MSc Distributed Computer Systems Engineering

Department of Electronic & Computer Engineering

Brunel University

A TestSuite for Extreme Programming

Dirk Kaller

March 2001

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

MSc Distributed Computer Systems Engineering

Department of Electronic & Computer Engineering

Brunel University

A TestSuite for Extreme Programming

Dirk Kaller

Dr. Ian D. Dear

March 2001

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

Nur der ist tot, der vergessen wird.

... in memory to my mother.

To Gabi

Abstract

Software can never be overtested, and testing usually starts too late. When testing does begin, system or integration testing is the norm, with formal testing and quality assurance efforts typically starting at the last minute – when the software is nearing release. Inevitably, pressure builds on testers to rush through the testing process. Needless to say, this does not encourage thoroughness. At the same time, the development team is urged to fix defects as quickly as possible. This, too, does not promote careful attention to detail.

Extreme Programming (XP), a new lightweight process for developing software, was conceived to meet the demands for better-quality software.
Logically testing is a key feature of XP. Extreme Programming is full of interesting ideas regarding to testing that encourage one to think – for example, how about "Test and then code"?

In this thesis, an open source testing framework called JUnit should be integrated into a GUI based TestSuite which makes it easier to run unit tests, more flexible and more powerful. Additionally the TestSuite must be easy to expand for acceptance and load/stress tests. The requirements for the TestSuite should be delivered by the projects at DaimlerChrysler, which are using JUnit at the moment.

Contents

1 Introduction

Today three of the most pressing issues in information technology are

- How can quality software be produced within ever-shrinking schedules and with minimal resources (e.g. budget)?
- How do software development teams deliver more and more functionality to business clients quickly?
- How do software development teams keep up with near-continous change?

Our businesses demand both speedy software delivery and the flexibility to change that software rapidly – with the requirement for low defects thrown in on top. Traditional software development models are not geared to the rates of change and speed necessary today. Extreme Programming (XP) is one of a number of approaches that offers an alternative to “heavy” development models.

Testing is a fundamental practice in Extreme Programming. XP uses three types of testing: unit, acceptance and load/stress testing. The practice for unit testing involves developing the test for the feature prior to writing the code. Acceptance tests are black box system tests and created from user stories. Load/stress tests are the same in XP projects as in other projects.

The purpose of the Masters’s thesis has been to develop a TestSuite for Extreme Programming to run and analyse unit, acceptance and load/stress tests. The focus has been concentrated to integrate the open source testing framework JUnit into to TestSuite in order to run unit tests based on JUnit much easier than up to now and to analyse failed tests more specific.

Another aspect of this thesis has been to study the Extreme Programming approach to software development and to point out its benefits and drawbacks. Especially the XP testing strategy should be analysed in detail.

1.1 Outline

The report is built on a theoretical background, which is summarized in chapter 2 to 4. A detailed software project plan is given in chapter 5. Then the user stories for the TestSuite are described in chapter 6. The design of the TestSuite is presented in chapter 7, followed by the testing efforts in chapter 8. Finally, the thesis is concluded with a summary and with recommendations for further development in chapter 9. A more detailed description of the different chapters is given below.

Chapter 2 describes and discusses some of the most common models of software development. These include the waterfall model, Boehm's spiral model and the Rational Unified Process (RUP).

Chapter 3 describes the Extreme Programming development process. It introduces the relevant elements of XP and dicusses the advantages and disadvantages of this new process.

Chapter 4 introduces the XP testing strategy. While unit and acceptance tests are the heart of this strategy, the load/stress tests complete it.

Chapter 5 provides an introduction into the thesis project and presents the software project plan. The plan includes a project work breakdown structure and a time table.

Chapter 6 desribes the user stories. The user stories "are" the requirements in other software development models.

Chapter 7 is a description of the choosen design for the TestSuite, mainly focusing on the major components.

Chapter 8 presents the testing efforts which are made to ensure the quality of the implemented TestSuite.

Chapter 9 concludes the thesis with a summary of the results, and ideas for future work.

Chapter 10 is a list of references used in the text.

Chapter 11 is a list of additional literature read in preparation for the thesis.

Appendix A contains the detailed software project plan.

Appendix B contains the Users Guide for the TestSuite.

2 History and evolution of software development models

This chapter gives an overview about the evolution of software development models. It describes and dicusses some of the most common types of models. The purpose of all the models is to control and track the way software is developed.

2.1 Definitions

A **software process** can be defined as the set of activities, methods, practices, and transformations that are used to develop and maintain software and the associated products [Paulk 1993]. There are many other definitions on this subject all with varying focus. According to [Zahran 1998] the software process can be viewed from three aspects: the process definition, the process knowledge by the users, and the process results (see Figure 1).The aspects are all important and to have an efficient process no aspect can be excluded.

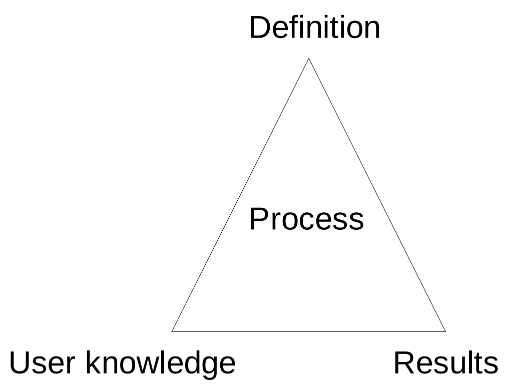


Figure 1 Process viewed from three different aspects

The four most fundamental activities of software processes are specification, development, validation, and evolution [Sommerville 1995]. These activities are common to all software processes but may be decomposed in many different ways.There is no “right” way to do such decomposition; i.e. there are many different ways to produce the same type of product.

Software process models can be seen as a description or a scheme to be followed when executing the process and developing the products [Höst 1996].

2.2 The classic process model – The waterfall model

The **waterfall model** is definitely one of the most famous, not to mention one of the oldest around. It was a response to the undisciplined code-and-fix mentality of the 1960s. Royce introduced the original version of this model, but soon many variants were developed. The basic idea behind all these models is that the development of the product follows the different phases in a **strict linear** and **sequential path**. During each phase a specific activity is executed. The phases specific for the waterfall model are requirements analysis and definition, system and software design, implementation and unit testing, integration and system testing, and operation and maintenance as shown in Figure 2. The transition between phases is formalised using documents and reviews. This means that each phase ends with a milestone, which often is nothing more than a formal document or review. Each document or review, produced at the end of phase, has to be approved by the management before a next phase can be initiated.

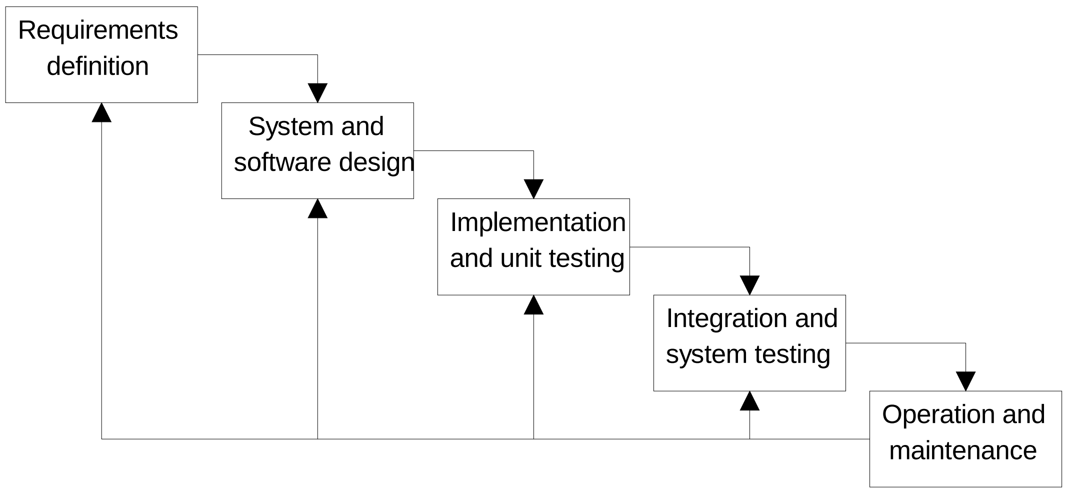

Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Figure 2 The waterfall model [Sommerville 1995]

But the waterfall model has two main drawbacks, namely the lack of iteration and the premature freezing of the requirements. This means that the users must specify at the beginning of the project once and for all exactly what they wanted. But very often the users didn't know, they contradicted themselves, they changed their minds or the programmers misunderstood the requirements. Conclusion: the system does not do, what the user wants it to do.

2.3 The next generation model – Boehm's spiral model

In the **spiral model**, the software development process is seen as a spiral: development spirals from initial conception to final product, hereby going through a number of cycles. A major feature of the spiral model is that it is **risk-driven** instead of being document-driven or code-driven [Boehm 1988]. Each cycle begins at the turn of the spiral, and is divided into the **six high-level phases** as shown in Figure 3. Starting at the centre of the spiral, development proceeds through successive turns, iterating between these six phases.

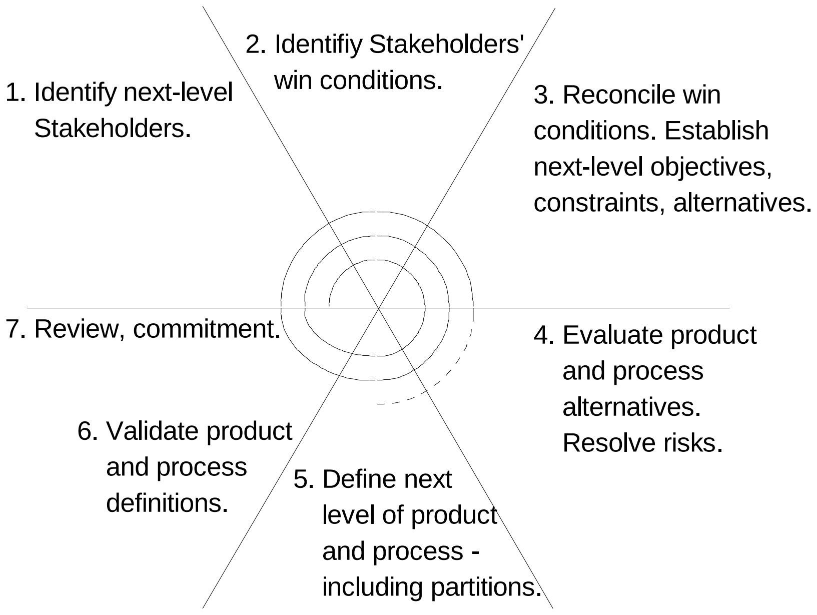

1. Identify next-level Stakeholders.
2. Identifiy Stakeholders' win conditions.
3. Reconcile win conditions. Establish next-level objectives, constraints, alternatives.
4. Evaluate product and process alternatives. Resolve risks.
5. Define next level of product and process - including partitions.
6. Validate product and process definitions.
7. Review, commitment.

Figure 3 Boehm's spiral model [Boehm 1994]

One advantage of using the spiral model is that requirements and design can be described in more detail as the knowledge of the system increases. The result of this is that not all requirements have to be written at the start of the project when they are not completely understood. But there are also points of concern. The first one is that the model is overly elaborate. Not only is the model hard to adopt, but far too expensive to. Consequently, small-time businesses may find the model unaffordable. The second point is related to the first one. The model is very elaborate, due to the large number of intermediate cycles, which makes it quite a challenge when trying to document it properly. The large number of cycles tends to produce piles of (mostly formal) papers. However, the biggest concern is formed by the contractual problems. The idea of the spiral model is to avoid premature decision-making. Risk assessments are spread throughout the project, and accordingly are the decisions that are based on it. Unfortunately, contractual agreements often enforce certain decisions, which are difficult to be altered later.

2.4 The Rational Unified Process (RUP)

Rational Software Corporation has developed a process for the entire life cycle of software development, which is called the **Rational Unified Process** [Rational 1998]. It is based on several methods, for example **Objectory** for the processes and the **Unified Modeling Language (UML)** for design and development. It is an iterative process, which is optimized for object-oriented programming. Rational sells software that supports the entire development cycle.

The Rational Unified Process includes a very detailed process description. This makes it more complicated, but with the tool support, it is still easy to get an overview of the different steps in the process. The process itself is iterative, where the initial iterations focus on the early phases, and the later iterations focus on the phases at the end of the development cycle, as shown in Figure 4.

The process is based on **six “best practices”**, that have been observed to be used by successful organizations. The first one is to **develop software iteratively**, since it is not possible to do the development in steps effectively. The understanding of a system increases at all stages, and this makes it necessary to revise requirements and other early documents. The second practice is to **manage requirements**. This includes finding the requirements that should be used and document them. By using use cases and scenarios it is easy to communicate with the customer, which should lead to a system that works as the customer wants. An effective way of designing reusable software is to use **component-based architectures**. The components are subsystems that fulfill a clear function, which makes it easier to put together larger programs from several components. Since a higher level of abstraction during development makes it easier to get an overview of the system, it helps to **visually model software**. The foundation for the modeling is UML. To ensure that the final product actually meets the requirements, it is necessary to **verify software quality**. Functionality, performance and reliability should be verified, all with the requirements specification as a foundation. It is essential to control changes to software in all software development, but it becomes even more so when the development is iterative.

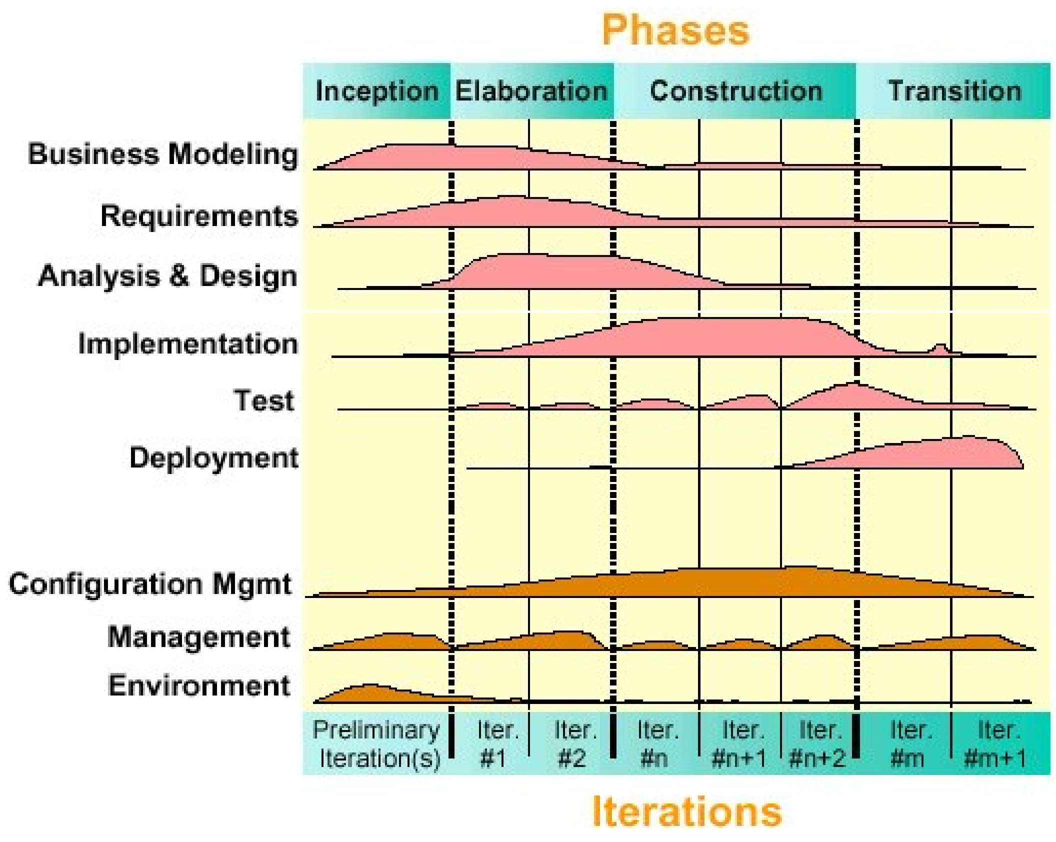


Figure 4 The Rational Unified Process (RUP) [Rational 1998]

The unified process is devided into four phases. Each one has a specific purpose, and ends with a milestone when the set of objectives has been reached. During the phases, nine different workflows are used. Figure 4 shows the different workflows, and describes how the organization is concentrated on the different workflows through the phases. This process structure makes it possible to continue to develop the documents that are made first [Rational 1998].

3 Extreme Programming – A lightweight software development process

This chapter introduces the relevant elements of Extreme Programming [Beck 1999]. It also discusses the advantages and drawbacks of XP.

As already explained in the previous chapter, heavyweight processes like “the waterfall model” or “Boehm’s spiral model” do not offer the flexibility which is expected from software development these days. Even if the requirements for the system are specified exactly in advance, the experience shows that during the development of the application the requirements are changing. This is often strengthened by the fact that the user only gets an idea of what is possible and feasible when he sees the system the first time in reality.

For this reason the demand for lightweight processes which meet this requirement is increasing. A development in this area is the Rational Unified Process. RUP can be configured for each project depending on the project size, the culture and the kind of project. Nevertheless it is a heavyweight process because developer as well as customer do not gain too much freedom.

3.1 Extreme Programming (XP)

Opposite to the Rational Unified Process stands the concept of Extreme Programming which takes the needs of change through the customer and the skills of the developer into consideration. Especially the ability of the consideration of the changing business needs of the customer makes XP to a lightweight and flexible process.

The following table [Link 2000] compares the Rational Unfified Process with the lightweight process Extreme Programming:

Rational Unified Process (RUP)	Lightweight Process Extreme Programming (XP)
As much detailled definition of all process steps as possible	As little process as possible
For medium and large sized teams	For small and medium sized teams
Maximum of documentation	Communication replaces most of the documentation

Table 1 The Rational Unified Process (RUP) vs. Extreme Programming (XP)

Extreme Programming is an approach based on a simple set of common-sense practices that, when used together, can lead to higher quality work.
The chapters from “Develop for today” to “XP tools and principles” summarizes the basic terminologies of Extreme Programming:

3.1.1 “Develop for today”

Two demands motivated the development of XP:

1. *Develop for today*. This rule means that when a developer is adding a new feature he should avoid thinking about all the uses for the capability that he’s going to need in the future. Instead he should assume that each modification is always feasible with an acceptable effort.
   But this is only possible if the second demand is taken into consideration.
2. *Do the simplest thing that could possibly work*. This means that always the simplest design should be used to solve a problem. *Simple Design* comprises different aspects:
   - *Fulfillment of requirements*: The design must meet the requirements as understanded up to now.
   - *No redundance*: The information must kept only once.
   - *Refactoring:* Refactoring is the process of changing a software system in such a way that it does not alter the behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs.

The cost of change

One of the universal assumptions of software engineering is that the cost of changing a program rises exponentially over time. This assumption has its origination from the classic waterfall life-cycle in which each activity (analysis, design, and so on) is done *once* for the entire set of system requirements.

The vertical axis in Figure 5 usually depicts the cost of changing requirements or finding defects late in the development cycle. Viewed from this perspective, traditional methods have concentrated on flexibility for changing requirements and defect prevention in early lifecycle stages.

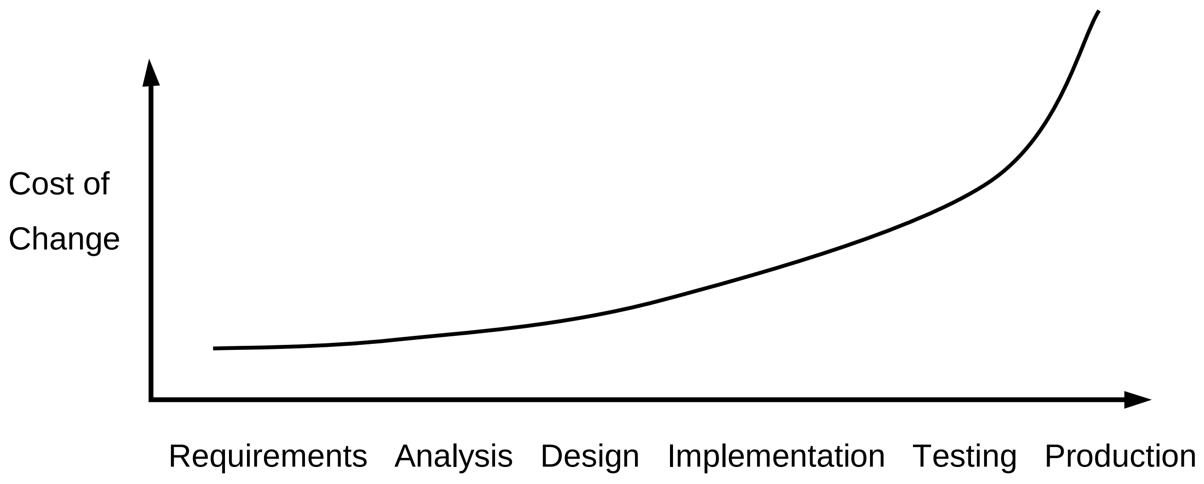


Figure 5 Cost curve of traditional development processes

The Extreme Programming process uses the notion of a flat cost of change curve as shown in Figure 6. Therefore it does not matter that much if decisions are delayed if they are not important enough or if they are to costly to implement right now.

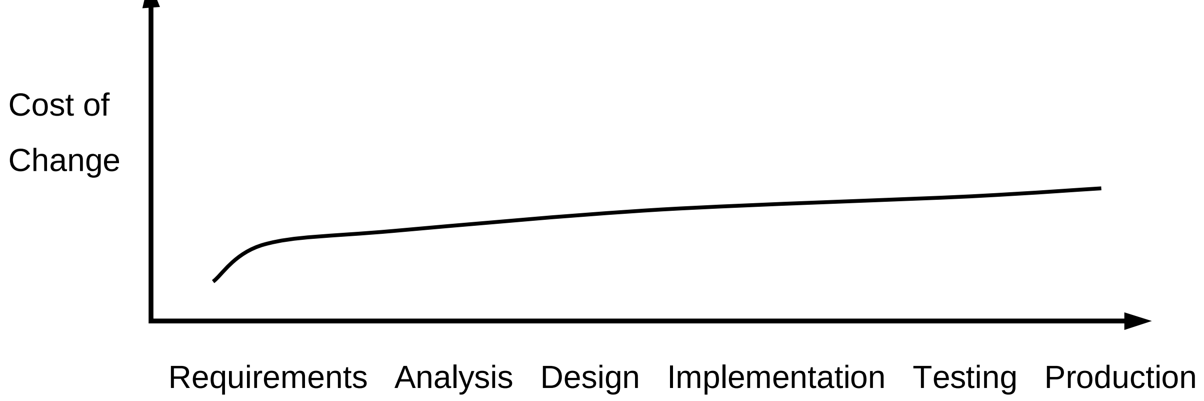


Figure 6 Cost curve of Extreme Programming

With that advantage in mind, it's possible to start with a very simple solution. This first result has to be only as flexible as necessary, because it can be extended without a big effort. Refactoring helps further leveling the expenditure by constantly keeping the software in a clear structure reducing the hard work needed for the future extension.

Extreme Programming uses a initial small upfront design of the simplest solution for the basic requirements to create a running system instead of the extensive upfront design for all requirements of traditional development processes.

3.1.2 Short development and iteration cycles

XP promotes “fast” (i.e., fine-grained) iterations, thus allowing system development to be highly reactive with respect to changing requirements. In principle XP distinguishes between releases, iterations and tasks:

- Release: A release is a version of a system with enough new features that it can be delivered to users outside the development group. The first release comprises three to six month. All further releases represents perhaps one to three month’ work.
- Iteration: Each release consists of a series of iterations. Iterations are of a fixed length, one to four weeks long.
- Task: Each iteration contains a dozen tasks of one to three days in length.

Figure 7 shows the evaluation of the “waterfall model” (a) and its long development cycles (analysis, design, implementation, test) to the shorter, iterative development cycles within, for example, the “spiral model” (b) to Extreme Programming’s (c) blending of all these activities, a little at a time, throughout the entire software development process.

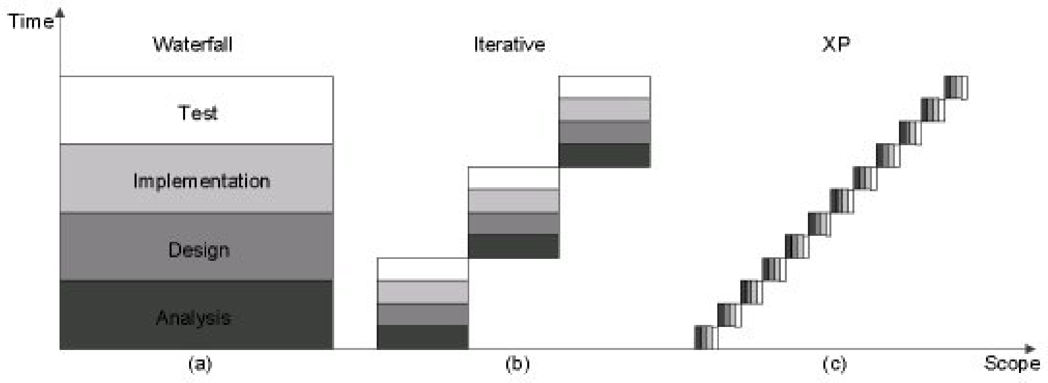


Figure 7 Development cycles in traditional processes and in XP

In Extreme Programming, analysis, design, implementation and test runs in parallel and in very small steps. For this reason, the possibility to react to changing requirements very quick is actually “build into” XP.

3.1.3 XP tools and prin ciples

Based on this key elements there are several practices developed which determine the XP process. All practices are (more or less) dependent on each other and have mutual influence.

User Stories

User Stories are written by the customers as things that the system needs to do for them. They are a lightweight form of use cases and replaces a large requirements document. Each User Story is a short description of about three sentences on blank cards in the language of the customer without techno-syntax.

User Stories should only provide enough detail to make a reasonably low risk estimate of how long the story will take to implement. When the time comes to implement the story developers will go to the customer and receive a detailed description of the requirements face to face.

User stories also drive the creation of the acceptance tests.

The Planning Game

After User Stories have been written a release plan meeting (in XP called the Planning Game) is used to create a release plan.

The essence of the release planning meeting is for the development team to estimate each user story in terms of ideal programming weeks. An ideal week is how long it would take to implement the story in code if there were no distractions, no other assignments, and the developer knew exactly what to do. Each user story should take one to three ideal programming weeks to implement. Longer than three weeks means that the user story should be broken down further. User stories which are shorter than one week can be grouped together. To convert the estimation into real time it must be multiplied with a load factor. A load factor from two to five real weeks per ideal week is normal.

The customer then sorts the stories by priority.

Afterwards developers and customers work together to decide what stories constitude the next release and when it will be ready to put into production. There are two ways to drive the commitment, story driven and date driven.

Story driven commitment: The customer decides which user stories are implemented in the next release. The development team calculates and announces the release date based on their estimations.

Date driven commitment: The customer picks a release date. The develop team calculates and announces the available real programming weeks between now and the date. The customer picks stories who adds up to that number.

About 80 user stories plus or minus 20 is a perfect number to create a release plan during release planning.

System Metaphor

The system metaphor describes the logical architecture of the system. It provides a common vision and a shared vocabulary to everyone – customers, programmers, and managers. It helps generate a new understanding of the problem and the system. In the Rational Unified Process the system metaphor is the architecture baseline.

Coding Standards

Code must be formatted to agreed coding standards. Coding standards keep the code consistent and easy for the entire team to read and refactor. In XP each development team can define their own coding standards. This offers the change that everybody follows the standards.

Iteration and Task Planning

As already mentioned a release consists of several iterations and an iteration of several tasks (Figure 8).

Iteration planning starts by again asking the customer to pick the most valuable stories, this time out of the stories remaining to be implemented in this release.The development team breaks the stories down into tasks, units of implementation that one person could implement in a few days. Next, programmers sign up for the tasks they want to be responsible for implementing. After all the tasks are spoken for, the programmer responsible for a task estimates it. Everyone’s task estimates are added up, and if some programmers are over and some are under, the under commited programmers take more tasks. In this way it is easy to find out if the estimation for the whole release is still valid.

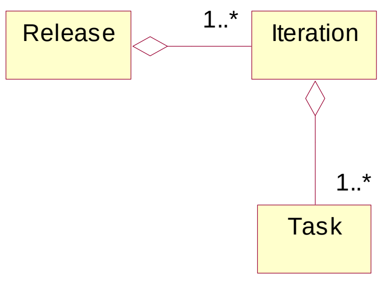


Figure 8 Coherence between release, iteration and task

Pair Programming

To implement a task, the responsible programmer first has to find a partner because all production code is written with two people at one machine.

When programming in pairs, a large part of the communication occurs as the two people look at the same code at one machine. Comments are expressed through verbal communication. Discussion about the current implementation, possible alternatives, errors, or style violations ensures that the result will be of high quality. Questions can be answered promptly, thus improving the effect of learning.

For all that pair programming is a controversial attribute of XP because managers view programmers as a scarce resource, and are reluctant to "waste" such by doubling the number of people needed to develop a piece of code.

Unit Tests

Unit testing in XP is not testing done by specialized testers, it is part of the daily development routine of a programmer. Unit testing means to test a unit of code. A unit of code is generally a class in an object oriented system.
One of the most useful times to write unit tests is before the developer starts implementing the task. By writing the test the developer is asking himself what needs to be done to add the function. Writing the test also concentrates on the interface rather than the implementation. In a project with changing and developing requirements, unit tests help to stabilize the system. They also allow to change much faster and more aggressive, because any errors introduced with a change are detected immediately. And they help to create a very short feedback loop for the developer. See the next chapter for more information.

Acceptance Tests

Where unit tests are owned by the developers, acceptance tests are owned by the customer. They are created from user stories. During an iteration the user stories selected during the iteration planning meeting will be translated into acceptance tests. The translation can either be done by programmers or by a separate team which also runs the actual tests. Each story can have one or many acceptance tests, what ever it takes to ensure the functionality works.
Acceptance tests should be automated so they can be run often. The acceptance test score is published to the team. It is the team's responsibility to schedule time each iteration to fix any failed tests.
Acceptance tests will be discussed in more detail in the chapter that follow.

Collective Code Ownership and Refactoring

Extreme programming considers code to belong to the project, not to an individual engineer. As engineers develop a task, they may browse into and modify any class. They are responsible for keeping all the unit tests running (and writing new ones for new functionality).

Whenever a developer thinks that the code he uses for implementing a task contains redundancy, unused functionality or an obsolete design, he should start to refactor. Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs.

Continous Integration

The aim is to have a permanent running system . For this reason each time a developer (pair) finishes a task, they should integrate what they have done. In any case they should never hold onto changes for more than a day. This approach often avoids diverging or fragmented development efforts, where developers are not communicating with each other about what can be reused, or what could be shared. After the integration of the implemented task, all the tests must run successful.

3.2 Advantages and Disadvantages of Extreme Programming

Kent Beck's *Extreme Programming explained: Embrace Change* [Beck 1999] makes a number of suggestions on how to improve software engineering practice. Although he provides little or no data to support his recommendations. Following the advantages and disadvantages for using XP are pointed out based on the opinion of the writer. All the statements are proved with data from [Humphrey 1995] and [Humphrey 2000] about the Personal Software Process and the Team Software Process on similar practices.

Advantages:

1. Emphasis on customer involvement: A major help to projects where it can be applied.

2. Emphasis on teamwork and communication: This is very important in improving the performance of just about every software team.
3. Programmer estimates before committing to a schedule: This helps to establish rational plans and schedules and to get the programmers personally committed to their schedules.
4. Emphasis on responsibility for quality: Unless programmers strive to produce quality products, they probably won't.
5. Continuous measurement: Since software development is a people-intensive process, the principal measures concern people. It is therefore important to involve the programmers in measuring their own work.
6. Incremental development: Consistent with most modern development methods.
7. Simple desing: Though obvious, worth stressing at every opportunity.
8. Frequent redesing, or refactoring: A good idea but could be troublesome with any projects.
9. Having engineers manage functional content: Should help control function creep.
10. Frequent, extensive testing: Cannot be overemphasized.
11. Continuous reviews: A very important practice that can greatly improve any programming team's performance.

Disadvantages:

1. Code-centered rather than design-centered development: Although the lack of XP design practices might not be serious for small programs, it can be disastrous when programs are larger than a few thousand lines of code or when the work involves more than a few people.
2. Lack of design documentation: Limits XP to small programs and makes it difficult to take advantage of reuse opportunities.
3. Producing readable code (XP's way to document a design) has been a largely unmet objective for the last 40-plus years. Furthermore, using source code to document large systems is impractical because the listings often contain thousands of pages.
4. Lack of structured review process: When engineers review their programs on the screen, they find about 10-25 % of the defects. Even with pair programming, unstructured onlinereviews would still yield only 20-40%.

With a structured review process, most engineers achieve personal review yields of 60-80%, resulting in high-quality programs and sharply reducing test time.

5. Quality through testing: A development process that relies heavily on testing is unlikely to produce quality products. The lack of an orderly design process and the use of unstructured reviews mean that extensive and time-consuming testing would still be needed, at least for any but the smallest programs.
6. Lack of a quality plan: Quality planning helps properly trained teams produce high-quality products, and it reduces test time by as much as 90%. XP does not explicitly plan, measure, or manage program quality.
7. Data gathering and use: In the opinion of the writer, unless the data are precisely defined, consistently gathered, and regularly checked, they will not be accurate or useful. The XP method provides essentially no data-gathering guidance.
8. Limited to a narrow segment of software work: Since many projects start as small efforts and then grow far beyond their original scope, XP's applicability to small teams and only certain kinds of management and customer environments could be a serious problem.
9. Methods are only briefly described: While some programmers are willing to work out process details for themselves, most engineers will not. Thus, when engineering methods are only generally described, practitioners will usually adopt the parts they like and ignore the rest.
10. Obtaining management support: The biggest single problem in introducing any new software method is obtaining management support. The XP calls for a family of new management methods but does not provide the management training and guidance needed for these methods to be accepted and effectively practiced.
11. Lack of transition support: Transitioning any new process or method into general use is a large and challenging task. Successful transition of any technology requires considerable resources, a long-term support program, and a measurement and analysis effort to gather and report results. At the moment there is no such support for the XP.

4 Testing – A core practice of Extreme Programming

This chapter describes the XP testing strategy. It introduces the following types of tests: unit, acceptance and load/stress tests. These types are the heart of this strategy.

Testing is a necessary evil to many programmers! For that reason it is obvious that developers generally perform an inadequate number of inadequate tests and figure that if the users don't find a bug, there is no bug. Why does this happen?

1. *The psychology of success and failure*. The developers are so focused on getting their code to work correctly that they generally shy away from bad news, from even wanting to take the chance of getting bad news. Better to do some cursory testing, confirm that it seems to be working okay, and then wait for others to find bugs, if there are any (as if there were any doubt).
2. *Deadline pressures*. In the age of the Internet, time to market determines all. It becomes the mantra of companies. Everything is needed yesterday. Therefore it is a wide spread practice to release pre-beta software as production and let the users test/suffer through the applications.
3. *Management's lack of understanding*. IT management is notorious for not really understanding the software development process. If the developers aren't given the time and authority to write (in the broadest sense, including testing, documentation, refinement, and so forth) their own code properly, they'll always end up with buggy junk that no one wants to admit owenership of.
4. *Overhead of setting up and running tests*. If it's a big deal to write and run tests, they won't get done. The developers say, that they don't have time, there's always something else to work on, and so on. One consequence of this point is that more and more of the testing is handed over to the quality assurance department, if there is one. That transfer of responsibility is, on the one hand, positive. Professional quality assurance testers can have a tremendous impact on application quality. Yet developers must take and exercise responsibility for unit testing of their

own code; otherwise, the testing/ quality assurance process is much more frustrating and extended.

The bottom line is that the code almost universally needs more testing.
In Extreme Programming testing has a very high priority:

"If testing is good, everybody will test all the time (unit testing), even the customers (functional testing)." (s. [Beck 1999], page XV)

In Extreme Programming the three fundamental test types are:

- Unit Tests (UTs)
- Acceptance Tests (ATs), changed from Functional Tests (FTs)
- Load and Stress Tests (LTs and STs)

4.1 Unit Tests (UTs)

Unit tests are one of the corner stones of Extreme Programming. Unit testing means to test a unit of code. A unit of code is generally a class in an object oriented system. However, it could also be a component or any other piece of related code. The tests are usually written in the same computer language as the production code. The test verification is therefore code, which compares actual results to expected results. If a result is unexpected, the test fails.

Unit tests XP style is a little different. XP recommends that:

- Before a developer writes production code, he should build unit tests to test that code. By taking this approach, he implements the interface to the production code, flushes out poor logic, and defines the requirements. ("This is what the code must do to be considered correct.")
- Before a developer fixes a bug, he should write a unit test to verify the bug. That way he build his test case suite as he go, he can make sure that the bug is fixed with his new test, and, best of all, other developers can easily add to the test suite and run the tests.
- Unit tests should should be run automatically and follow a "red light, green light" approach: If all of the code works, he gets a "green light". If anything goes wrong in the test, the "red light" tells the developer which program failed.

- Before a developer releases production code, every unit test in the entire system must be running at 100 %.

For autmatically verifying test results of unit tests, Kent Beck wrote a testing framework for Smalltalk some years ago (called SUnit). He and Erich Gamma then provided the JUnit tool for Java. Today, there are CPPUnit, PerlUnit, PyUnit for Python, VBUnit for Visual Basic, and many other languages as well. All these testing frameworks are Open Source and within the limitation of the programming language, they work the same way.

4.2 Acceptance Tests (ATs)

Acceptance tests give customers and developers confidence that the whole software product is progressing in the right direction. They provide quality assurance to the customer.

Acceptance tests typically treat the whole system as a black box as much as possible. In a GUI-based system, they operate through the GUI. In a file or database update program, the test just look at how the data is changed for certain inputs.

Acceptance tests must also be automated. It is not possible to have comprehensive repeatable tests if the results must be checked manually. There are a lot of ways this can be done. For example:

- If the program is a batch program, which reads input and produces output, the customers can make a series of input files. After running the program, the customers have to check the output manually (and carefully) once and then the programmers can write simple scripts that compare the test output to the known-good output.
- Another possibility is to build an input recorder into the product and to let the customers enter their test values. The recorder saves the user input and the generated output. The customers can verify and correct the generated output. The test case can be repeated automatically after each iteration and the output is compared with the verified output.
- Also simple file based tools like grep, diff etc. can be used to extract test data from spreadsheets and to check the results.

4.3 Load (LTs) and Stress Tests (STs)

Companies are deploying e-commerce applications at record pace. For this kind of applications load and stress tests are absolutely mandatory and a useful addition to unit and functional tests.

The most significant difference between load and stress tests [Goglia 1993] is what is expected to be gained from each. Stress testing is an attempt to push the system so hard that it breaks. The part of the system which malfunctioned is investigated to see how it can be corrected or improved upon. Load testing loads the system with activity which simulates legitimate user activity. The statistics which result from load testing are then used to predict what performance and response time users will see.

There are at least two ways to implement load and stress testing. One method is accomplished by installing agents on several user workstations involved in the test. A test controlling program on one workstation communicates with all workstations via their agent programs. The test to be executed is passed to the agent. When the test is completed, the agent communicates with the controlling program. The controller at this point may have additional testing for the agent, or ist tasks may be done.

Another technique for performing load and stress testing is to have a single workstation simulating a number of client workstations. A test program, running on a workstation, simulates the activity of a number of user workstations. This is possible because normal interaction between a client and the server includes relatively little communication interspersed by a lot of pauses. The test program needs to keep the individual client sessions with the server isolated.

5 TestSuite for Extreme Programming – Introduction and Software Project Plan

This chapter provides a brief introduction into the thesis project. It also includes a detailed project plan.

5.1 Introduction

In March of 1996 Kent Beck started the Chrysler Comprehensive Compensation (C3) project. The aim was to replace the payroll department's three twenty years old systems by a unified system. Using Kent Beck's Extreme Programming process, DaimlerChrysler started over from scratch and delivered a very successful result.

The system was implemented with VisualWorks Smalltalk V2.5.1. For the execution of the over 3000 unit tests they used Kent Beck's Testing Framework. The over six hundred functional tests ran under the same Testing Framework, but with a lot more support to do all the necessary set up (input files, running a payment workplan, checking output files, etc).

By and by the Testing Framework was ported from Smalltalk to many other languages including C, C++, VisualBasic, Perl, Java etc.

Today most of the software systems at DaimlerChrysler are implemented using the Java language and HTML. To support the upcoming and already started Java projects , the testing group at DaimlerChrysler decided to provide a TestSuite for the 3 most common test types:
These are:

- unit tests
- functional tests
- and load/stress tests

For this reason they first tried some testing tools from leading vendors, but none of them met the requirements. Then they evaluated 3 potential tools for unit testing Java code. JUnit [JUnit], the Java porting of Kent Beck's Testing Framework evaluated as the leader and was recommended to new projects.

At the same time they offered a project in Stuttgart which was choosen for this Master's Thesis. The goal was to integrate JUnit into a GUI based TestSuite which makes it first easier to run and analyse unit tests and which is second expandable for other kinds of tests, in particular for acceptance and load/stress tests. The requirements for the TestSuite should be delivered by the C3 project and other projects, which are using JUnit at the moment.

In a next version of the TestSuite (i.e. after another Master's Thesis), support for all the three test types should be integrated into this TestSuite.

5.2 Software Project Plan

This software project plan serves as the foundation for the software engineering efforts throughout the analysis, design, coding and testing of the TestSuite being developed. It provides baseline scheduling information like the project work breakdown structure and the time table.

Planning in XP is a very specific and rigorously defined process. Once familiar in the technique, the process can be modified to the needs of the projects. The following lines summarizes the process of planning and developing in XP and complete the already mentioned in chapter 3:

1. The customer writes user stories.
2. The developer estimate user stories.
3. The customer and the manager decide which user stories go in this release.
4. The developers take the release stories and break them into tasks.
5. The developers sign up for tasks.
6. The developers estimate tasks.
7. Any discrepencies between story estimates and task estimates are resolved (task estimates override story estimates).

8. The customer and the manager make a final decision about what is in this release, what is in each iteration (a release will consist of three to four iterations of one to four weeks each).
9. The programmers implement the user stories.
10. The manager trackes progress every few days.
11. The customer is immediately notified if the progress is faster or slower than expected, so that s/he can make appropriate scope decisions.
12. The customer runs acceptance tests on completed user stories.

The user stories “are” the requirements. No further analysis is down. The on site customer answers questions about user stories during story creation, estimation, planning and development.

Design documentation is only done when absolutely necessary, and clear in the developers minds. CRC cards or UML class and sequence diagrams are the recommended design methods.

5.3 Project Work Breakdown Structure

See description in *Appendix A - Detailed Project Plan.*

5.4 Project time table

See description in *Appendix A - Detailed Project Plan.*

6 Analysis

This chapter presents the user stories for the TestSuite. The user stories are listed below in ascending order.

Analysis can be loosely described as the process of finding out what the customer wants. It can be done very formally, with objects and diagrams, or it can be done very informally. Extreme Programming chooses informal analysis. In this case a developer workshop was organized and the user stories were written down. These stories “are” the requirements in other software development models.

6.1 User stories

User story 1:	The TestSuite must be implemented in Java and it must be able to execute existing unit tests written in Java and using the JUnit testing framework.
User story 2:	The TestSuite should be expandable at any time e.g. for special unit tests of Enterprise Java Beans (EJBs), Servlets etc., but also for other types of tests e.g. functional tests or load and stress tests.
User story 3:	After altering and compiling production code, the unit tests must be reloaded and executed with only one mouse click to make sure that the developers didn’t introduce new bugs.
User story 4:	There should be a possibility to filter the failed tests because these are the actual interesting unit tests.
User story 5:	When a test case fails the TestSuite must be able to print detailed information.

User story 6: The passed unit test should have another colour (for example green) to the failed tests (for example orange or red) in the GUI.

User story 7: A message in the GUI should show how many unit tests passed or failed.

User story 8: There must be a possibilty to select and execute a subset of the open unit tests.

User story 9: To be able to use the TestSuite in Integrated Development Environments (e.g. VisualAge for Java), the unit tests must be opened and executed by typing the package and class name into a dialog box (e.g. `com.foo.BarTest`).

User story 10: After selecting a directory in a file chooser dialog all the unit tests within this directory must be opened/executed in the TestSuite.

User story 11: A “recursive“ directory feature should recursively find and open/execute all unit tests from a starting point in the filesystem without modifying of source code.

User story 12: Recently opened unit tests should be shown under the unit test menu. This provides a handy shortcut for quickly accessing the last few test runs.

User story 13: For long running unit tests a progress bar should indicated the percentage of the test that is completed.

7 Design

This chapter describes the design choosen for the TestSuite. For the purpose of modelling, class and sequence diagrams were used based on the Unified Modelling Language (UML) [Fowler 1998]. These diagrams only contain the classes and methods necessary to describe the corresponding functionality. All other helper classes and methods were hidden. For a better understanding of this chapter it may be helpful to have a look at *Appendix B - Users Guide*.

7.1 Overview of the main packages

The diagram below shows the main packages that make up the TestSuite. The package diagram's principal focus is the system's packages and their dependencies. Basically there are four main packages:

- `junit.framework`
- `com.daimlerchrysler.testsuite`
- `com.daimlerchrysler.testsuite.gui`
- `javax.swing`

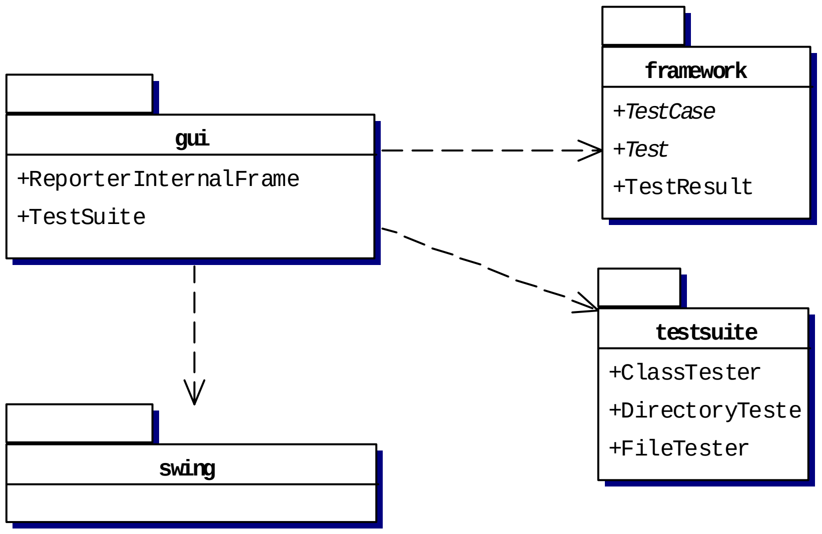


Figure 9 The four main packages and their relationships

Each package contains a number of classes. Many of the existing classes were left out of the diagram (including all the helper classes in other packages) and only the most important were mentioned. Additionally classes and dependencies to the Java base libraries were left out, but an exception for `swing` was made because that line in the diagram should illustrate that those classes use a GUI.

Furthermore the diagram shows that the GUI classes have relationships to classes within the package `com.daimlerchrysler.testsuite` to find, load and check tests and to classes within `junit.framework` to execute them.

7.2 Design of the framework package

As required from the customer, the TestSuite integrates JUnit for the execution of the unit tests. The most interesting package in JUnit is the framework package. Figure 10 shows a class diagram of this package.

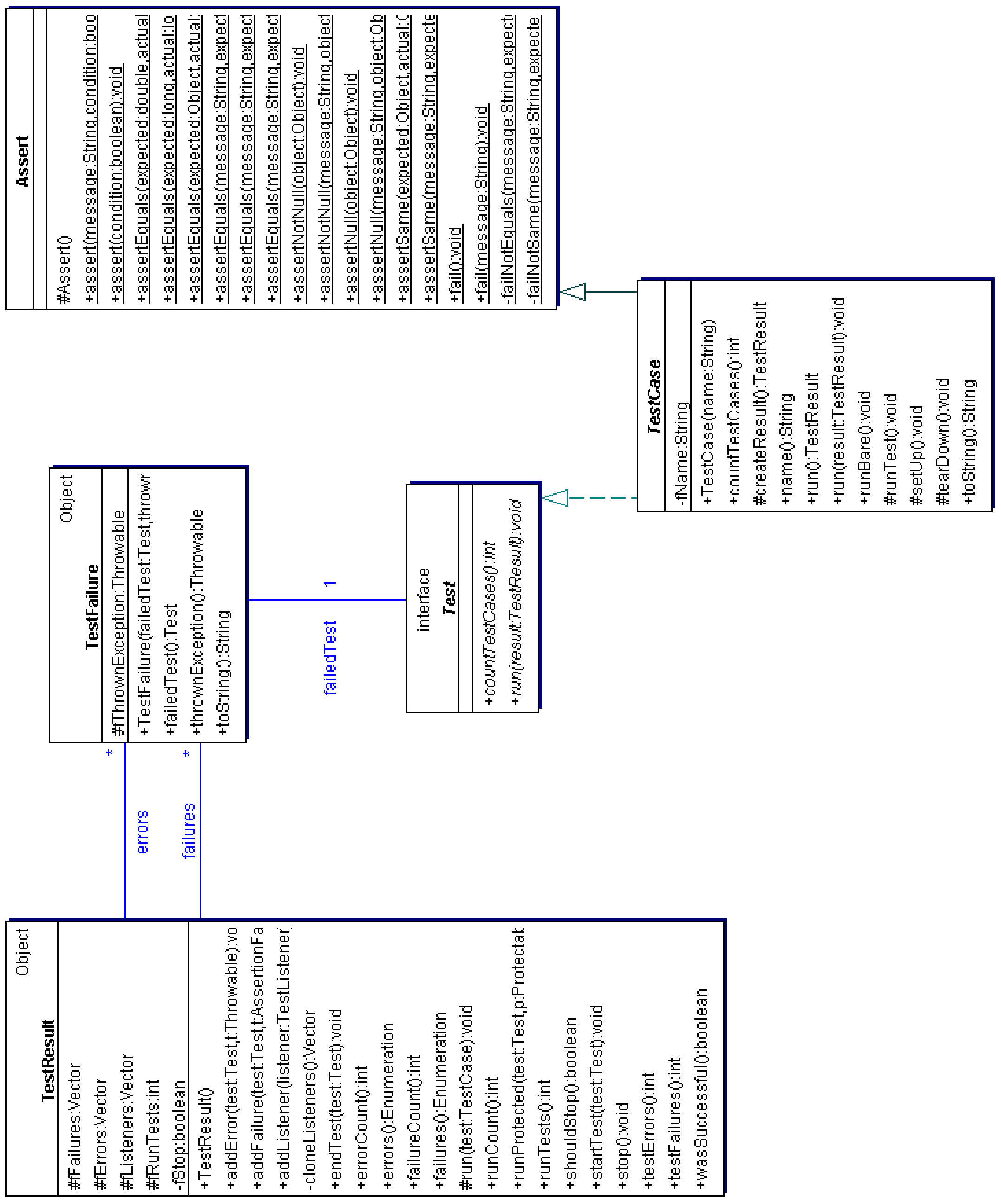

Assert
#Assert()
+assert(message:String,condition:boo
+assert(condition:boolean):void
+assertEquals(expected:double,actual
+assertEquals(expected:long,actual:lo
+assertEquals(expected:Object,actual:
+assertEquals(message:String,expect
+assertEquals(message:String,expect
+assertEquals(message:String,expect
+assertNotNull(object:Object):void
+assertNotNull(message:String,object
+assertNull(object:Object):void
+assertNull(message:String,object:Ob
+assertSame(expected:Object,actual:C
+assertSame(message:String,expecte
+fail():void
+fail(message:String):void
-failNotEquals(message:String,expect
-failNotSame(message:String,expecte
TestCase
-fName:String
+TestCase(name:String)
+countTestCases():int
#createResult():TestResult
+name():String
+run():TestResult
+run(result:TestResult):void
+runBare():void
#runTest():void
#setUp():void
#tearDown():void
+toString():String
interface
Test
+countTestCases():int
+run(result:TestResult):void
Object
TestFailure
#fThrownException:Throwable
#fFailedTest:Test
+TestFailure(failedTest:Test,thrown
+failedTest():Test
+thrownException():Throwable
+toString():String
failedTest
1
errors
*
failures
*
Object
TestResult
#fFailures:Vector
#fErrors:Vector
#fListeners:Vector
#fRunTests:int
-fStop:boolean
+TestResult()
+addError(test:Test,t:Throwable):vo
+addFailure(test:Test,t:AssertionFa
+addListener(listener:TestListener
-cloneListeners():Vector
+endTest(test:Test):void
+errorCount():int
+errors():Enumeration
+failureCount():int
+failures():Enumeration
#run(test:TestCase):void
+runCount():int
+runProtected(test:Test,p:Protectat
+runTests():int
+shouldStop():boolean
+startTest(test:Test):void
+stop():void
+testErrors():int
+testFailures():int
+wasSuccessful():boolean

Figure 10 The most important classes of the package junit.framework

The class `TestCase` represents a single test. The `run()` method, inherited from the interface `Test` will run the test case by executing `setUp()`, `runTest()`, and `tearDown()` in sequence. `runTest()` executes the method with the actual testing code by using reflection. Such testing code will do some things and then use `assert()`, `assertEquals()`, `assertNotNull()` or another method inherited from the class `Assert` to test expressions or values. `setUp()` and `tearDown()` are defined as blank methods, and the programmer of a test case can override them to setUp and tearDown a test fixture.

When a programmer or a tester run a test case, the results are stored in an instance of the class `TestResult`. This instance keeps two collections of test failures; these are indicated by the two association lines to the class `TestFailure`. The failures collection contains details of where assertions failed, indication a test failure.The errors collection contains details of any unhandled exception in the code that would otherwise lead to crashes. Each instance of `TestFailure` records both the test and the exception that generated it.

The sequence diagram below (Figure 11) shows how a test runs. For example a developer presses the *Re-Exercise* button. After sending some GUI-specific messages, the instance of `ReporterInternalFrame` creates a new object of the type `TestResult`. Next, the same object tells the `TestSuiteExtension` to run, passing the created instance of `TestResult` as an argument. The argument acts as a Collecting Parameter, described in [Beck 1996], for all the test cases. The `TestSuiteExtension` now executes run on all the test cases. Figure 11 shows 3 test cases, one success, one error, and one failure.

In the successful test run, nothing untoward happens, and the test case returns. With an error, an exception is raised somewhere and the run method handles it. The handler calls `addError()` in the `TestResult`.

With the failure, the test calls the `assert()` method to check a value. If the value is not correct, the `assert()` method throws an `AssertionFailed` exception that the `run()` method catches. The handler then adds a failure to the instance of `TestResult`.

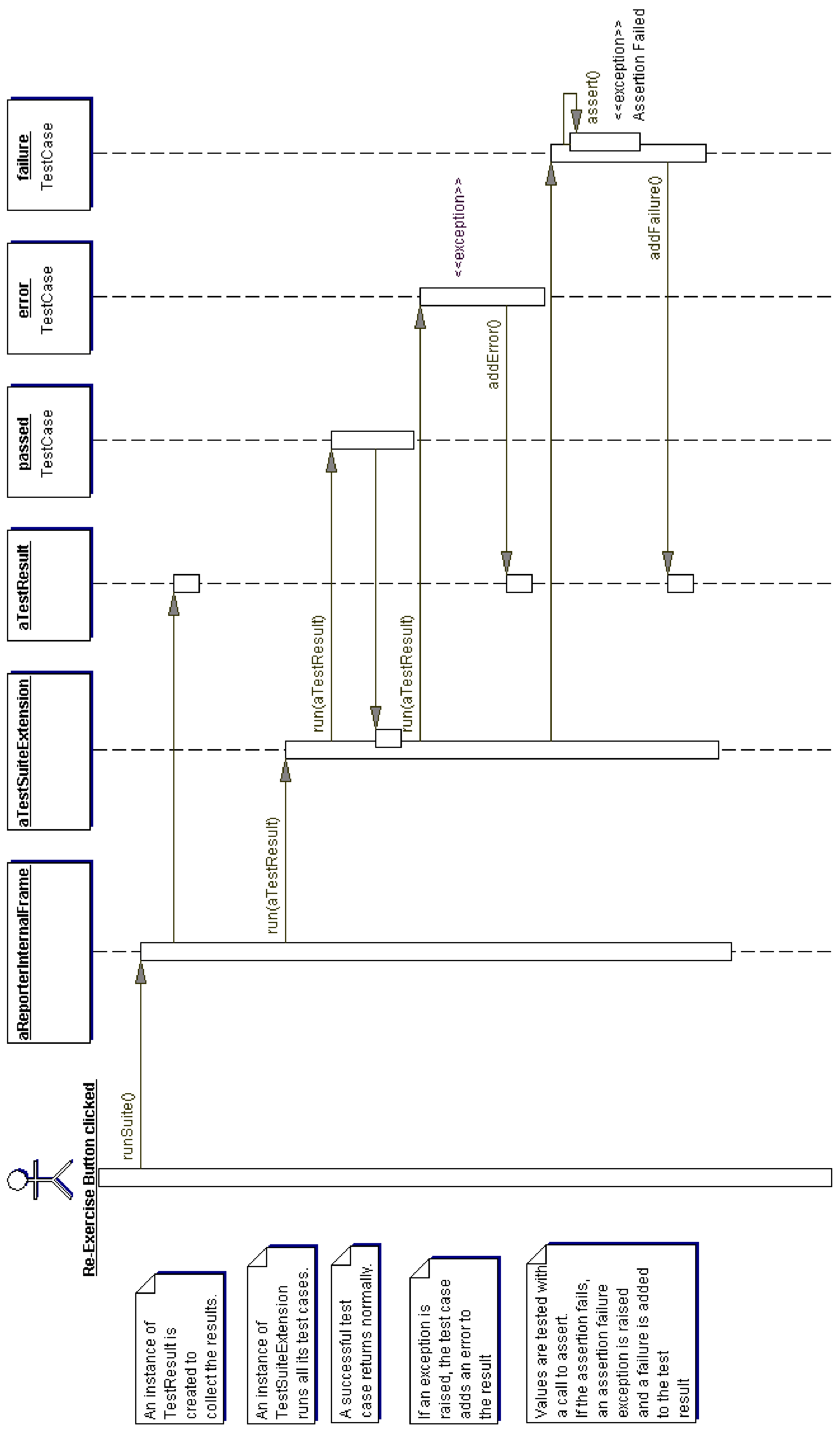

failure
TestCase
error
TestCase
passed
TestCase
aTestResult
aTestSuiteExtension
aReporterInternalFrame
Re-Exercise Button clicked
runSuite()
run(aTestResult)
run(aTestResult)
run(aTestResult)
<<exception>>
addError()
assert()
<<exception>>
Assertion Failed
addFailure()
An instance of TestResult is created to collect the results.
An instance of TestSuiteExtension runs all its test cases.
A successful test case returns normally.
If an exception is raised, the test case adds an error to the result
Values are tested with a call to assert. If the assertion fails, an assertion failure exception is raised and a failure is added to the test result

Figure 11 Sequence diagram of the three test cases

7.3 Design of the testsuite package

Before the test cases can be executed, they must be found in a directory tree or in a Integrated Development Environment (IDE), loaded and finally they must be checked, if they are test cases. Figure 12 shows the main classes of the package `com.daimlerchrysler.testsuite`, which are responsible for that.

`TestSuiteExtension` findes the test cases among the class names from `ClassTester`, `FileTester` and `DirectoryTester`. With the method `addClassIfTestCase()` determines it whether the class is derived from `TestCase`. If not, it is not a test case and it must not be added into the `ArrayList`.

`ClassTester` finds a class given by its class name (e.g. `junit.samples.SimpleTest`) and `FileTester` finds the class given by a file name (e.g. `d:/projects/junit3.2/junit/samples/SimpleTest.class`). `DirectoryTester` searches a directory or recursively a directory tree for class files.

`FileSystemClassLoader` loads each class found by `ClassTester`, `FileTester` and `DirectoryTester` with the method `loadClass()`.

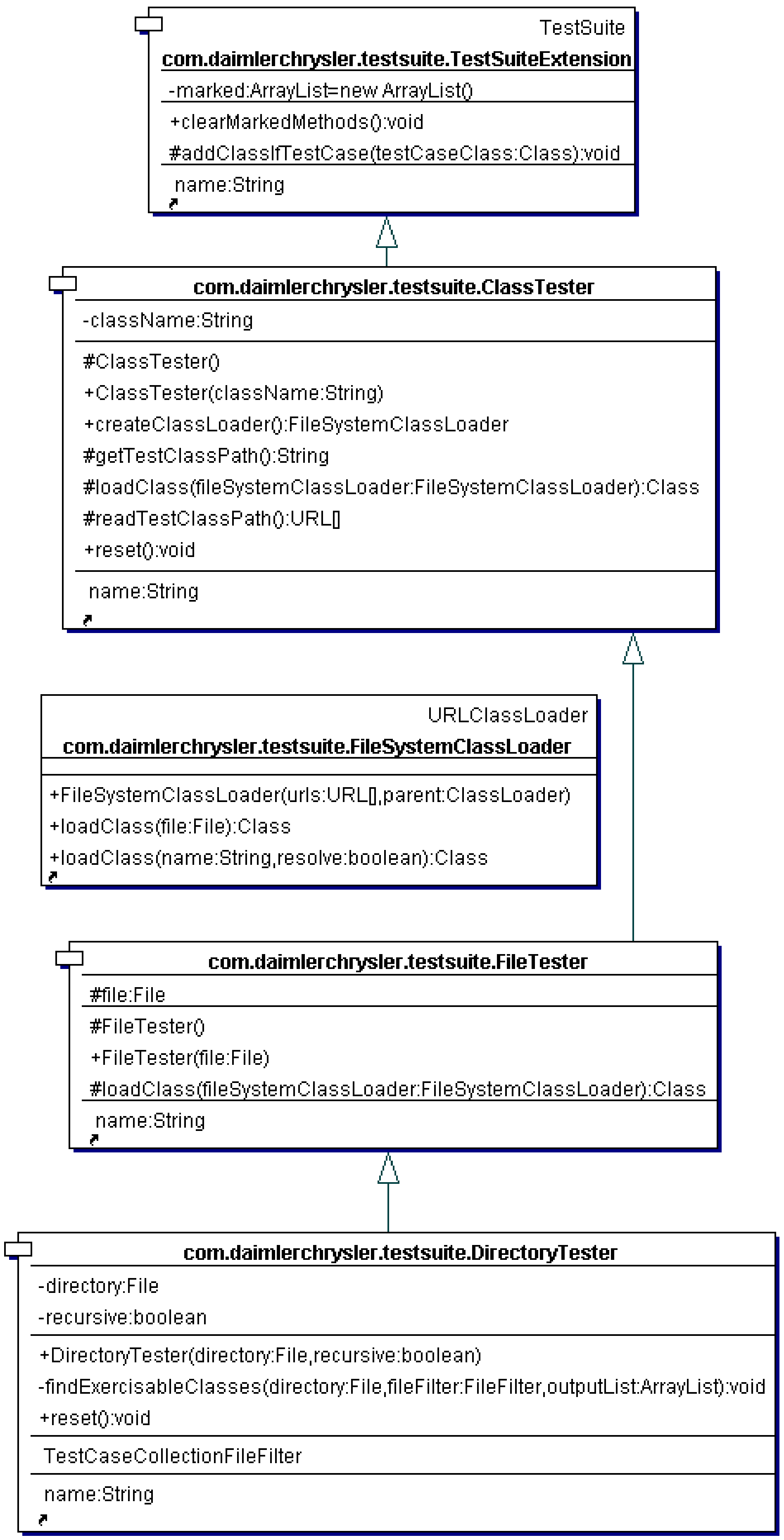

TestSuite
com.daimlerchrysler.testsuite.TestSuiteExtension
-marked:ArrayList=new ArrayList()
+clearMarkedMethods():void
#addClassIfTestCase(testCaseClass:Class):void
name:String
com.daimlerchrysler.testsuite.ClassTester
-className:String
#ClassTester()
+ClassTester(className:String)
+createClassLoader():FileSystemClassLoader
#getTestClassPath():String
#loadClass(fileSystemClassLoader:FileSystemClassLoader):Class
#readTestClassPath():URL[]
+reset():void
name:String
URLClassLoader
com.daimlerchrysler.testsuite.FileSystemClassLoader
+FileSystemClassLoader(urls:URL[],parent:ClassLoader)
+loadClass(file:File):Class
+loadClass(name:String,resolve:boolean):Class
com.daimlerchrysler.testsuite.FileTester
#file:File
#FileTester()
+FileTester(file:File)
#loadClass(fileSystemClassLoader:FileSystemClassLoader):Class
name:String
com.daimlerchrysler.testsuite.DirectoryTester
-directory:File
-recursive:boolean
+DirectoryTester(directory:File,recursive:boolean)
-findExercisableClasses(directory:File,fileFilter:FileFilter,outputList:ArrayList):void
+reset():void
TestCaseCollectionFileFilter
name:String

Figure 12 The most important classes of the package com.daimlerchrysler.testsuite

The following sequence diagrams (Figure 13 and Figure 14) show what happens when a user wants to search a directory for test cases. First he selects the corresponding directory within a file chooser dialog. After doing some GUI specific work an instance of `DirectoryTester` is created with the following two parameters: The name of the directory and a boolean value which indicates whether the whole directory tree (`true`) or only the specified directory (`false`) should be searched for test cases.

The method `findExercisableClasses()` then stores all candidates (class files and directories) of the root directory in an `ArrayList`. If the complete directory tree should be searched for class files then all candidates are checked if they are actually a directory. If one is a directory, the code recurse into it. If the candidate is a file then it is stored in another `ArrayList`.

Finally all the class files in the `ArrayList` are loaded by an instance of `FileSystemClassLoader`. If the class is a test case, the method `addClassIfTestCase()` adds the class to its list of test cases.

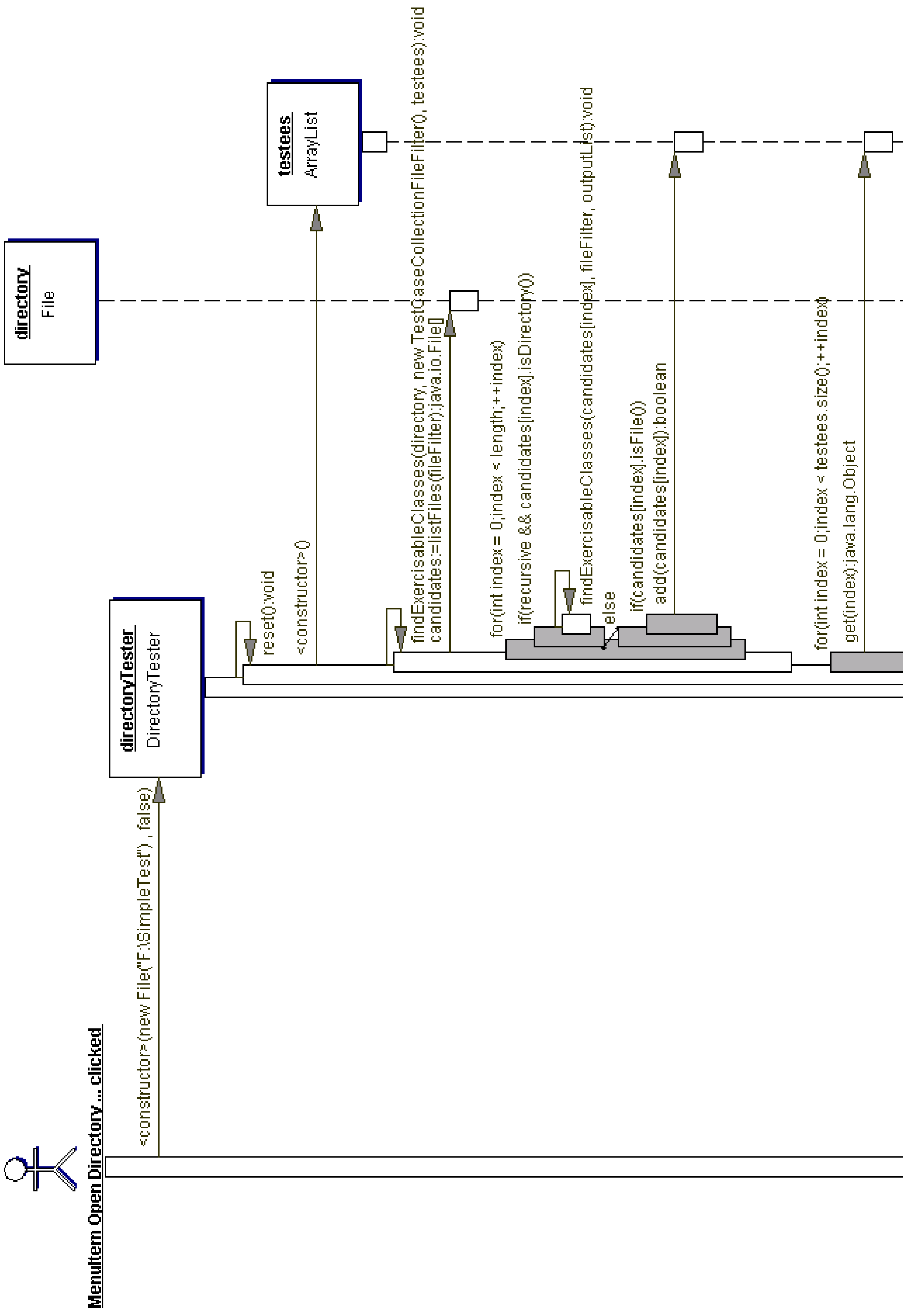

MenuItem Open Directory ... clicked
<constructor>(new File("F:\\SimpleTest"), false)
directoryTester
DirectoryTester
directory
File
testees
ArrayList
reset():void
<constructor>()
findExercisableClasses(directory, new TestCaseCollectionFileFilter(), testees):void
candidates:=listFiles(fileFilter):java.io.File[]
for(int index = 0;index < length;++index)
if(recursive && candidates[index].isDirectory())
findExercisableClasses(candidates[index], fileFilter, outputList):void
else
if(candidates[index].isFile())
add(candidates[index]):boolean
for(int index = 0;index < testees.size();++index)
get(index):java.lang.Object

Figure 13 Sequence diagram of a directory search for test cases, part 1

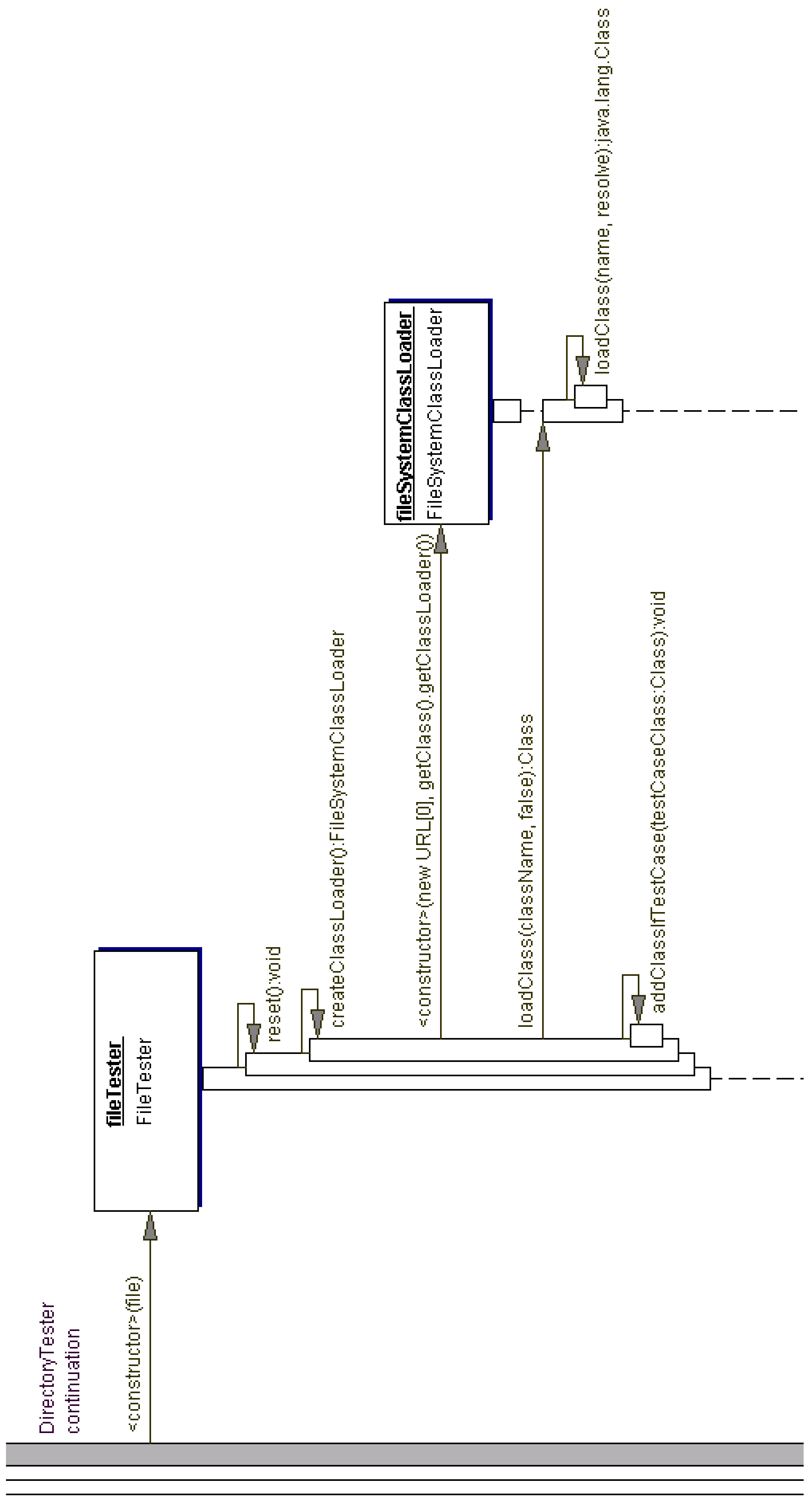

DirectoryTester continuation
<constructor>(file)
fileTester
FileTester
reset():void
createClassLoader():FileSystemClassLoader
<constructor>(new URL[0], getClass().getClassLoader())
loadClass(className, false):Class
addClassIfTestCase(testCaseClass:Class):void
fileSystemClassLoader
FileSystemClassLoader
loadClass(name, resolve):java.lang.Class

Figure 14 Sequence diagram of a directory search for test cases, part 2

7.4 Design of the GUI package

The last class diagram (Figure 15) shows the most import GUI classes of the TestSuite. `TestSuite` is the main frame for the GUI. It contains all menus and its items, for example the *Unit Tests* menu. One item of this menu is the *mru item*. *mru* is an acronym for *most recently used*. This works to achieve the effect found in many popular software products: keeping track of the items that a user has chosen most recently for easy retrieval. Together with an `ApplicationProperties` object, a `MruMenuManager` will load and store the `mru items` for persistence between client sessions.

An instance of `ReporterInternalFrame` graphically displays all the open test cases and the test results after its execution. Additionally it contains an *Re-Exercise* button to reload changed test cases and two radio buttons to filter failed tests.

A `ProgressInternalFrame` object shows information during the execution of the selected test cases, e.g. how many test cases are executed up to now, how many failed etc.

`ClassNameDialog`, `ClassFileFilter` and `DirectoryFilter` are classes to enter or select class, file or directory names. A `ClassNameDialog` gathers a class name from the user, a `ClassFileFilter` only displays class file names and a `DirectoryFilter` only displays directories in a file chooser dialog.

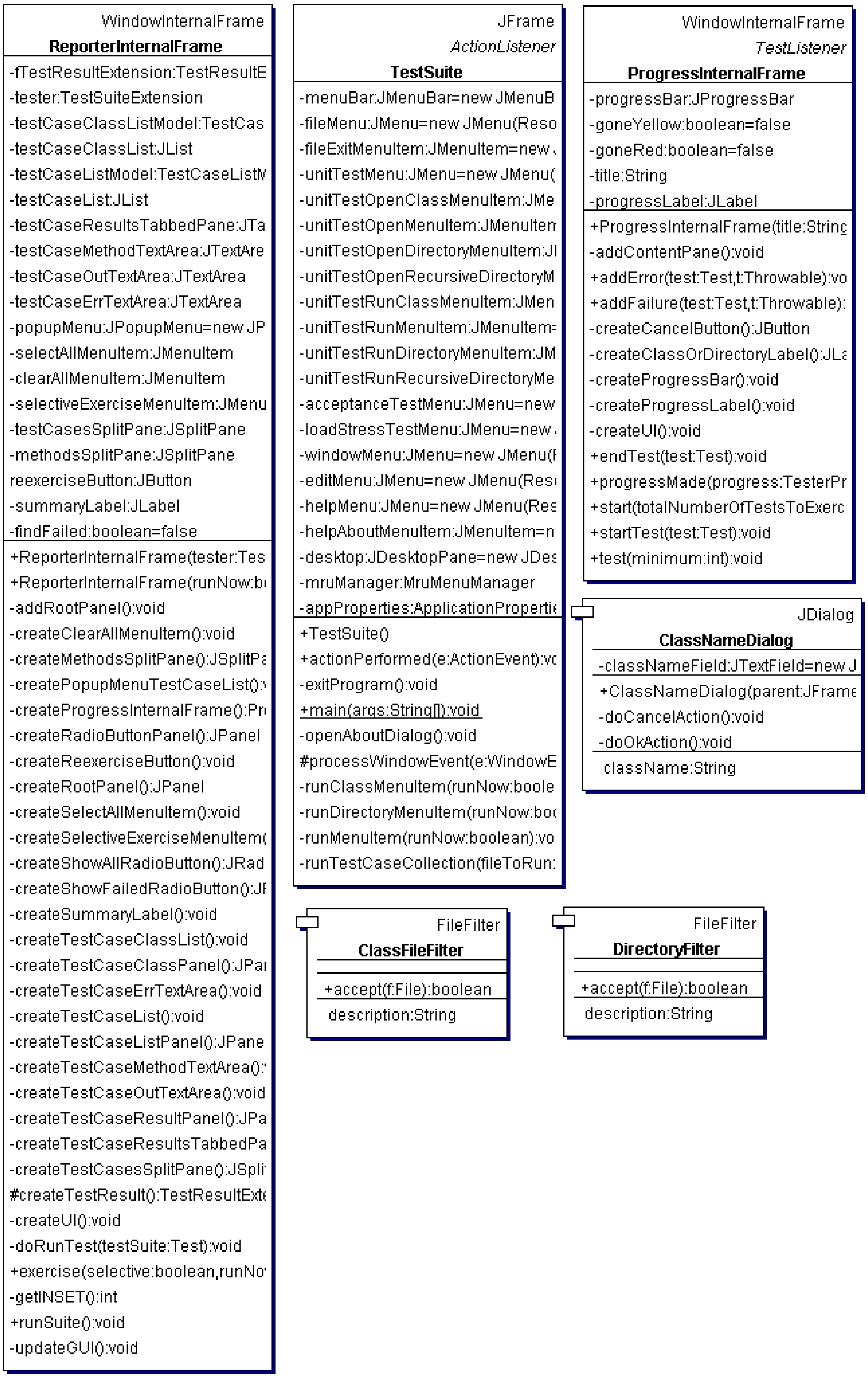

WindowInternalFrame
ReporterInternalFrame
-fTestResultExtension:TestResultE
-tester:TestSuiteExtension
-testCaseClassListModel:TestCas
-testCaseClassList:JList
-testCaseListModel:TestCaseListM
-testCaseList:JList
-testCaseResultsTabbedPane:JTa
-testCaseMethodTextArea:JTextAre
-testCaseOutTextArea:JTextArea
-testCaseErrTextArea:JTextArea
-popupMenu:JPopupMenu=new JP
-selectAllMenuItem:JMenuItem
-clearAllMenuItem:JMenuItem
-selectiveExerciseMenuItem:JMenu
-testCasesSplitPane:JSplitPane
-methodsSplitPane:JSplitPane
reexerciseButton:JButton
-summaryLabel:JLabel
-findFailed:boolean=false
+ReporterInternalFrame(tester:Tes
+ReporterInternalFrame(runNow:b
-addRootPanel():void
-createClearAllMenuItem():void
-createMethodsSplitPane():JSplitP
-createPopupMenuTestCaseList():
-createProgressInternalFrame():Pr
-createRadioButtonPanel():JPanel
-createReexerciseButton():void
-createRootPanel():JPanel
-createSelectAllMenuItem():void
-createSelectiveExerciseMenuItem(
-createShowAllRadioButton():JRad
-createShowFailedRadioButton():JR
-createSummaryLabel():void
-createTestCaseClassList():void
-createTestCaseClassPanel():JPa
-createTestCaseErrTextArea():void
-createTestCaseList():void
-createTestCaseListPanel():JPane
-createTestCaseMethodTextArea():
-createTestCaseOutTextArea():void
-createTestCaseResultPanel():JPa
-createTestCaseResultsTabbedPa
-createTestCasesSplitPane():JSpli
#createTestResult():TestResultExt
-createUI():void
-doRunTest(testSuite:Test):void
+exercise(selective:boolean,runNo
-getINSET():int
+runSuite():void
-updateGUI():void
JFrame
ActionListener
TestSuite
-menuBar:JMenuBar=new JMenuB
-fileMenu:JMenu=new JMenu(Reso
-fileExitMenuItem:JMenuItem=new
-unitTestMenu:JMenu=new JMenu(
-unitTestOpenClassMenuItem:JMe
-unitTestOpenMenuItem:JMenuItem
-unitTestOpenDirectoryMenuItem:J
-unitTestOpenRecursiveDirectoryM
-unitTestRunClassMenuItem:JMen
-unitTestRunMenuItem:JMenuItem=
-unitTestRunDirectoryMenuItem:JM
-unitTestRunRecursiveDirectoryMe
-acceptanceTestMenu:JMenu=new
-loadStressTestMenu:JMenu=new
-windowMenu:JMenu=new JMenu(
-editMenu:JMenu=new JMenu(Res
-helpMenu:JMenu=new JMenu(Res
-helpAboutMenuItem:JMenuItem=n
-desktop:JDesktopPane=new JDes
-mruManager:MruMenuManager
-appProperties:ApplicationProperti
+TestSuite()
+actionPerformed(e:ActionEvent):v
-exitProgram():void
+main(args:String[]):void
-openAboutDialog():void
#processWindowEvent(e:WindowE
-runClassMenuItem(runNow:boole
-runDirectoryMenuItem(runNow:bo
-runMenuItem(runNow:boolean):vo
-runTestCaseCollection(fileToRun:
WindowInternalFrame
TestListener
ProgressInternalFrame
-progressBar:JProgressBar
-goneYellow:boolean=false
-goneRed:boolean=false
-title:String
-progressLabel:JLabel
+ProgressInternalFrame(title:String
-addContentPane():void
+addError(test:Test,t:Throwable):vo
+addFailure(test:Test,t:Throwable):
-createCancelButton():JButton
-createClassOrDirectoryLabel():JL
-createProgressBar():void
-createProgressLabel():void
-createUI():void
+endTest(test:Test):void
+progressMade(progress:TesterPr
+start(totalNumberOfTestsToExerc
+startTest(test:Test):void
+test(minimum:int):void
JDialog
ClassNameDialog
-classNameField:JTextField=new J
+ClassNameDialog(parent:JFrame
-doCancelAction():void
-doOkAction():void
className:String
FileFilter
ClassFileFilter
+accept(f:File):boolean
description:String
FileFilter
DirectoryFilter
+accept(f:File):boolean
description:String

Figure 15 The most important classes of the package com.daimlerchrysler.testsuite.gui

8 Testing

This chapter describes the testing efforts which were made to ensure the quality of the product. Primarly two types of tests were made: unit and acceptance tests. Load/stress tests were not written and executed because the TestSuite is not a performance critical application and thus those tests do not make sense.

8.1 Unit Tests

For every class or component of the TestSuite unit tests were written. Normally they were written before the implementation of the “production” code for the TestSuite. This approach helped to understand how a class or a component could be used.

For example, `java.lang.String` is a powerful and convenient class but it can also lead the unwary into some unpleasant performance problems. For this reason a `MutableString` class was implemented based on some general rules (for further details see [Halter 2000], page 56).

To test this new class, first some test methods were written. The code snippet below shows 3 test methods of 41 altogether:

```
public class MutableStringTest extends TestCase {
   public void testAppendChar() {
      MutableString ms = new MutableString("hello, world");
      ms.append('x');
      assert(ms.equals("hello, worldx"));
   }
   public void testHashCode() {
      String s = "This is a longer string now";
      MutableString ms = new MutableString(s);
      assert(s.hashCode() == ms.hashCode());
   }
   public void testTrim() {
      MutableString ms = new MutableString("    hello, world     ");
      ms.trim();
      assert(ms.equals("hello, world"));
   }
   . . .
}
```

8.2 Acceptance Tests

Together with the customer the user stories were translated into one or more acceptance tests. Because of the low numbers of user stories the decision was taken, not to automate them. Instead of it the customer has to perform the tests manually at the GUI. For that purpose the customer has to read the test instructions and the expected output values from a spreadsheet.

The acceptance tests for user story three and four are shown in the table below:

User Story	Test No.	Instructions

3	1	1. Open the class `AcceptanceTest.java` of the package `com.daimlerchrysler.testsuite.tests` in an editor of your choice. The implementation of the class is:
		`public class AcceptanceTest extends TestCase {`
		`   public AcceptanceTest(String name) {`
		`      super(name);`
		`   }`
		`   public void testAssertTrue() {`
		`      assert(true);`
		`   }`
		`   public void testAssertFalse() {`
		`      assert(false);`
		`   }`
		`   public void testAssertionFailedError() {`
		`      throw new AssertionFailedError("Ooops");`
		`   }`
		`}`
		2. Compile the class and open it in the TestSuite.
		3. Press the “Re-Exercise” button.
		4. 1 of 3 test cases must pass.
		5. Change the method `testAssertFalse()` to:
		`   public void testAssertFalse() {`
		`      assert(true);`
		`   }`
		6. Compile the class and press the “Re-Exercise” button in the TestSuite.

User Story	Test No.	Instructions

		7. This time 2 of 3 test cases must pass.
		8. Finally change the method `testAssertFalse()` back to:
		`public void testAssertFalse() {`
		`assert(false);`
		`}`
		and compile the class.

4	1	1. Open the class `com.daimlerchrysler.testsuite.tests.AcceptanceTest` in the TestSuite. 2. Press the "Re-Exercise" button. 3. If the radio button "Show all" is selected, all the 3 test cases must be visible in the GUI. 4. If the radio button "Show failed" is selected, only 2 test cases (testAssertFalse and testAssertionFailedError) must be visible.

Table 2 Acceptance tests for user story three and four

8.3 Conclusions

Writing a unit test for every class or component of the TestSuite may sound nonsensical (especially since the tests were written before the production code), but here are some reasons why this approach was chosen:

- *The TestSuite contains no error*
  That of course is too nice to be true. One never knows whether the system still contains errors, after all, the tests might be wrong themselves. But chances are very high that there are less errors than without having tests.
  But what is known is that the features, which have been tested, work as expected.

- *"Micro"-desing improvements*
  This is one of the most important areas, besides reducing the number of bugs in the system.

The components in a system developed with unit tests are less dependent on each other, and the methods communicate much more error information, for example via exceptions. This is a result of the “caller“-perspective while writing the tests, instead of the “implementor“-perspective usually taken by developers.

- *Short feedback loop*

  In traditional development, functionality written by a developer is usually tested weeks later by the testing group, if there is one. Until a bug is found, so much code has been written in the meantime which could be responsible for the problem, that it takes a lot of time and effort for the developers to verify and fix the bug. With unit tests, unanticipated problems are detected as soon as the tests run the next time, which is usually only minutes after the change has been coded. The developer will still remember what changes he had made since the last time the tests were run successfully, so one of these changes has to be responsible for the error.

- *Improved changeability of the system*

  With unit tests in place, the system can be changed much more aggressively than without. There is a lot less need to worry whether the system will still be working after the change, because the unit tests will either work or highlight the point of failure immediately.

- *Regression testing*

  Unit tests help to verify that the code does what it is conceived to do anytime, and as often as needed. Since JUnit reports the elapsed time for the tests, it is possible to use that information to detect if a critical component is getting faster or slower. This will of course not replace performance testing.

- *Communicating design / documenting code*

  Unit tests document the “micro“ design of the system by showing how the code is intended to be used and which combinations of input are valid. It furthermore documents which error conditions are reported. When trying to understand a piece of code, one sometimes would like to step through the code in the debugger to see exactly what happens. But which feature does one need to invoke in a larger system so that a certain breakpoint will be reached ? It is much easier to just go into the test-class for the class one is interested in, and start stepping through the test cases.

Unfortunately testing GUI classes with unit tests is very difficult and sometimes even impossible. Therefore all the business logic was moved into non-visual classes which are unit-testable. The result is that the GUI classes contain only a minimum of untested code.

The acceptance tests were executed for two reasons: First they guarantee that the customers requirements have been met and second they help to test the code of the GUI classes.

9 Summary and further development

The thesis' main purpose has been to integrate JUnit into a GUI based TestSuite which makes it first easier to run and analyse unit tests and which is second expandable for other kinds of tests, in particular for acceptance and load/stress tests. The aim is that after another master thesis all the three types of tests of the Extreme Programming testing strategy are integrated within one TestSuite according to the requirements of DaimlerChrysler.

Most of the software systems developed at DaimlerChrysler are web applications for the Intranet and Internet. A typical web application is shown in Figure 16:

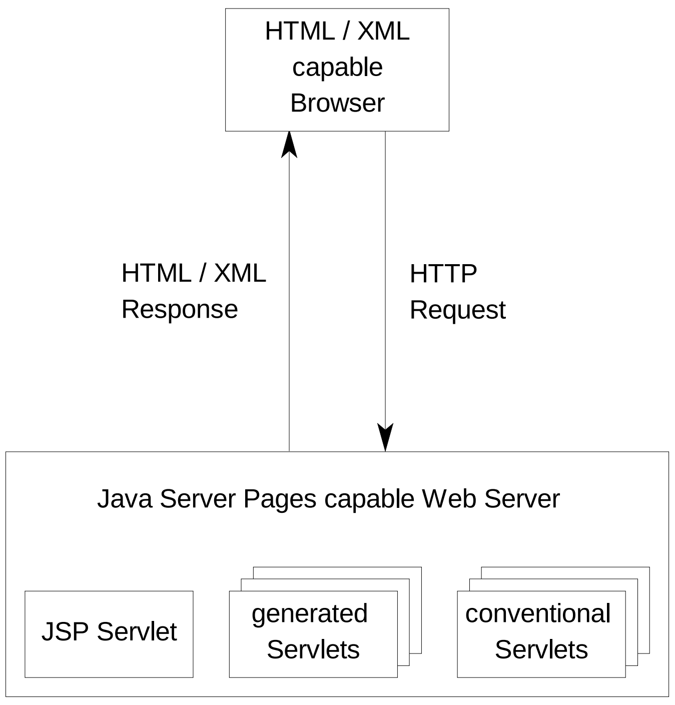


Figure 16 A typical web application

With the current release of the TestSuite the IT projects at DaimlerChrysler are able to test their business code in the servlets.

To run automatic acceptance tests, the following architecture of a "Capture and Playback" tool for the TestSuite is imaginable:
The tool consists of a request and response recorder and of a driver for the acceptance tests.

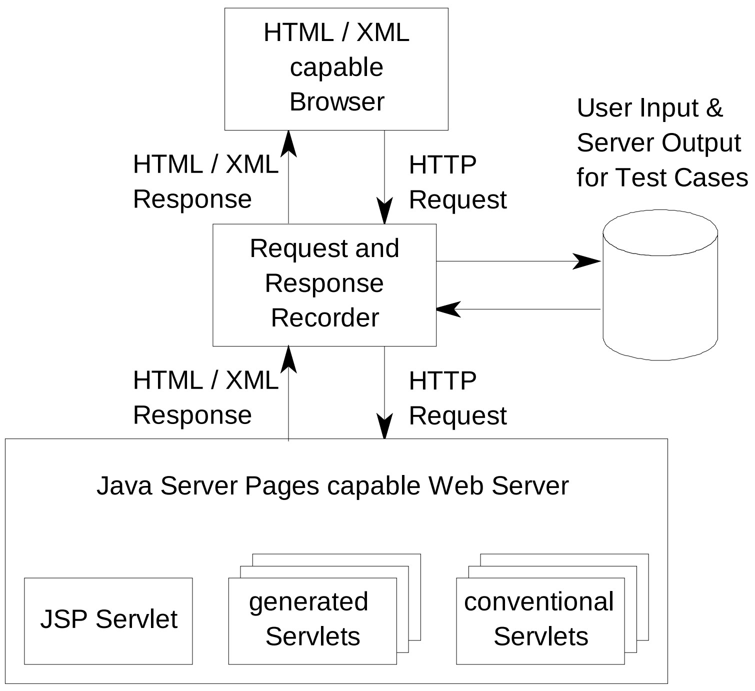

HTML / XML capable Browser
HTML / XML Response
HTTP Request
User Input & Server Output for Test Cases
Request and Response Recorder
HTML / XML Response
HTTP Request
Java Server Pages capable Web Server
JSP Servlet
generated Servlets
conventional Servlets

Figure 17 The request and response recorder

The customer enters its test values within the Web Browser and presses a *Submit button*. The recorder saves the user input and the generated server output. The customer can verify and correct the generated output, also a repetition of the test case is possible. If the functionality is not yet implemented in the server, the customer can edit the desired values into an editor and save them (see Figure 17).

With this recorder the customers can write, run, and maintain their own acceptance tests.

The other part of the tool consists of a driver (see Figure 18), with that the customers and the programmers can firstly run automatic acceptance tests several times a day like the unit tests and secondly execute load and stress tests.

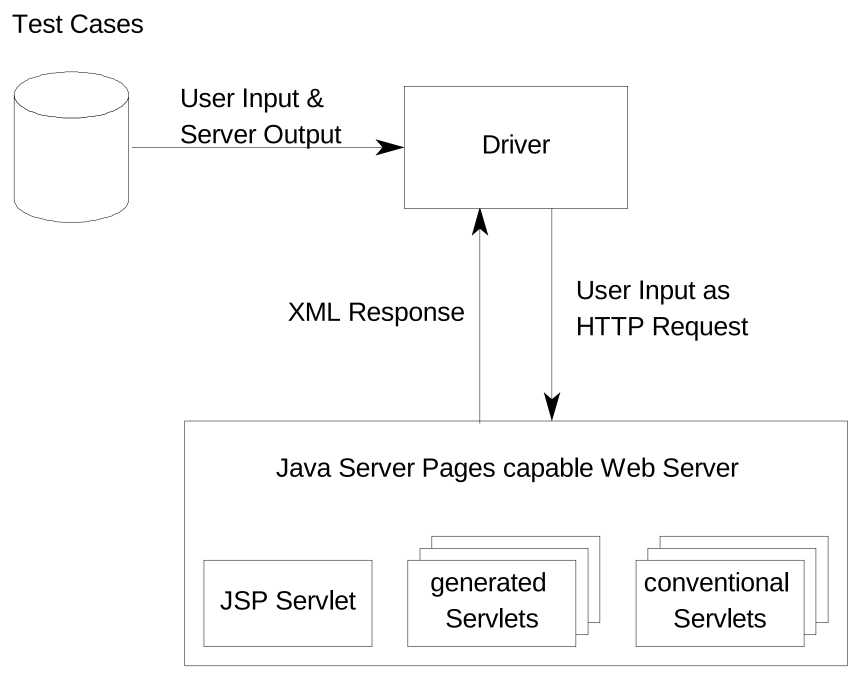


Figure 18 The driver for the acceptance tests

Thus the tasks for the next thesis would be the following:

- To analyse the requirements for the request and response recorder and for the driver.
- To evaluate commercial or free available “Capture and Playback“ tools resp. recorder/driver according to the analysed requirements.
- If none of these tools match the requirements to design and implement/test the recorder and the driver. The driver and the associated GUI must be designed in the manner that it is possible to execute load and stress tests.
- To integrate the recorder and the driver into the TestSuite.
- To update the documentation for the TestSuite.

10 References

[Beck 1996]	Beck, K. (1996) *Smalltalk Best Practice Patterns*, Prentice Hall
[Beck 1999]	Beck, K. (1999) *Extreme Programming explained: Embrace Change*, Addison Wesley Publishing Company
[Boehm 1988]	Boehm, B. W. (1988 May) *A Spiral Model of Software Development and Enhancement*, IEEE Computer
[Boehm 1994]	Boehm, B. W., Bose, P. (1994 August) *A Collaborative Spiral Software Process Model Based on Theory W.*, Third International Conference on the Software Process
[Fowler 1998]	Fowler, M., Kendall S. (1998) *UML konzentriert – Die neue Standard-Objektmodellierungssprache anwenden*, Addison Wesley Longman Verlag GmbH
[Goglia 1993]	Goglia, P. A. (1993) *Testing Client/Server Applications*, John Wiley & Sons
[Halter 2000]	Halter, S. L., Munroe, S. J. (2000) *Enterprise Java Performance*, Prentice Hall
[Höst 1996]	Höst, M. (1996) *Impact Analysis in Software Process Improvement.* [Technical Report No. 124]. Department of Communication Systems, Lund Institute of Technology, Lund
[Humphrey 1995]	Humphrey, W. S. (1995) *A Discipline for Software Engineering.* Addison-Wesley, Reading, Mass.

[Humphrey 2000]	Humphrey, W. S. (2000) *The Team Software Process (TSP).* [Technical Report CMU/SEI-2000-TR-023]. The Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Penn.
[JUnit]	Beck, K., Gamma, E., *JUnit, Unit testing framework for Java*, The testing framework for Java (and other programming languages) can be found under http://www.xprogramming.com/software.htm
[Link 2000]	Link, J. (2000), *eXtreme Testing – Der Kern von eXtreme Programming*, Lecture slides at the OBJEKTforum February 2000, Karlsruhe
[Paulk 1993]	Paulk, M. C., Curtis, B., Chrissis, M. B., Weber, C. V. (1993), *Capability Maturity Model for Software, Version 1.1.* CMU/SEI-93-TR-024, ESC-TR-93-177, Software Engineering Institute
[Rational 1998]	Rational Software Corporation (1998) *Rational Unified Process – Best Practices for Software Development Teams*, http://www.rational.com/products/rup/prodinfo/whitepapers/dynamic.jtmpl?doc_key=100420
[Sommerville 1995]	Sommerville, I. (1995) *Software Enginnering* (5th Edition), Harlow, England: Addison-Wesley
[Zahran 1998]	Zahran, S. (1998) *Software Process Improvement –* Practical Guidelines for Business Success, Harlow, England: Addison-Wesley

11 Bibliography

- Beck, K., Fowler, M. (2000) *Planning Extreme Programming*, Addison Wesley Publishing Company

- Binder, R. V. (1999) *Testing object-oriented systems: models, patterns, and tools*. Reading, Mass.: Addison-Wesley

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) *Pattern-Oriented Software Architecture. A System of Patterns.* New York: John Wiley & Sons.

- Cockburn, A. (1998) *Surviving Object-Oriented Projects*, Addison Wesley Publishing Company

- Fowler, M. et al. (1999 November) *Refactoring: Improving the Design of Existing Code*, Addison Wesley Publishing Company

- Gamma, E. et al. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Publishing Company

- Jacobson, I., Booch, G., Rumbaugh, J. (1998) *The Unified Software Development Process.* Reading, Mass.: Addison-Wesley

- Jeffries, R., Anderson, A., Hendrickson, C. (2000) *Extreme Programming Installed*, Addison Wesley Publishing Company

- Kaner, C., Falk, J., Nguyen, H. Q. (1999*) Testing Computer Software, Second Edition*, John Wiley & Sons

Appendix A Detailed Project Plan

1 Project Work Breakdown Structure

The development of the *TestSuite for Extreme Programming* has been broken down into two major phases, *TestSuite for Extreme Programming 1.0* and *TestSuite for Extreme Programming 2.0*. The first phase has been the task of this thesis and includes the following activity packages:

Requirements Definition

This activity package encompasses the initial meeting with the customer, followed up by a requirement definition Working Group Meeting. A signed requirements statement and an approved project proposal are milestones for this activity package.

Initial Planning

This activity package encompasses developing the Software Project Plan and writing the interim report.

User Stories Definition

Together with the customer the User Stories will be defined and estimated. Subsequently the decision will be taken, which User Stories go in this release.

Final Planning

Taking into account the work done for the previous activity packages, this phase will break the release stories into tasks and the tasks will be estimated. If there are any discrepencies between story estimates and task estimates they will be resolved (task estimates override story estimates). At the end of this activity package together with the customer a final decision will be taken about what is in this release and what is in each interration (a release will consist of 3-4 iterrations of 1-3 weeks each). The milestone for this activity package will be a signed Release Plan.

Design and Implementation

This activity package will include writing the unit tests, creating Design Diagrams and coding as outlined in the diagrams for each iteration. If the progress is faster or slower than expected, the customer will immediately be notified, so that s/he can make appropriate scope decisions. At the a code and design review with the customer will be conducted. A signed Design Specification will be the milestone of this activity package.

Testing and Documentation

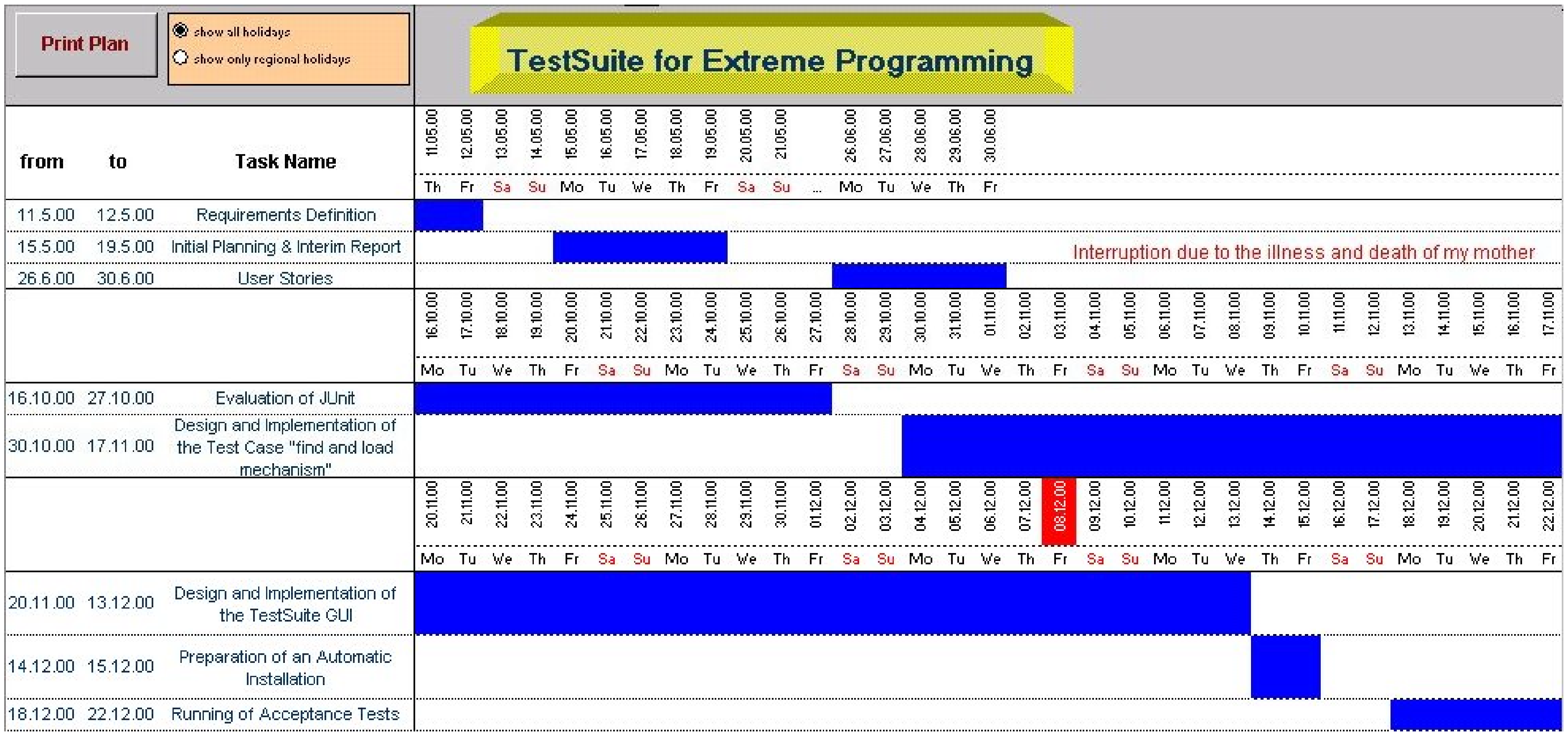
Working concurrently, the customer runs acceptance tests on the completed user stories and the thesis report including the Users Guide will be written.

Demonstration

This activity package will be the culmination of all the analysis, design, and implementation of the first phase of the *TestSuite for Extreme Programming* effort. A successful demonstration and the approval of the thesis by the project advisor Dr. Dear will close the first phase and will give the green light to proceed with *TestSuite for Extreme Programming 2.0.* The content of this second phase will be the integration of tools for running and analysing acceptance and load/stress tests. The realization will be the part of another master thesis.

2 Project time table

The project time table in the Interim Report doesn’t correspond with the following time table because of the illness and death of my mother.

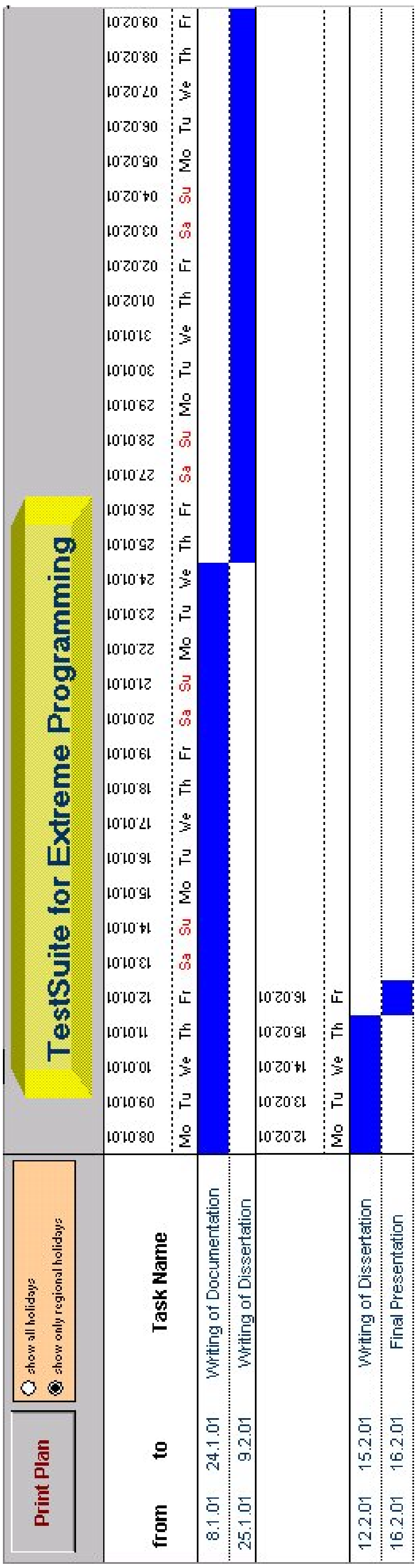


[✓ show all holidays] show only regional holidays

from	to	Task Name
11.5.00	12.5.00	Requirements Definition
15.5.00	19.5.00	Initial Planning & Interim Report
26.6.00	30.6.00	User Stories
16.10.00	27.10.00	Evaluation of JUnit
30.10.00	17.11.00	Design and Implementation of the Test Case "find and load mechanism"
20.11.00	13.12.00	Design and Implementation of the TestSuite GUI
14.12.00	15.12.00	Preparation of an Automatic Installation
18.12.00	22.12.00	Running of Acceptance Tests

Interruption due to the illness and death of my mother

Figure 19 Project time table part 1


Print Plan
show all holidays
show only regional holidays
TestSuite for Extreme Programming

from	to	Task Name	08.01.01	09.01.01	10.01.01	11.01.01	12.01.01	13.01.01	14.01.01	15.01.01	16.01.01	17.01.01	18.01.01	19.01.01	20.01.01	21.01.01	22.01.01	23.01.01	24.01.01	25.01.01	26.01.01	27.01.01	28.01.01	29.01.01	30.01.01	31.01.01	01.02.01	02.02.01	03.02.01	04.02.01	05.02.01	06.02.01	07.02.01	08.02.01	09.02.01
			Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr
8.1.01	24.1.01	Writing of Documentation	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■																
25.1.01	9.2.01	Writing of Dissertation																		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■

from	to	Task Name	12.02.01	13.02.01	14.02.01	15.02.01	16.02.01
			Mo	Tu	We	Th	Fr
12.2.01	15.2.01	Writing of Dissertation	■	■	■	■	
16.2.01	16.2.01	Final Presentation					■

Figure 20 Project time table part 2

The task implementation consists of writing unit tests and writing production code.

Appendix B Users Guide

1 Introduction

This document explaines the process of creating a unit test for Java classes, running it within the *TestSuite*, and then displaying the results. Additionally it provides information about the system and the hardware requirements, the installation of the *TestSuite* and finally it contains a chapter with tips and tricks.

The GUI of the *TestSuite* is already prepared for creating and running acceptance tests and load/stress tests. These enhancements will be implemented in a next version.

2 System and Hardware Requirements

Any platform that supports Java Virtual Machine 1.2.

The *TestSuite* has been successfully tested with the following setups:

- Windows NT, Solaris and Linux
    - Sun JDK 1.2.2

The *TestSuite* has been successfully tested on the minimal configuration

- PC with 133 Intel Pentium processor
- Memory 32 MB

3 Installation

The first thing to do before installing the *TestSuite*, is to ensure that the system includes a Java Virtual Machine, i.e. that it can run Java byte code. This can be checked by typing

```
java –version
```

at the command prompt. If the response isn't something like

```
java version "JDK1.2.2"
```

java support should be added to the system before trying to run the *TestSuite*. This is done by installing the JDK (Java Development Kit), which includes the JRE (Java Runtime Environment). Sun Microsystem's can be downloaded for free from Javasoft (the URL is quoted in the Resources section). Be sure to get a version supporting Java 1.2, i.e. JDK 1.2 or a later version.

There are also many Java Development Kits and Virtual Machines available from different vendors. Some of them are commercial and some are freely distributable. Any system that claims to be compatible with Java 1.2 or later should work with the *TestSuite*.

After having ensured that there is support for Java, the installation of the *TestSuite* can commence.

The *TestSuite* is delivered as a single JAR (Java Archive) file including all the *TestSuite* java classes as well as installation scripts for Windows and UNIX. Choose the directory you wish to install the *TestSuite* to and move the `testsuite.jar` archive to that directory. Several files and subdirectories will be created to main installation directory, so you probably want to create a new, clear, directory called testsuite.

In that directory, type

```
      jar -cvf testsuite.jar winInstall
```

if you are installing the *TestSuite* on a Windows 95 or Windows NT system, and type

```
      jar -cvf testsuite.jar unixInstall
```

if you are installing to a Unix machine. This command extracts the installation script for your system. After this you simply type

```
      winInstall
```

or

```
      unixInstall
```

which automatically installs the *TestSuite* on your computer. The PATH and CLASSPATH environment variables, required by the Java Runtime Environment will automatically be modified, and thus the only thing you have to do after this is to type

```
      testsuite
```

to start the *TestSuite*.

4 How to write a unit test

The centre of the *TestSuite* is *JUnit*, which is an open-source testing framework for Java written by Erich Gamma and Kent Beck.

Any class that contains unit tests must be a subclass of `junit.framework.TestCase`. A test-method is public, starts with the word "test" and takes no parameters, for example `public void testRead()`. The basic concept is to compare results in a way that a successful comparison returns true, and an unsuccessful false, instead of manual verification of the output ot the tests, e.g. looking at the output of `System.out.println()`-statements. For example if a result value is expected to 5, it could be written as `assertEquals(5, resultValue)`. This assert-method compares both parameters on equality. If `resultValue` has not the value 5 during execution of the test, the comparison will return false, the test-method is aborted and the next test-method is executed. The *TestSuite* will display one or more messages describing the failure. The class `TestCase` contains various methods to verify results, e.g. `assert(boolean)`, `assertEquals(Object, Object)` etc.

The execution scheme of a test-method in *JUnit* is typically as follows:

```
loop over all test-methods{
 setUp();   // template-method, executed before each test-method
 runTheTest();   // execute the test-method, starting with ”test”
 tearDown();   // template-method, guranteed to run after each test-
                  method
}
```

A test-class may contain a `setUp()` and a `tearDown()` method, plus any number of test-methods. If a test fails, the execution of the test-method stops immediately, but `tearDown()` will be executed in any case. A test-method typically tests one or more methods of a class of the production code.

As an example, let's think about a simple telephone number object. What behavior should it have? At a minimum, it will have accessors for the components of a German number: country code, area code and extension. We should also have a formatter that gives a nice string representation.

The first important question is now: What do we have to test? The rule of thumb is: If it can possibly fail, write a test for it. There are some exceptions to this rule, however. For example, since simple accessor methods rarely fail, they are rarely tested. Of course, if your accessor does more than just set an instance variable, you should write a test for it. Choosing precisely what to test takes some practice.
Here is a test method for the string formatting:

```
import junit.framework.*;

public class TelephoneNumberTests extends TestCase {
   public void testSimpleStringFormatting() throws Exception {
      // Build a complete phone number
      TelephoneNumber number = new TelephoneNumber("49", "7022",
                                                   "907770");
      assertEquals("Bad string", "+49-7022-907770",
                  number.formatNumber());
   }
}
```

The `assertEquals()` method takes a message string, the expected value, and the actual value. Internally, if `expected.equals(actual)` returns false, the assertion fails.

This method verifies the basic case, but we can think of many ways for string formatting to fail. What happens if any or all of the parts are null? `toString()` must do something reasonable. Let's say the country code is optional, but the area code and the extension must be present or a `NullPointerException` will be thrown.
Here are the new test methods:

```
public void testNullCountryCode() throws Exception {
   // Build a phone number without country code
   TelephoneNumber number = new TelephoneNumber(null, "7022",
                                                "907770");
   assertEquals("Bad string", "7022-907770", number.toString());
}
public void testNullAreaCode() throws Exception {
   // Build a phone number without area code
   TelephoneNumber number = new TelephoneNumber("49", null,
                                                "907770");
   try {
      number.toString();
      assert("Should have thrown a NullPointerException", false);
   } catch(NullPointerException npe) {
      // expected behavior
   }
}
public void testNullExtension() throws Exception {
   // Build a phone number without extension
   TelephoneNumber number = new TelephoneNumber("49", "7022", null);
   try {
      number.toString();
      assert("Should have thrown a NullPointerException", false);
   } catch(NullPointerException npe) {
      // expected behavior
   }
}
```

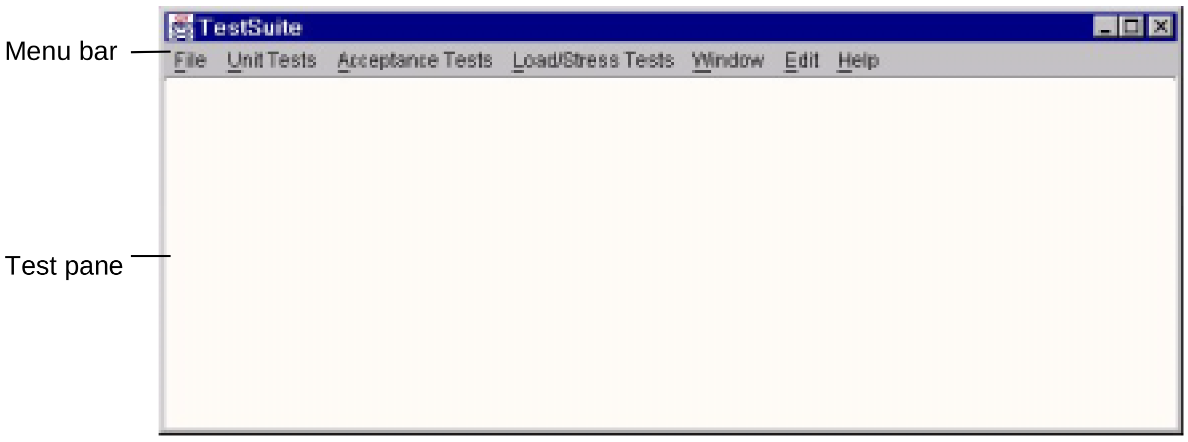
These tests define the behavior of the `TelephoneNumber`. You can probably come up with more ways to break the object, as it's pretty loosely defined right now.

5 The TestSuite

Once your test cases are written and compiled, you're ready to run them. Several seconds after starting the *TestSuite*, the main window is displayed on your desktop.

The *TestSuite* main window contains the following key elements:

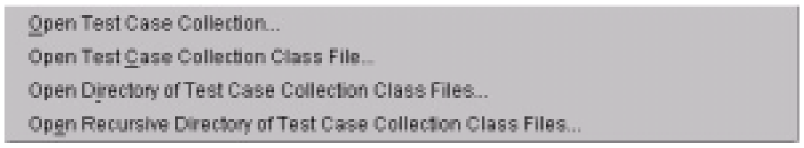
- *Menu bar*, displaying menus of the *TestSuite* commands
- *Test pane,* displaying tests and results once a test is selected



5.1 Opening one or more test case collections

In order to open one or more test case collections (a test case collection is a synonym for a subclass of `junit.framework.TestCase`), you have 4 different possibilities.

Select *Unit Tests* and the following menu opens:

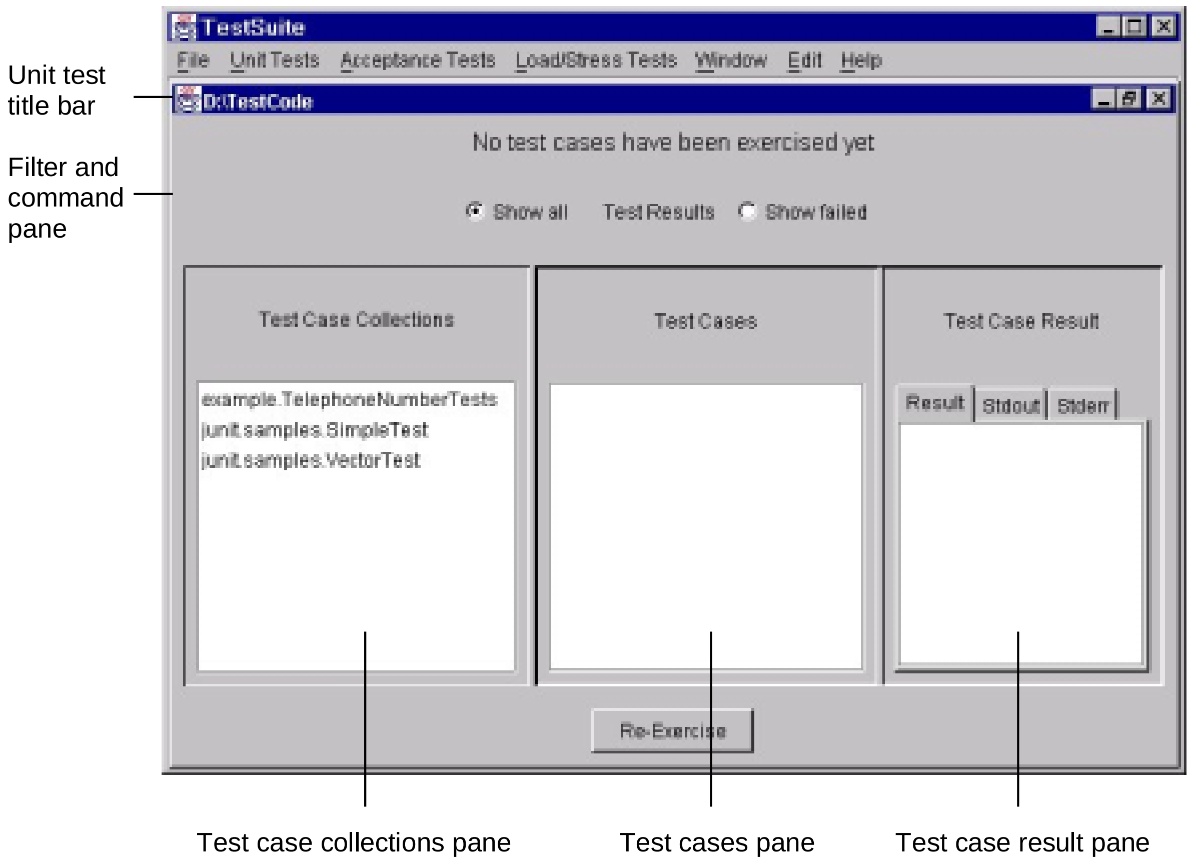

Open Test Case Collection...
Open Test Case Collection Class File...
Open Directory of Test Case Collection Class Files...
Open Recursive Directory of Test Case Collection Class Files...

Command	Description
Open Test Case Collection...	Opens a test case collection after entering the package and the class name. This command is necessary to open a test case collection within a Java IDE (e.g. Visual Age for Java).
Open Test Case Collection Class File...	Opens a test case collection after selecting a class in a file chooser dialog.
Open Directory of Test Case Collection Class Files...	Opens all test case collections in a directory after selecting the directory in a file chooser dialog.
Open Recursive Directory of Test Case Collection Class Files...	Opens all test case collections in a directory and the subdirectories after selecting the root directory in a file chooser dialog.

After choosing one or more test case collections a *Test pane* opens within the main window.

It contains the following key elements:

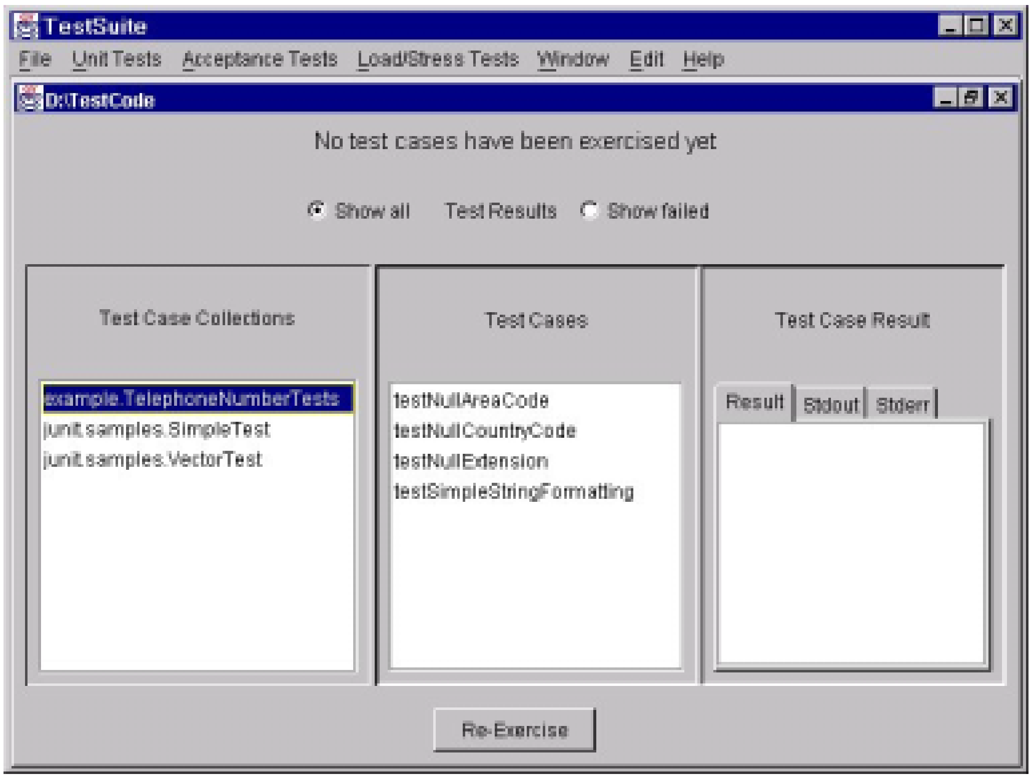
- *Unit test title bar*, displaying the name of the currently open test case collection or the name of the directory where the test case collections are placed
- *Filter and command pane,* containing buttons to filter and exercise tests
- *Test case collections pane,* displaying the open test case collections
- *Test cases pane,* displaying the available test cases (a test case is a synonym for a test-method)
- *Test case result pane,* containing 3 tabs to view the test case results:
  - Result: contains the results of JUnit
  - Stdout: contains output data typically printed to the standard console
  - Stderr: contains error data typically printed to the standard console



No test cases have been exercised yet

example.TelephoneNumberTests
junit.samples.SimpleTest
junit.samples.VectorTest

If you want to see the available test cases, select one or more test case collections in the left pane as shown in the figure below.

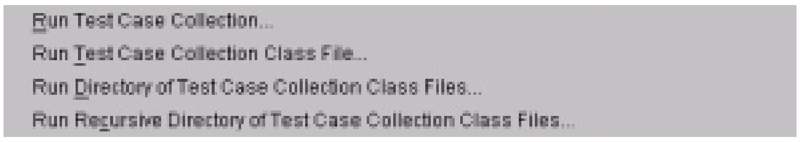


5.2 Running one or more test cases or whole test case collections

In order to run one or more test case collections, you can either run them directly or you can open the collections and run them afterwards.

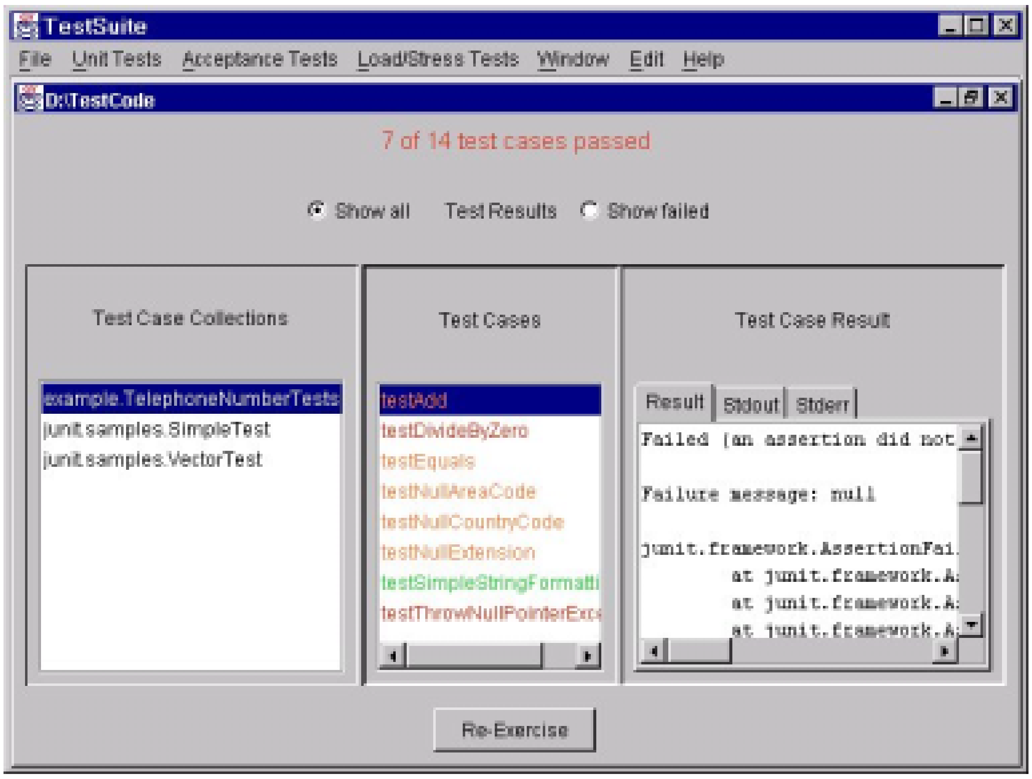
5.2.1 Running test case collections directly

To run one or more test case collections directly, select *Unit Tests* and the following menu opens:


Run Test Case Collection...
Run Test Case Collection Class File...
Run Directory of Test Case Collection Class Files...
Run Recursive Directory of Test Case Collection Class Files...

Command	Description
Run Test Case Collection...	Runs a test case collection after entering the package and the class name. This command is necessary to run a test case collection within a Java IDE (e.g. Visual Age for Java).
Run Test Case Collection Class File...	Runs a test case collection after selecting a class in a file chooser dialog.
Run Directory of Test Case Collection Class Files...	Runs all test case collections in a directory after selecting the directory in a file chooser dialog.
Run Recursive Directory of Test Case Collection Class Files...	Runs all test case collections in a directory and the subdirectories after selecting the root directory in a file chooser dialog.

After choosing one of the 4 possibilities, the test cases are executed and the results are displayed in the *Test case result pane*.



7 of 14 test cases passed

example.TelephoneNumberTests
junit.samples.SimpleTest
junit.samples.VectorTest

testAdd
testDivideByZero
testEquals
testNullAreaCode
testNullCountryCode
testNullExtension
testSimpleStringFormatti
testThrowNullPointerExc

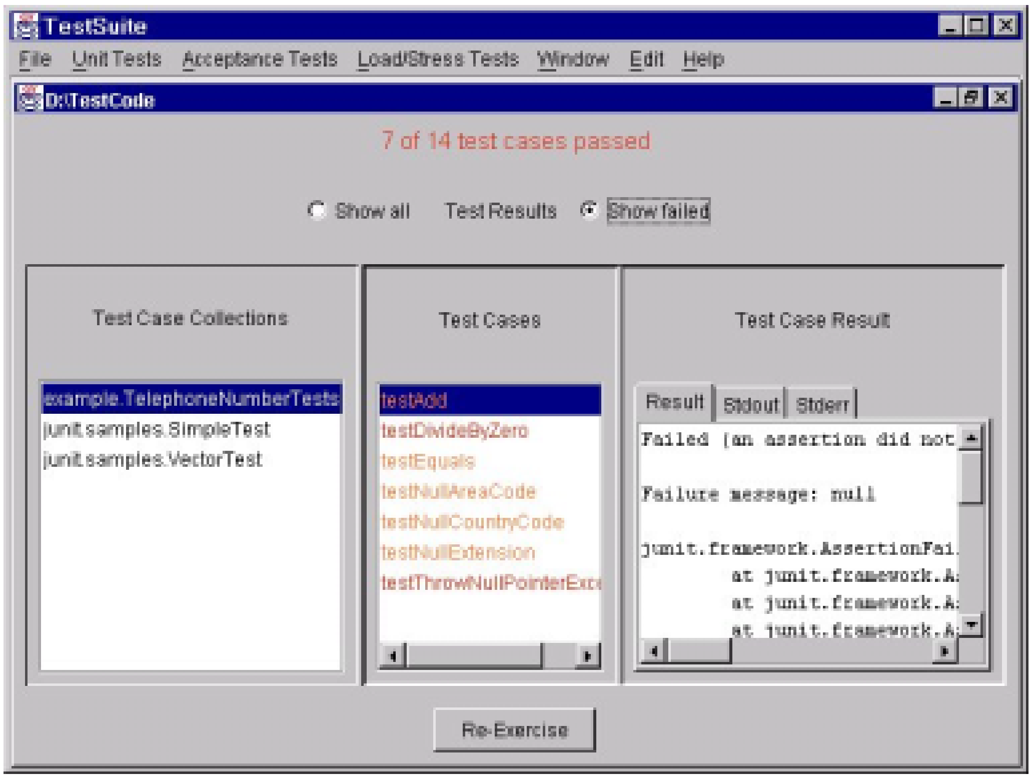
```
Failed (an assertion did not

Failure message: null

junit.framework.AssertionFai
        at junit.framework.A:
        at junit.framework.A:
        at junit.framework.A:
```

As you can see in the *Filter and command pane*, 7 of 14 test cases passed, the other failed. The test cases which passed are green, they which failed are orange or red. A test case is orange, if the expected and the real value are different. If the test case throws an exception, then it is read.

Additionally the passed test cases can be hidden.



7 of 14 test cases passed

example.TelephoneNumberTests
junit.samples.SimpleTest
junit.samples.VectorTest

testAdd
testDivideByZero
testEquals
testNullAreaCode
testNullCountryCode
testNullExtension
testThrowNullPointerExc

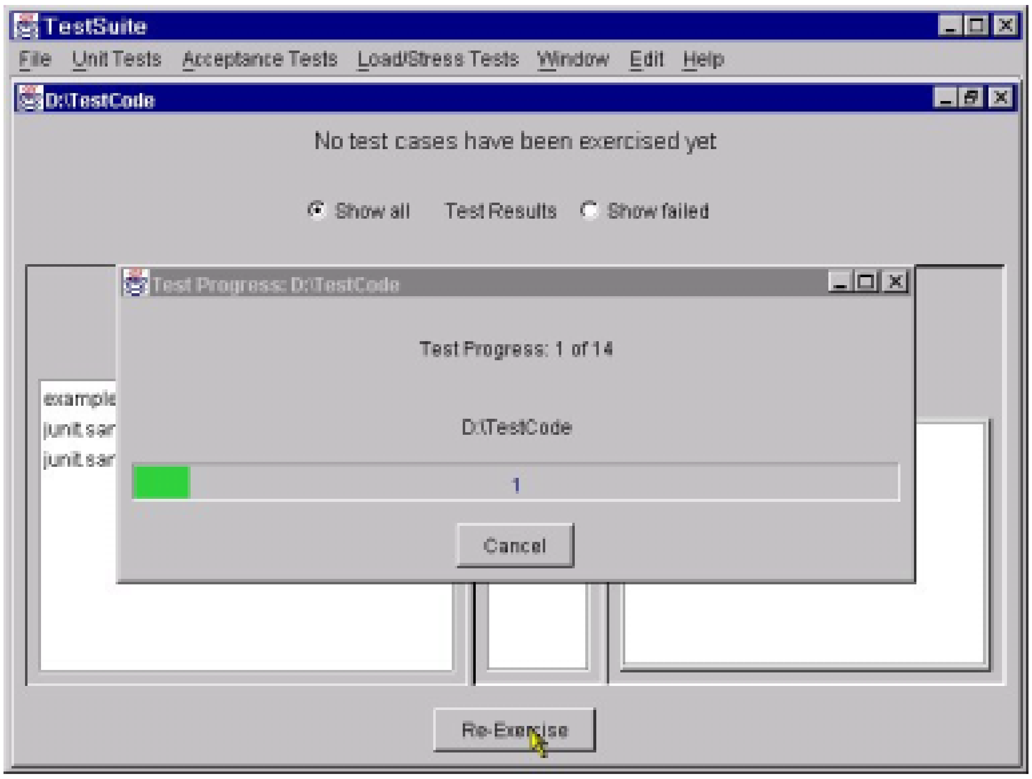
```
Failed (an assertion did not

Failure message: null

junit.framework.AssertionFai
        at junit.framework.A
        at junit.framework.A
        at junit.framework.A
```

5.2.2 Run test cases after opening them

You can also run test cases after opening them (for more information about opening test case collections, see chapter 5.1). In this case it is possible to run either all open test case collections or only a subset of open test cases.
To run all open test case collections, click the *Re-Exercise* button of the *Test pane*. Afterwards all open test case collections are reloaded and exercised.

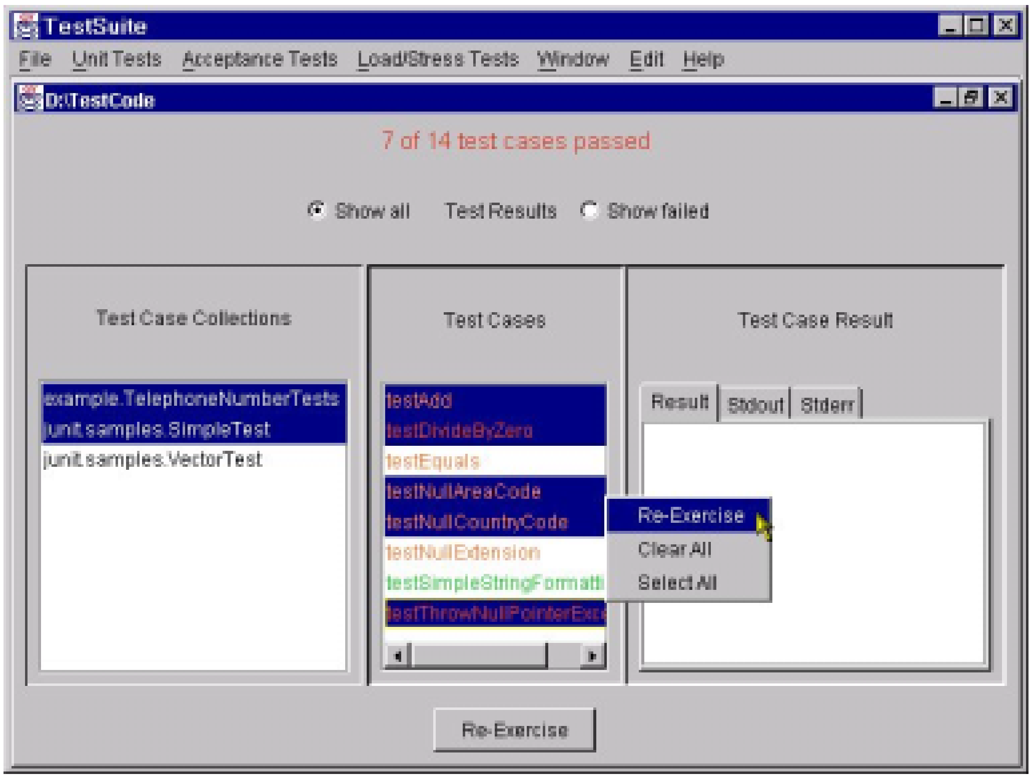


No test cases have been exercised yet

Test Progress: 1 of 14

D:\TestCode

To run only a subset of all open test cases, select the test cases and press the right mouse button. Then choose *Re-Exercise* in the menu and the selected test cases are reloaded and exercised.



7 of 14 test cases passed

example.TelephoneNumberTests
junit.samples.SimpleTest
junit.samples.VectorTest

testAdd
testDivideByZero
testEquals
testNullAreaCode
testNullCountryCode
testNullExtension
testSimpleStringFormatti
testThrowNullPointerExc

6 Tips and Tricks

Like any toolkit, the *TestSuite* and the integrated testing framework *JUnit* can be used effectively and ineffectively. This chapter discusses good and bad ways to use them and provides practical recommendations for its use by development teams.

Name tests properly

Name the test case `MyClassTest` or `MyClassTests`. For example, the test case for the class `TelephoneNumber` should be `TelephoneNumberTest` or `TelephonNumberTests`. That makes it simple to work out what class a test case tests. Test methods' names within the test case should describe what they test:

```
testSimpleStringFormatting()
testNullCountryCode()
testNullAreaCode()
testNullExtension()
```

Proper naming helps code readers understand each test's purpose.

Document tests in javadoc

Test plans documented in a word processor tend to be error-prone and tedious to create. Also, word-processor-based documentation must be kept synchronized with the unit tests, adding another layer of complexity to the process. If possible, a better solution would be to include the test plans in the tests' `javadoc`, ensuring that all test plan data reside in one place.

Avoid visual inspection

Testing user interfaces, and other systems that produce complex output is often left to visual inspection. Visual inspection -- a human inspecting output data for errors -- requires patience, the ability to process large quantities of information, and great attention to detail: attributes not often found in the average human being. Below are some basic techniques that will help reduce the visual inspection component of your test cycle.

Swing:

When testing a Swing-based UI, you can write tests to ensure that:

- All the components reside in the correct panels.
- You've configured the layout managers correctly.
- Text widgets have the correct fonts

A more thorough treatment of this can be found in the worked example of testing a GUI, referenced in the Resources section.

XML:

When testing classes that process XML, it pays to write a routine that compares two XML DOMs for equality. You can then programmatically define the correct DOM in advance and compare it with the actual output from your processing methods.

Keep tests in the same location as the source code

If the test source is kept in the same location as the tested classes, both test and class will compile during a build. This forces you to keep the tests and classes synchronized during development. Indeed, unit tests not considered part of the normal build quickly become dated and useless.

Ensure that tests are time-independent

Where possible, avoid using data that may expire; such data should be either manually or programmatically refreshed. It is often simpler to instrument the class under test, with a mechanism for changing ist notion of today. The test can then operate in a time-independent manner without having to refresh the data.

Consider locale when writing tests

Consider a test that uses dates. One approach to creating dates would be:

```
Date date = DateFormat.getInstance ().parse ("dd/mm/yyyy");
```

Unfortunately, that code doesn't work on a machine with a different locale. Therefore, it would be far better to write:

```
Calendar cal = Calendar.getInstance ();
Cal.set (yyyy, mm-1, dd);
Date date = Calendar.getTime ();
```

The second approach is far more resilient to locale changes.

Avoid writing test cases with side effects

Test cases that have side effects exhibit two problems:

- They can affect data that other test cases rely upon
- You cannot repeat tests without manual intervention

In the first situation, the individual test case may operate correctly. However, if incorporated into the TestSuite that runs every test case on the system, it may cause other test cases to fail. That failure mode can be difficult to diagnose, and the error may be located far from the test failure.

In the second situation, a test case may have updated some system state so that it cannot run again without manual intervention, which may consist of deleting test data from the database (for example). Think carefully before introducing manual intervention. First, the manual intervention will need to be documented. Second, the tests could no longer be run in an unattended mode, removing your ability to run tests overnight or as part of some automated periodic test run.

Do not use the test-case constructor to set up a test case

Setting up a test case in the constructor is not a good idea. Consider:

```
public class TelephoneNumberTests extends TestCase
   public TelephoneNumberTests (String testName) {
      super (testName);
      // Perform test set-up
   }
}
```

Imagine that while performing the setup, the setup code throws an `IllegalStateException`. In response, JUnit would throw an `AssertionFailedError`, indicating that the test case could not be instantiated. Here is an example of the resulting stack trace:

```
junit.framework.AssertionFailedError: Cannot instantiate test case:
                                       testNullAreaCode
```

```
at junit.framework.Assert.fail(Assert.java)
at junit.framework.TestSuite$1.runTest(TestSuite.java)
at junit.framework.TestCase.runBare(TestCase.java)
at junit.framework.TestResult$1.protect(TestResult.java)
at com.daimlerchrysler.testsuite.TestResultExtension.
                       runProtected(TestResultExtension.java)
at junit.framework.TestResult.run(TestResult.java)
at junit.framework.TestCase.run(TestCase.java)
at junit.framework.TestSuite.run(TestSuite.java, Compiled Code)
at com.daimlerchrysler.testsuite.gui.ReporterInternalFrame$
                       TesterThread.run(ReporterInternalFrame.java)
```

This stack trace proves rather uninformative; it only indicates that the test case could not be instantiated. It doesn't detail the original error's location or place of origin. This lack of information makes it hard to deduce the exception's underlying cause.

Instead of setting up the data in the constructor, perform test setup by overriding `setUp()`. Any exception thrown within `setUp()` is reported correctly. Compare this stack trace with the previous example:

```
java.lang.IllegalStateException: Oops
at example.TelephoneNumberTests.setUp(TelephoneNumberTests.java)
at junit.framework.TestCase.runBare(TestCase.java)
at junit.framework.TestResult$1.protect(TestResult.java)
at com.daimlerchrysler.testsuite.TestResultExtension.
                       runProtected(TestResultExtension.java)
at junit.framework.TestResult.run(TestResult.java)
...
```

This stack trace is much more informative; it shows which exception was thrown (IllegalStateException) and from where. That makes it far easier to explain the test setup's failure.

Don't assume the order in which tests within a test case run

You should not assume that tests will be called in any particular order. Consider the following code segment:

```
public class TelephoneNumberTests extends TestCase {
   public TelephoneNumberTests(String testName) {
      super (testName);
   }
   public void testDoThisFirst () {
      ...
   }
   public void testDoThisSecond () {
   }
}
```

In this example, it is not certain that JUnit will run these tests in any specific order when using reflection. Running the tests on different platforms and Java VMs may therefore yield different results, unless your tests are designed to run in any order. Avoiding temporal coupling will make the test case more robust, since changes in the order will not affect other tests. If the tests are coupled, the errors that result from a minor update may prove difficult to find.

Call a superclass's setUp() and tearDown() methods when subclassing

When you consider:

```
public class TelephoneNumberDatabaseTests extends
TelephoneNumberTests{
   // A connection to a database
   private Database theDatabase;
   public TelephoneNumberDatabaseTests (String testName) {
      super (testName);
   }
   public void testSimpleStringFormatting () {
      ...
   }
   public void setUp () {
      // Clear out the database
      theDatabase.clear ();
   }
}
```

Can you spot the deliberate mistake? `setUp()` should call `super.setUp()` to ensure that the environment defined in `TelephoneNumberTests` initializes. Of course, there are exceptions: if you design the base class to work with arbitrary test data, there won't be a problem.

Do not load data from hard-coded locations on a filesystem

Tests often need to load data from some location in the filesystem. Consider the following:

```
public void setUp () {
   FileInputStream inp ("C:\\TestData\\dataSet1.dat");
   ...
}
```

The code above relies on the data set being in the `C:\TestData` path. That assumption is incorrect in two situations:

- A tester does not have room to store the test data on `C:` and stores it on another disk .
- The tests run on another platform, such as Unix.

One solution might be:

```
public void setUp () {
   FileInputStream inp ("dataSet1.dat");
   ...
}
```

However, that solution depends on the test running from the same directory as the test data. If several different test cases assume this, it is difficult to integrate them into the TestSuite without continually changing the current directory.

To solve the problem, access the dataset using either `Class.getResource()` or `Class.getResourceAsStream()`. Using them, however, means that resources load from a location relative to the class's origin.

Test data should, if possible, be stored with the source code in a configuration management (CM) system. However, if you're using the aforementioned resource mechanism, you'll need to write a script that moves all the test data from the CM system into the classpath of the system under test. A less ungainly approach is to store the test data in the source tree along with the source files. With this approach, you need a location-independent mechanism to locate the test data within the source tree. One such mechanism is a class. If a class can be mapped to a specific source directory, you could write code like this:

```
InputStream    inp    =    SourceResourceLoader.getResourceAsStream
(this.getClass(), "dataSet1.dat");
```

Now you must only determine how to map from a class to the directory that contains the relevant source file. You can identify the root of the source tree (assuming it has a single root) by a system property. The class's package name can then identify the directory where the source file lies. The resource loads from that directory. For Unix and NT, the mapping is straightforward: replace every instance of '.' with `File.separatorChar`.

Utilize JUnit's assert/fail methods and exception handling for clean test code

Many JUnit novices make the mistake of generating elaborate try and catch blocks to catch unexpected exceptions and flag a test failure. Here is a trivial example of this:

```
public void exampleTest() {
   try {
      // do some test
   } catch (SomeApplicationException e) {
      fail ("Caught SomeApplicationException exception");
   }
}
```

JUnit automatically catches exceptions. It considers uncaught exceptions to be errors, which means the above example has redundant code in it.

Here's a far simpler way to achieve the same result:

```
public void exampleTest () throws SomeApplicationException {
   // do some test
}
```

In this example, the redundant code has been removed, making the test easier to read and maintain (since there is less code).

Use the wide variety of assert methods to express your intention in a simpler fashion. Instead of writing:

```
assert("+49-7022-907770" == number.formatNumber());
```

Write:

```
assertEquals("Bad string", "+49-7022-907770",
                            number.formatNumber());
```

The above example is much more useful to a code reader. And if the assertion fails, it provides the tester with more information. JUnit also supports floating point comparisons:

```
assertEquals ("some message", result, expected, delta);
```

When you compare floating point numbers, this useful function saves you from repeatedly writing code to compute the difference between the result and the expected value.

Use `assertSame()` to test for two references that point to the same object. Use `assertEquals()` to test for two objects that are equal.

7 Resources

- The Javasoft Website:
  http://www.javasoft.com
- The JUnit Website:
  http://www.junit.org
- A directory of xUnit implementations for different technologies:
  http://www.xprogramming.com/software.htm
- A worked example of testing a GUI with JUnit:
  http://users.vnet.net/wwake/xp/xp0001/index.shtml