April 2003. Elsevier. To be published in ENTCS.

[MDJ+97] Lehman M., Perry D., Ramil J., Turski W., and Wernick P. Metrics and Laws of Software Evolution — The Nineties View. In *Proc. Metrics'97 Symposium*, Albuquerque, NM, 1997.

[MEGK99] Audris Mockus, Stephen G. Eick, Todd L. Graves, and Alan F. Karr. On Measurement and Analysis of Software Changes. Technical report, National Institute of Statistical Science, 1999.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1997.

[MH02] Audris Mockus and James D. Herbsleb. Expertise Browser: A Quantitative Approach to Identifying Expertise. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, May 2002.

[MHT00] Guido Malpohl, James J. Hunt, and Walter F. Tichy. Renaming Detection. In *Proc. Intl. Conf. on Automated Software Engineering (ASE'00)*, Grenoble, France, September 2000. IEEE.

[Mil88] Harlan Mills. *Software Productivity*. Dorset House, New York, 1988.

[MM85] Webb Miller and Eugene W. Myers. A File Comparison Program. *Software Practice and Experience*, 15(11):1025–1040, November 1985.

[MNGL98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An Empirical Study of Static Call Graph Extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, April 1998.

[Moo02] Leon Moonen. *Exploring Software Systems*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, Netherlands, December 2002.

[MPHN02] Lars Mathiassen, Jan Pries-Heje, and Ojelanki Ngwenyama. *Improving Software Organizations*. Addison-Wesley, Boston, MA, 2002.

[MRB97] Robert C. Martin, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[Mug03] Rick Mugridge. Challenges in Teaching Test-Driven Development. In *Proc. 4th Intl. Conf. on eXtreme Programming and Agile Processes in Software Engineering (XP2003)*, Genova, Italy, May 2003.

[MW03] E. Michael Maximilien and Laurie Williams. Assessing Test-Driven Development at IBM. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 564–569, Portland, OR, May 2003.

[OJ90] William F. Opdyke and Ralph Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving OO Systems. In *Proc. SOOPPA Conference*, September 1990. In SIGPLAN Notices.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Orr02] Ken Orr. CMM Versus Agile Development: Religious Wars and Software Development. *Agile Project Management*, 3(7), 2002.

[Pau02] Mark C. Paulk. Agile Methodologies and Process Discipline. *Crosstalk*, October 2002.

[PB02] Claudia Pons and Gabriel Baum. Reasoning About the Correctness of Software Development Process. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, Orlando, FL, May 2002.

[Pip02] Jens Uwe Pipka. Refactoring in a Test First-World. In *Proc. 3rd Intl. Conf. on eXtreme Programming and Agile Processes in Software Engineering (XP2002)*, Alghero, Sardinia, Italy, May 2002.

[Pip03] Jens Uwe Pipka. Development Upside Down: Following the Test First Trail. Practitioners Report at ECOOP'03, 2003. Online at http://www.daedalos.de/DE/DOWNLOAD/presentations/development_upside_down.pdf.

[PJ88] Meilir Page-Jones. *The Practical Guide to Structured Design*. Yourdon Press Computing Series, Englewood Cliff, NJ, second edition, 1988.

[PM90] Shari L. Pfleeger and C. L. McGowan. Software Metrics in a Process Maturity Framework. *Journal of Systems and Software*, 12(3):255–261, 1990.

[Pow98] A.L. Powell. A Literature Review on the Quantification of Software Change. Technical Report YCS-98-305, University of York, Department of Computer Science, 1998.

[PS90] Adam A. Porter and Richard W. Selby. Empirically Guided Software Development Using Metric-Based Classification Trees. *IEEE Software*, 7(2):46–54, 1990.

[PV94] A.A. Porter and L.G. Votta. An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 103–112, 1994.

[PWC95] M.C. Paulk, C.V. Weber, and B. Curtis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, Reading, MA, 1995.

[Ree92] Jack Reeves. What is Software Design? *C++ Journal*, 2(2), 1992.

[RH94] Gregg Rothermel and Mary Jean Harrold. Selecting Tests and Identifying Test Coverage Requirements for Modified Software. In *Proc. Intl. Symposium on Software Testing and Analysis*, pages 169–184. ACM Press, 1994.

[SDF+03] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehm, John Kellerman, and Pat McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003.

[Sea99] Carolyn B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July/August 1999.

[SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 74–83, Portland, OR, May 2003.

[Sim01] Frank Simon. *Meßwertbasierte Qualitätssicherung*. PhD thesis, Technische Universität Cottbus, Germany, 2001.

[SKN+99] Norman Schneidewind, Barbara Kitchenham, Frank Niessink, Janice Singer, Anneliese von Mayrhauser, and Hongji Yang. Panel: Software Maintenance is Nothing More Than Another Form of Development. In *Proc. IEEE Intl. Conf. on Software Maintenance (ICSM)*. IEEE Computer Society, 1999.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2), 1974.

[Som92] Ian Sommerville. *Software Engineering*. Addison-Wesley, fourth edition, 1992.

[Spi03] Diomidis Spinellis. *Code Reading*. Addison-Wesley, 2003.

[SS03] Jason McC. Smith and David Stotts. SPQR: Flexible Automated Design Pattern Extraction from Source Code. Technical Report TR03-016, Department of Computer Science, University of North Carolina, Chapel Hill, NC, May 2003.

[ST02] Amitabh Srivastava and Jay Thiagarajan. Effectively Prioritizing Tests in Development Environment. In *Proc. Intl. Symposium on Software Testing and Analysis*, pages 97–106. ACM Press, 2002.

[Sta03] Marne Staples. Test Effectivness. Microsoft Research Faculty Summit, 2003.

[Ste01] Lukas Steiger. Recovering the Evolution of Object-Oriented Software Systems Using a Flexible Query Engine. Master's thesis, Universität Bern, Switzerland, 2001.

[Swa76] E.B. Swanson. The Dimensions of Maintenance. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, pages 492–497, 1976.

[Tic98] Walter F. Tichy. Should Computer Scientists Experiment More? *IEEE Computer*, 31(5):32–40, May 1998.

[Wak03] William C. Wake. *Refactoring Workbook*. Addison-Wesley, Boston, MA, 2003.

[WB85] David M. Weiss and Victor R. Basili. Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory. *IEEE Transactions on Software Engineering*, 11(2):157–168, 1985.

[WBP+02] David M. Weiss, David Bennett, John Y. Payseur, Pat Tendick, and Ping Zhang. Goal-Oriented Software Assessment. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, Orlando, FL, May 2002.

[WC03] Laurie Williams and Alistair Cockburn. Agile Software Development: It's about Feedback and Change. *IEEE Computer*, 36(6):39–43, June 2003.

[Wei71] Gerald M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.

[Wei93] Gerald M. Weinberg. *Quality Software Management — First-Order Measurement*, volume 2. Dorset House, New York, 1993.

[Wel93] E.F. Weller. Lessons Learned from Three Years of Inspecting Data. *IEEE Software*, 10(5):38–45, September 1993.

[WG01] Christian Wege and Frank Gerhardt. Learn XP: Host a Boot Camp. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*, pages 489–500. Addison-Wesley, 2001. Proc. XP2000 Conference.

[Wil00] Laurie Williams. *The Collaborative Software Process*. PhD thesis, University of Utah, 2000.

[Wil01] Dwight Wilson. Teaching XP: A Case Study. In *Proc. XP Universe*, Raleigh, NC, July 2001.

[WK02] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley, 2002.

[WL01] Christian Wege and Martin Lippert. Diagnosing Evolution in Test-Infected Code. In *Extreme Programming Perspectives*, pages 153–166. Addison-Wesley, 2001. Proc. XP2001 Conference.

[Won02] Yuk Kuen Wong. Use of Software Inspection Inputs in Practice. In *Proc. Intl. Conf. on Software Engineering (ICSE)*, Orlando, FL, May 2002.

[WSP+02] C. Williams, H. Sluiman, D. Pitcher, M. Slavescu, J. Spratley, M. Brodhun, J. McLean, C. Rankin, and K. Rosengren. The STCL Test Tools Architecture. *IBM Systems Journal*, 41(1):74–88, 2002.

[YC79] Ed Yourdon and Larry L. Constantine. *Structured Design*. Prentice Hall, 1979.

[YTFB89] M. Young, R. Taylor, K. Forester, and D. Brodbeck. Integrated Concurrency Analysis in a Software Development Environment. In *Proc. ACM SIGSOFT'89 3rd Symposium on Software Testing, Analysis, and Verification*, pages 200–209. ACM Press, 1989.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, December 1997.

[ZW98] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental Models for Validating Technology. *IEEE Computer*, 31(5):23–31, May 1998.

Online References

[URL:AntDev] http://marc.theaimsgroup.com/?l=ant-dev&m=105880552631305&w=2

[URL:Ant] http://ant.apache.org

[URL:Apache] http://www.apache.org

[URL:BYacc] http://troi.lincom-asg.com/rjamison/byacc

[URL:CVS] http://www.cvshome.org

[URL:Clover] http://www.thecortex.net/clover

[URL:CmEclipse] http://www.lucas.lth.se/cm/cmeclipse.shtml

[URL:Dynabook] http://www.computer.org/SEweb/Dynabook

[URL:EclipeBugs] https://bugs.eclipse.org/bugs

[URL:Freese] http://tammofreese.de/Research.html

[URL:Hansel] http://hansel.sourceforge.net

[URL:Hillside] http://hillside.net/patterns/onlinepatterncatalog.htm

[URL:J2EEPatterns] http://java.sun.com/blueprints/patterns/catalog.html

[URL:JFlex] http://www.jflex.de

[URL:JMetra] http://www.jmetra.com

[URL:JUnit] http://www.junit.org

[URL:JXCL] http://jxcl.sourceforge.net

[URL:Maven] `http://maven.apache.org`

[URL:NoUnit] `http://nounit.sourceforge.net`

[URL:ODC] `http://www.research.ibm.com/softeng`

[URL:QDox] `http://qdox.codehaus.org`

[URL:SPICE] `http://www.sqi.gu.edu.au/spice`

[URL:SVC] `http://subversion.tigris.org`

[URL:StatCvs] `http://statcvs.sourceforge.net`

[URL:ThoughtWorks] `http://www.thoughtworks.com`

[URL:Tomcat] `http://jakarta.apache.org/tomcat`

[URL:WikiUT] `http://c2.com/cgi/wiki?UnitTest`

[URL:YAGNI] `http://c2.com/cgi/wiki?YouArentGonnaNeedIt`