Bachelorarbeit
im Studiengang „Angewandte Informatik“

Component Testing with xUnit within the EU Project Daidalos II

Lars Reimann

am Lehrstuhl für Telematik

Bachelor- und Masterarbeiten des Zentrums für Informatik an der Georg-August-Universität Göttingen
22. September 2006
Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de
Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Component Testing with xUnit within the EU Project Daidalos II

Lars Reimann

Abstract
Testing is one of the most important things to accomplish when software is developed. Without testing end users would have to deal with faulty programs far more often. The question remains what is so difficult or special about testing software that even great companies with large dedicated testing departments are unable to create reliable software products. This bachelor thesis will introduce and discuss those techniques.

Daidalos II, 'Designing Advanced network Interfaces for the Delivery and Administration of Location independent, Optimized personal Services', is a huge software development project of the European Union. This bachelor thesis will address different aspects of testing of software components on the lowest level, the so-called unit level. The main question in the wider context of this bachelor thesis is, whether unit testing is applicable to such a highly integrated project.
# Table of Contents

1. Introduction .................................................................................................................... 2

2. Basic concepts ................................................................................................................ 4
   2.1 Development stages ................................................................................................. 6
   2.2 Test methodology .................................................................................................. 7
   2.3 Dynamic tests ........................................................................................................ 9
   2.4 Static tests ............................................................................................................ 12
   2.5 Test Driven Development ..................................................................................... 13

3. Unit tests in JUnit .......................................................................................................... 17
   3.1 Introduction ............................................................................................................. 17
   3.2 JUnit’s inner life .................................................................................................... 18
   3.3 Assert-methods ...................................................................................................... 20
   3.4 Test methodology .................................................................................................. 21

4. Advanced techniques .................................................................................................. 27
   4.1 Isolated testing with reduced functionality objects .............................................. 27
   4.2 Inheritance ........................................................................................................... 29
   4.3 Polymorphism ....................................................................................................... 30

5. The Daidalos II Project ............................................................................................... 31
   5.1 Introducing the Daidalos II Project ...................................................................... 31
   5.2 Details about the obtained source code ................................................................. 33

6. Applicability of techniques to Daidalos II ................................................................. 36
   6.1 Integration of techniques ...................................................................................... 36
   6.2 Automation ............................................................................................................ 45
   6.3 Feasibility .............................................................................................................. 46

7. Conclusions .................................................................................................................. 47
   7.1 Summary and discussion ....................................................................................... 47
   7.2 Perspective to JUnit 4 .......................................................................................... 49

Appendix ........................................................................................................................... 51
   A. JUnit setup instructions ......................................................................................... 51
   B. JUnit extensions ...................................................................................................... 55

Acronyms .......................................................................................................................... 58

Bibliography ..................................................................................................................... 59

Online Resources ............................................................................................................. 61
1. Introduction

Testing is one of the most important things to accomplish when software is developed. It can assure that a program is relatively defect free. Without testing end users would have to deal with faulty programs far more often.

But nevertheless, defective software – whether it is open source or commercial projects – exists in a variety of ways. The question remains what is so difficult or special about testing software that even great companies with large dedicated testing departments are unable to create a software that is free of defects. The reason may either be the nescience of testing techniques or a matter of the costs of an external test team. This document clarify facts about testing and introduces relatively new approaches like test driven development.

Computer scientist Edsger W. Dijkstra recognized the core problem when he stated in the 1970s: "The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence." Dijkstra was right about this problem at that time. Until today, software testing has changed a great deal, up to the point where there are methods to symbolically prove the correctness of a program. The following document will introduce and discuss those techniques.

Daidalos II, 'Designing Advanced network Interfaces for the Delivery and Administration of Location independent, Optimized personal Services', is a big software development project of the European Union. It aims at creating "a blueprint of a true Beyond 3G Framework" [URL: TMG_UniGoe] in order to integrate "mobile and broadcast communications and following a user-centered, scenario-based and operator-driven approach to deliver services pervasively and seamlessly across heterogeneous networks" [URL: TMG_UniGoe]. Within this project lies a strong focus on testing with Java being the primary programming language. This bachelor thesis will address different aspects of testing of Daidalos II components on the lowest level, the so-called unit level (or class level in Java). The main question in the wider context of this bachelor thesis is, whether unit testing is applicable to the highly integrated project Daidalos II.

In the following a short outline of the various aspects of software testing that are introduced and discussed in this paper will be given.

Chapter 2 gives an overview of the basic concepts of software testing. Because of software testing being such a wide field with a tremendous amount of different tests, the chapter focuses on a selection of important testing techniques which can lead to more reliable programs. The chosen sample consists of the most important traditional as well as more innovative testing methods. The tests, their applicability, as well as their advantages and disadvantages are discussed here.

Chapter 3 contains details about the JUnit framework, its basic usage and the idea behind JUnit as a reference implementation to the xUnit model. Additionally, it contains practical test methodologies which are useful for developers or a quick study of best practices.

Chapter 4 addresses some advanced techniques mainly based on the programming language Java. However, most of the principles also apply to every other object-oriented programming language. Testing software with the help of reduced functionality objects and methods like dummy and mock objects is presented as well as testing properties of other object-oriented features like inheritance and polymorphism.

Chapter 5 goes on to present the Daidalos II Project in general. It shows central aims and the vision of Daidalos. The technical dependencies will be described as well as a scenario based example. Furthermore, details about the Daidalos II source code (working package 4) which was used throughout this document will be presented.
Chapter 6 handles testing within the Daidalos II project. It practically presents and evaluates the applicable testing methodologies to a subsection of a large and complex project. It also presents an automation library which can help creating so-called mock objects, simulated objects which can help to create independent tests when other parts of a project are not yet available. It comments on the feasibility of independently creating tests while the implementation of the software still continues.

Chapter 7 summarizes the main aspects of this document and the drawn conclusions. A short perspective and the new version 4 of JUnit will be presented.

The appendix contains important practical tips for installing JUnit with and without the presence of an integrated development environment. A list of available extensions to JUnit is also given.
2. Basic concepts

Software tests can be categorized in many ways. The central aspect of differentiation would be whether real-time applications are tested or not. Tests for real-time applications also need special attention because it must be guaranteed that the application does fulfill its tasks in a well-defined time frame. But testing real-time applications is out of the scope for this document\(^1\). It covers software / component tests where real-time testing is irrelevant.

![Classification of software tests](image)

Figure 1: Classification of software tests

Figure 1 shows a partial overview of how software tests can be classified. It is a partial overview because it only covers the main categories which can be divided into many more subcategories\(^2\).

The most accurate definition of component tests\(^3\) found in the sighted literature is the following:

"Component testing is the act of subdividing an object-oriented software system into units of particular granularity, applying stimuli to the component's interface and validating the correct responses to those stimuli, in the form of either a state change or reaction in the component, or elsewhere in the system" [Silverst00].

The only shortcoming of this definition is that it is limited to an "object-oriented software system" because components or units are also found in non-object-oriented software development, such as the C-programming language. However, most of today's programming techniques are object-oriented so this definition covers the major part of

---

\(^1\) Facts worth knowing about real-time programming can be found under [URL: Realtime].

\(^2\) A more detailed classification can be found in [Liggesmeyer02], p. 34.

\(^3\) Component tests are also called unit tests and those terms are used as a synonym in this document.
software systems. But for every programming language where certain behavior is put into units, like separate modules or files, component tests for this behavior can be created and testing frameworks do exist for these languages in the majority of cases.

Another definition often found in more practically orientated literature is: "A unit may initially be a manageable, in its context complete, piece of program code. In the context of automated tests, one should then be able to test this code independently of other code – and independently of other units respectively" [Meffert06].

Essential definitions to describe what is actually being tested are:

- **Object under test (OUT):** means that the actual object together with its methods is currently under test.
- **Class under test (CUT):** means that the collection of objects which form a class are being tested.
- **System under test (SUT):** means that the whole system itself is currently under test, for example by a team of testers.

Also important is the definition of defect and error: A defect is the actual flaw within the source code. This may be a design flaw, a type error and so on. An error is the manifestation of the defect when the program is executed, for example resulting in a program crash.

Figure 2 shows the relevance of today's practical techniques. Formal techniques are rarely used. Static reviewing and coding-style analyses play a central role because many of the defects can be found using this technique. The unit test approach is widely used and is a very significant dynamic test method.

---

4 This definition was translated from the original source (German).

5 The figure uses extracted data from the given source; however, data values remain unchanged.
2. Basic concepts

2.1 Development stages

Software testing and debugging often involves executable versions of a software package which are not yet ready for commercial use. These pre-released versions are often limited in their functionality or still contain defects. In the majority of cases they are released only to a closed user group, predominantly the testing group. Sometimes these preliminary versions are released to a wider public, for example the Microsoft public beta of Windows Vista [URL: MS_Vista]. They are used to increase the feedback and the chance of finding defects in the released product. The different stages are briefly described here so that the state of source code can be judged using these terms.

Pre-alpha

Pre-alpha development stages are also known as 'development releases' or 'nightly builds'. When this releases are made available the designated features of the software are not yet "feature complete" [URL: Wiki_1] or even not implemented at all. Developers may still think about which additional features to add. Testing is not very common in this stage. Also, a pre-alpha version can still be unstable.

Alpha

An alpha version should contain most of the features planned for the final version of a product. This release marks the first major achievement and the implementation is mostly stable when it comes to its execution. But the product still needs significant parts of debugging and testing and this work is mainly done in this stage of development.

In the Open Source community the intent is often to present some useful or new features to other developers and convince them that the final release may be a useful piece of software. Additional programmers who help developing the concerning software are often found via the early stages of software development. When Closed Source software is made publicly available in an early state, the developers are often almost enthusiastic about their software. They may consider their development release worthy to be publicly available and thereby show their confidence about the next stages of the development tree or even the final version.

Beta

A beta version is considered feature complete and often the addition of new features is frozen when this version is released. A beta test often includes the installation on external systems outside the initial development group. The beta test is carried out by external testers because it is more likely that they will find additional flaws by using a fresh approach to the software. Additional background information is available at [URL: Wiki_1].

Gamma and final releases

Gamma or release candidate versions imply that the software product is code complete.

---

6 Nightly build could also be available when the software has already reached a stable or final version and the development still continues.
This means that no new source code will be added in this stage. Also, this version is considered to be very close to the final version of a software product. There may still be some defects left in a gamma release but for this reason it is – like a beta version – released to the public. The testing phase is also in its final stage and close to completion.

The final version of a software product is ready for being deployed to the end user. It is considered stable and bug-free but that is never a guarantee that the source code is actually perfect. Hidden defects could still reside within the code and only occur on rare occasions. Commercial software is sometimes released when not all defects have been removed for financial reasons. The testing and the development still continues and after a shorter time period (e.g. from some weeks to several months) a service or security update is released to compensate for additional defects.

2.2 Test methodology

2.2.1 Black-box testing

With black-box testing the software under test is treated like a black box whose inside structure is not accessible. Therefore, the source code is not inspected. Sometimes though, it is not even possible to inspect the source code because the concerning software is closed source. Instead a formal or informal specification of the software is used to write the tests. A good specification contains all necessary data like the accepted input values, expected output values, side effects or even semantics. This data about pieces of the software is also known as the contract in object-oriented programming.

Since the specification contains all necessary information about the desired behavior of a program, tests can be written before the actual software is. This procedure can be time-saving especially when a dedicated test team performs those test while the remaining parts of the software program are being implemented.

The black-box approach has some advantages especially over its counterpart, the white-box testing approach:

- Tests are not very closely matched to the source code. If the same people who implement the source code also create the black-box tests, it would be possible that they overlook certain problems because they are so deeply involved in the programming.

- Black-box tests are acting like a review for the specification. Problems during the testing will occur if the specification is imprecise in any way.

- If a technically mature test description language (TDL) is used to create the specification, black-box tests can be easily ported to other platforms.

The term black-box testing is also used when security related software is tested\(^7\). A computer system is probed or attacked to find out whether vulnerable software is running on the target system. From the attacker's point of view the target system behaves like a black box because at the beginning of the penetration test he does not know anything about the target system. Details are discovered when automated tools are probing open ports on

\(^7\) This kind of black-box testing is also called zero-knowledge penetration.
the target host and analyzing the response. When every port has been probed the attacker may have gotten more information about the system allowing him to draw conclusions from his findings and plan further steps to compromise the target system.

Although security related black-box testing generally uses the same principle of a system being not accessible for an outsider it should not be mixed up with black-box testing within software engineering. In software engineering a specific program is tested and not the security of a whole system.

2.2.2 White-box testing

White-box tests are the exact opposite of black-box tests. They rely on the knowledge about the internal structure of an implementation to create test cases. Thus, the source code must be available to the tester. White box testing is also called glass-box testing because of its inside structure being transparent to the outside world. Normally, black- and white-box procedures have both to be applied to eliminate defects in program code. If a defect is discovered via black-box testing it is only obvious that an error is located somewhere in the program. White-box tests are responsible for identifying the location of the defect within the source code. Also, "whitebox testing is mainly done in the implementation phase", normally by the developers themselves [Ebner04].

Since the tests are not derived from the specification it is not guaranteed that a program which passes all white-box tests behaves semantically correct. Also, even if all white-box test pass there is no proof that the software is free of defects. This can only be assured by a symbolic test which executes all possible branches in a program. The possibility that two errors in the same module cancel each other out is more likely to be detected by a white-box test because the developers have insight into the source code and it is more probable that some test will uncover at least one of the defects.

2.2.3 Gray-box testing

Gray-box tests are trying to combine the advantages of both black- and white-box testing methodology. This technique depends on the point of view of the tester. Normally, a modified black-box approach is used where the testers are given access to some parts of the source code. The entire source code is not available to every tester most of the time. An example would be access to database related code or code that has been identified as problematic.

The other point of view would be that a tester who normally performs white-box testing gives up his access to the source code and tries to test the code with a black-box approach. As the tester still knows how parts of the code are implemented, he can benefit from this knowledge when testing the system from a more distant point of view.

Using this approach developers try to combine the advantages of both the black-box and the white-box technique. This particular method of design may have to be learned over a longer time period and may also require additional methods and discipline applying those techniques.

Every x-box testing approach has its own advantages and disadvantages. The fact that they

---

8 Additional methods may include the use of automated tools to perform the tests.
are also used in different phases of software development makes it hard to identify one 
technique as the better one. The correct technique may depend on the personal preferences 
and skills of the developer and the overall organization of the whole development process 
of a software package.

2.3 Dynamic tests

The property of a dynamic test of a software package is that the software is executed on a 
real or virtual CPU and fed with input data. The behavior is observed, the output data is 
then sighted and compared to the expected values. This technique behaves like a spot 
sample of the concerning software because today not every input data can possibly be run 
through the program either manually or automatically. In the majority of cases the set of 
input values is too big and testing all possibilities would require too much time. Tests must 
also be cost-effective; it is undesirable to keep a test running for a very long time.

Dynamic tests can only result in incomplete conclusions about the correctness of a 
software because it is not possible to test the complete input set. Only formal verification 
can prove the consistency of specification and implementation⁹.

But in fact, dynamic tests have a high practical relevance¹⁰. Not only because they are 
easier to understand than formal verification techniques but because they are also 
affordable. Today, there are so many subclassifications of dynamic testing methods 
available that it is most likely that a suitable method can be found. Unfortunately, not 
every technique can be presented as it would exceed the scope of this document.

2.3.1 Functional test

A functional test is a sub-category of dynamic testing and it makes intense use of the 
software specification to verify the correct behavior of a computer program. The 
specification is normally created at the beginning of a software development process. Such 
a design process is called Big Design Up Front (BDUF) because creating the software 
specification takes a rather large amount of time compared to the rest of the development 
process. Functional testing also plays an important role in every testing phase. For example 
when testing software components on the unit level, the units specification is examined. 
The exact behavior of a unit is defined there and functional tests can be written using this 
definition. If these implementations do not behave as formalized in the specification, the 
underlying source code is likely to contain a defect.

Since functional tests use the software specification, it is mostly the desired features of a 
software product which are tested. Thereby, functional tests mainly confirm that all desired 
features have been completely implemented. If functional tests are successful, they are also 
able to heighten the trust in this software entity. But functional tests cannot verify that the 
source code has been tested completely. It is not guaranteed that the input data led to a 
complete test of the program's properties. Every functional testing approach is also a black-
box technique because the source code is ignored.

⁹ See chapter 2.4 for an overview of verification techniques.
¹⁰ See figure 2 for the practical usage of today's techniques.
2.3.2 Structural test

A structural test also belongs to the category of dynamic testing. Compared to the functional test, it uses the specification only to analyze the output of the structural tests\textsuperscript{11}. Actually, these kind of tests are judged on a basis called code coverage. Tests are complete if they cover every part of the source code. But there are different levels of code coverage and although complete tests are desirable for every programmer, they require much time and are only used in important software\textsuperscript{12}.

Structural or control flow oriented tests are using the internal processing logic of a program to create the tests. This includes conditional statements, program loops or branches. This kind of test is primarily used in the unit test. A problem occurs when a specific branch of a program is overlooked by the tester because there is often no method of exposing the absence of tests of a specific part of the source code.

The most important techniques of structural testing will now be briefly presented\textsuperscript{13}. The statement coverage test requires that every statement in the source code is at least executed once. This is a very basic form of testing because it is error-prone to some operations, for example the alternative branch of an if-statement. Since a statement coverage test is a very simple technique, it is never used as a standalone test. But automated statement coverage tests are able to uncover statements which are never executed and they can be removed. Another widely used technique is code instrumentation. It uses counters to check how often a statement has been executed or if it was executed at all. By comparing the results to the statements of the source code problematic statements can be identified and corrected.

A branch coverage test is a more strict type of test and completely includes properties of the statement coverage test. It is also called decision coverage test because boolean conditions are to be evaluated to both TRUE and FALSE statements. Generally, branch of internal control flow is being tested. This has the advantage that branches are detected which are never traversed. With the appropriate input data it is also possible to detect branches which are passed very often. These branches can then be optimized to speed up the overall program performance. Visualizing the control flow in a program often helps the tester to create more efficient tests.

A condition coverage test attends to the fact that conditions often depend on other more or less complex pieces of conditions. If conditions depend on additional conditions, these may not be trivial to solve, to understand or to test. Therefore, this test evaluates atomic conditions in several levels of accuracy. An atomic condition is not linked with other conditions and compares two values using only one comparison operator. For example the simple condition coverage test requires the evaluation of all atomic decisions against TRUE or FALSE. The strictness increases with the minimal multiple condition coverage which additionally requires the evaluation of the overall condition (composed of all atomic conditions) and also the evaluation of all conglomerated conditions against TRUE or FALSE. The most comprehensive test of this type is the multiple condition coverage test which requires the evaluation of all combinations of all atomic conditions. "This technique produces an exponentially growing number of test cases as for a condition composed of $n$...

\textsuperscript{11} For example if the output is within the correct type of value (int, float, string).
\textsuperscript{12} For example if a software failure would cost lots of money or even the loss of life.
\textsuperscript{13} A more detailed description of structural test can be found in [Liggesm02].
2. Basic concepts

atomic conditions it produces $2^n$ test cases\(^{14}\) [Liggesm02]. This technique is only used for software which runs in a critical environment.

The path coverage test is only of minor practical importance because it requires that every program path is tested. A path consists of distinct branches. Thus the set of all paths also increases exponentially. For example if a program contains ten conditions, 1024 paths have to be tested. A problem is also that loops introduce a vast number of paths. Despite the computing power of modern computers or even grid computing a small number of nested loops could exceed today's computational power. Therefore it is hardly used.

2.3.3 Diversified test

A diversified test does not use the software specification nor the percentage of code coverage to judge the functionality of a program. It compares different versions to each other, often with the help of automation tools.

The back to back test for example works in this manner: Independent groups of developers implement a software using the same specification. The different versions are then fed with the same input test data. If automated tools recognize no differences in the output then each version is correctly implemented. If there occur inconsistencies in the output, the source code has to be examined to clarify where the error was reported. As long as the automated tool reports no errors, no manual intervention is required.

The mutation test uses a completely different approach. It is mainly used to test the thoroughness of the tests by introducing deliberate errors into the source code, thereby creating new versions. The original version is considered to be correct. It is then checked if the tests are able to identify the introduced errors. If not the tests have to be revised and improved.

2.3.4 Boundary level analysis

Boundary level testing assumes that errors often arise around fixed boundaries in the domain of the input values and it is an essential heuristic for testing. Naturally, the specification is considered to evaluate predefined boundaries in the input data. In the majority of programming languages also special values exist which should be tested, for example the null-reference of an object or minimum and maximum values of the primitive data types. Additionally, passing an empty string variable should be tested as well as a string variable which holds a very large string. An overview about boundaries which should be tested is available at [URL: TestingCat].

2.3.5 Random Test

A useful combination with the presented methodologies is the use of a random test. Those kind of tests use randomly created data which is then used as an input for the program under test. This stochastic approach guarantees that test case developers do not always use the same or very simple input values. For example a randomized character set or a random number generator may be used to test a computer program. This approach may reveal defects which would be difficult to find with ordinary techniques. But to use it cost-
effectively the random data must be automatically created.

An interesting variant is to record input data created by normal users of the program. This input data is then randomized using only the recorded values. Thereby a day-to-day use of operation can be simulated and it can be evaluated how stable the software under test behaves under normal circumstances.

2.3.6 Error guessing

Error guessing is also known as special value testing. Skillful developers / testers often have the ability to know from experience at which values a software is most likely to contain defects or arithmetic errors, for example errors around arithmetic boundaries or typical mistakes made by inexperienced developers\(^\text{15}\). Thus, the probability that errors are found with the help of error guessing by an experienced tester can be much higher than the error-detect-rate of the previously described testing approaches. But it is not a process which can be learned theoretically because it is based on longtime experience.

Error guessing shows clear signs of an ad hoc approach to testing. It looks like defects are searched without any rules or systematic plan. But if closer examined, it is an experience-based technique because the person who practices error guessing needs the proper experience to create the correct test data which identifies the defect within the program. Also the tester may have a personal, structured plan of finding errors committed only to his memory.

A critical point would be if error guessing was used as the only method to discover defects. That would be inappropriate because testing would rely to much on one person to discover defects. Also there is no way to say how many percent of the source code is covered using error guessing. But if this approach is combined with systematic testing it can be a reasonable addendum.

2.4 Static tests

Static tests all have in common that the corresponding program is not executed. Therefore it can not be observed how the program behaves in its designated environment. For example, if an error is located in the program's start-up routine the program will not start at all or crash instantly. Although the rest of the program code might behave as expected, such an error would not be found if other static testing methods fail. Nevertheless, there are many techniques to prevent such errors. Each of the following principles can be assigned to the category verifying test or analyzing test.

A verifying test tries to prove the correctness of an implementation by using formal or symbolic methods. A prerequisite for using this technique is that the specification of the software was composed using a formal description language. Only then an interpreter can execute certain modules in an artificial environment with these symbolic values and eventually prove the correctness of a program for a complete set of input parameters. This method also requires a solid knowledge of mathematical theorems. This may be a reason why this approach is not widely used (see figure 2).

\(^{15}\) For example null-references or floating point operations.
Several methods exist where source code can be statically analyzed. An automated test to the compliance with coding guidelines would be such a test. Also, a widespread method is the use of static reviews of the source code (see figure 2). It has the advantage that the analysis can begin even if the implementation is incomplete and not yet executable. Skilled developers with distributed tasks review the code. They leave comments about obvious or possible errors in the source code next to the location of the defect. It is also possible to discuss the outcome of the analysis in a meeting where all involved people participate. A detailed description of reviewing techniques is available in [Graham93].

2.5 Test Driven Development

At first, it should be made clear that test driven development (TDD) is not a testing methodology but a software development methodology. The following definition was taken from testdriven.com, the leading TDD community site on the Internet:

"TDD is a lightweight programming methodology that emphasizes fast, incremental development and especially writing tests before writing code" [URL: Testdriven].

The term "lightweight programming" is actually an out-dated expression. Today, the term agile software development is more common. It describes a development process which respects that software development has become very dynamic in the recent years. Integral parts for examples are a motivated team that works very flexibly together with the customer, no big design up fronts which produces a complicated software specification, increased importance of writing clean source code, reduced importance of a detailed software documentation and the development in short pieces, the so called iterations. The full Agile Manifesto which was created in 2001 is available under [URL: Agile_Manifesto].

A short excursion on XP

However, TDD is also an integral part of Extreme Programming (XP). It was developed by Kent Beck, Ward Cunningham and Ron Jeffries during their work for the commercial project "Comprehensive Compensation System" (C3 project) at the automobile manufacturer Chrysler from the year 1995 to 2000 [Case98]. Before the project was set up with Extreme programming techniques it was considered a failure by Chrysler as other approaches by other parties had failed. All source code was discarded. Since the successful development by the founders of Extreme Programming the system has been used to pay approximately 87.000 employees and it got operational by mid 1999 [XP1_03].

In 2001 Ron Jeffries defined Extreme Programming [URL: XP]:

"Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation."

Additionally, XP introduces techniques like pair programming: Two developers working together on one computer. One person is actually programming and the second developer provides a constant review of the source code. Changing their roles is scheduled from time to time to counteract boredom.
Another keyword in XP is continuous integration: Developers try to build an executable version of their software several times per day. Using this technique the feedback is increased which should improve overall software stability. Additionally the method of evolutionary design plays an important role during the development process. A strong focus on design improvements is considered more important than a carefully designed software specification [URL: XP_Dead]. Thereby, flexibility is maintained throughout the implementation. More details about XP can be found in [Jeffries00].

According to Kent Beck TDD consists of three main rules: [Beck02]:

- Red – Write a little test that doesn't work, and perhaps doesn't even compile at first.
- Green – Make the test work quickly, committing whatever sins necessary in the process.
- Refactor – Eliminate all of the duplication created in merely getting the test to work.

Naming the paradigm red / green / refactor arose because of the visual feedback testing frameworks signal to the user. At the beginning when only a test is written and no source code exists a red bar which signals the outcome of the test shows an error because there is method or function which could be tested. After adding the appropriate functionality the test succeeds and the user interface of the test framework shows a green bar. After a successful test, the code is then refactored removing code duplicates and improving overall code readability and organization.
2. Basic concepts

**TDD and traditional methods**

In contrast to the TDD approach figure 3 shows the traditional software development process. The **waterfall model** clearly distinguishes between different phases of the development process. Although more modern methodologies exist, like TDD or the spiral model [URL: Spiral], this technique may still be in use in a variety of projects [URL: Waterfall04].

The waterfall model dates back to 1970 when it was originally introduced by Royce [URL: Wiki_2]. Even in the original publication Royce himself described the model as "risky and [inviting failure]" [URL: Royce70]. It is not clear why the original waterfall model gained so much popularity; maybe because of its simplicity. In fact, this model describes how an inexperienced person, who does not know alternative methods, would plan the development of a software project. For example the implementation phase can have influences on the design specification when the BDUF contains impractical requirements. When this is recognized much time has already been spent on the first phase. Furthermore, the testing phase has a great influence on the previous coding phase when defects are removed. Sometimes even the whole design has to be changed when testing reveals an unstable implementation.

Figure 4 shows in which of the different phases of software development errors are produced and discovered. Additionally, it contains the errors-correction costs to remove a single defect in the concerning phase. It is stated that a waterfall model which uses a
BDUF is more effective because the planning is so detailed that errors are already found in the first phase [URL: Wiki_2]. But under the assumption that no software is perfectly flawless in its final version, figure 4 strongly favors the TDD approach. In TDD the analysis/planning phase is very short or even nonexistent. The development and coding phase is strongly driven by the unit tests. So if the tests control and drive the development of these phases, it is unlikely that many defects can find their way into the source code. Thereby, the error-correction costs would be reduced to a minimum because the defects found in the system-test phase and in the field would be greatly reduced.

![Figure 4: Number of errors and error-correction costs](Liggesm02)
3. Unit tests in JUnit

3.1 Introduction

JUnit is an exemplary implementation of the xUnit model. Therefore, and because of Java properties like packages, it is used as a reference implementation in this document. The differences between this and other implementations such as CppUnit can be found in [Hamil04].

For example: A rudimentary class:

```java
public MyClass {
    private String version="1.00a;

    public getVersion(){
        return version;
    }
}
```

A test skeleton for the rudimentary class:

```java
import junit.framework.TestCase;

public class MyTestCase extends TestCase {
    public void testRudimentary() {
        MyClass myObj = new MyClass();
        assertEquals("1.00a", myObj.getVersion());
    }
}
```

The test skeleton shows the basic principle of a unit test. So called assertions are used to verify the state of objects and variables.

In the majority of cases, unit tests are concurrently developed with production code but are not built into the final software product. Today's techniques allow that the test code is kept in the same logical package but is located in a different local directory on the storage device. This has the advantage that the production code is not cluttered up with test code and the test code can still access methods whose visibility is protected or package scope.

JUnit supports the combining of test cases to test suites. In the majority of cases they group tests which have a common theme. AllTests-classes which combine every single test of a package should be created to later combine these tests with AllTest-classes from other packages into a project-wide superclass. It is advisable to think of such things before writing the first unit tests. It would save time if project-wide testing guidelines were created.
3. Unit tests in JUnit

There are also certain needs for an automated unit test framework [Link05]:

- "the programming language of the unit test framework must be the same as the programming languages"
- there should be a possibility to separate test code and application code
- the execution of the test cases must be completely independent of each other
- test cases can be fully combined to test suites
- the success (or failure) of a test case has to be clearly visible"

JUnit fulfills these prerequisites perfectly as we will see later in this document.

When new to unit testing the question will arise whether every single method or every get / set method has to be tested. The answer depends on how the code is structured and how complex it is. But it is considered a good practice if the behavior of a program is tested. For example if all the behavior is implemented in a single method it surely has to be tested. But more often a program depends on several methods, including get / set methods. If they do nothing more than simply assign a primitive value, they most likely do not need to be tested because nothing can go wrong. But if there is something computed within the get / set methods they will be worth a test because there can always be computational errors. Another example would be a stack with the default methods push, pop and peek. All these methods act together to form the full functionality of the stack. It would make no sense to test one of the methods in isolation or to focus too strong on how these methods are really implemented because it could differ as each programmer has his own style of programming. Therefore, it is highly recommended to test every object as a whole.

It is also most common to test classes entirely through their public interfaces. Not only because most of the time there are separate files created which contain the test methods and thereby the private and hidden properties of a class cannot be accessed. But also because a test cannot only rely on program internals because often tests are written by a dedicated testing department. Surely they have insight into the source code but it is time saving when the team does not have to develop additional methods 'just for testing'. Therefore, it is also a good practice if the complete functionality is testable through the public interface.

3.2 JUnit's inner life

The source code of JUnit 3.8.2 is approximately 150 KB in size and contains 56 files. To say it with Martin Fowler's words:

"Never in the field of software development was so much owed by so many to so few lines of code".

The source code for the JUnit framework includes three different user interfaces, a test runner and extensions. Figure 5 shows an UML class diagram of the most important classes within the package junit.framework. The class Assert contains static assert-methods which compare actual values with expected ones. If the comparison fails a failure occurs. Table 1 contains some more important assert-methods. TestCase contains the fixtures for running and managing multiple test cases. Every custom tests has to extend the TestCase class.
The class TestResult takes care that the results of the tests are collected. The status of the results is afterwards displayed with the help of the user interfaces. TestSuite allows to add test cases into test suites for grouping purposes. A TestListener interface can be implemented to get information about errors and failures. For example, it is used in the different TestRunner classes.

JUnit is separated into several different packages as shown in Figure 6. As already presented, the framework package contains core functionalities. There is also a package containing samples. This is useful when one wants to take a quick look at how things work. It is well commented by the authors. The package test contains unit tests for the JUnit framework itself. So the developers stayed true to themselves and programmed tests for their own product. There are also packages for extensions and runners, responsible for running tests. The last packages contain the three different user interfaces which signal the success or failure of a unit test to the environment. There is a textual interface, an Abstract Windowing Toolkit (AWT) interface and a more modern Swing implementation.

Figure 5: JUnit-class diagram of the package junit.framework

---

16 Most of today's IDEs include their own implementation of a graphical user interface which is sometimes more practical to use than JUnit's build-in user interfaces.
3.3 Assert-methods

The assert-methods play a central role among the JUnit functions. They test whether specific conditions correspond with our expectations of the state of an object. `TestCase` extends `Assert` so that assertions can be written without having to refer to an outside class. The methods need one or more parameters which are evaluated at run-time. If an assertion fails the test case fails and an `AssertionFailedError` is thrown. JUnit proceeds with the next test (if there does exist one). Table 1 contains the most important assert-methods but there are some more\(^\text{17}\).

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertTrue(boolean condition)</code></td>
<td>satisfy condition</td>
</tr>
<tr>
<td><code>assertFalse(boolean condition)</code></td>
<td>satisfy condition == false</td>
</tr>
<tr>
<td><code>assertNull(Object object)</code></td>
<td>ensure that object is null</td>
</tr>
<tr>
<td><code>assertNotNull(Object object)</code></td>
<td>ensure that object is not null</td>
</tr>
<tr>
<td><code>assertSame(Object expected, Object actual)</code></td>
<td>guarantee object-identity (expected == actual)</td>
</tr>
<tr>
<td><code>assertNotSame(Object expected, Object actual)</code></td>
<td>guarantee object-distinction (expected != actual)</td>
</tr>
<tr>
<td><code>assertEquals(Object expected, Object actual)</code></td>
<td>ensure object-equality(expected.equals(actual))</td>
</tr>
<tr>
<td><code>assertEquals(&lt;primitive&gt; expected, &lt;primitive&gt; actual)</code></td>
<td>ensure equality of the primitive types int, short, long, byte and char</td>
</tr>
</tbody>
</table>

\(^{17}\) If one wants to take a quick look at all the assert methods the online javadoc of JUnit could be used [URL: JavaDocAssert].
3. Unit tests in JUnit

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertEquals(double expected, double actual, double delta)</td>
<td>ensure that the difference between expected and actual is $\leq$ delta</td>
</tr>
<tr>
<td>assertEquals(float expected, float actual, float delta)</td>
<td>ensure that the difference between expected and actual is $\leq$ delta</td>
</tr>
<tr>
<td>fail(String message)</td>
<td>create a failure</td>
</tr>
</tbody>
</table>

*Table 1: Overview of the primary assert-methods*

3.4 Test methodology

3.4.1 Failures and errors

JUnit 3.8.2 still distinguishes between failures and errors\(^{18}\). A failure indicates that an assertion has failed and the production code has behaved in a way that was not expected. Normally a failure suggests that there is a problem with the production code that needs to be fixed. An error is detected by JUnit if a test itself or the production code throws an exception. It is recommended to fix the errors first because other failures could depend on the error and by fixing the error more tests pass.

If unchecked exceptions (like `RuntimeException`) make their way into test code, they are caught by JUnit and recorded. There is no need to handle checked exceptions in your test code. They can be handed to the JUnit framework by adding a `throws exception` clause to the corresponding method.

3.4.2 Naming conventions

Normally there are already coding or naming conventions in larger projects. This helps to communicate the different meanings of the code to understand the properties of an object and what the overall system does. This is not different when writing test code. Naming conventions for tests are also very important if other people want to read and understand your tests. According to Rainsberger ([Rainsb04], p.17), "a programmer ought to be able to sit down at a computer, browse your code, and form an accurate mental model of your system by just reading the names. If not, then the names are wrong". This statement is a little rough but it reflects the importance of right naming within your projects. The de facto standard is the following: the name of the test methods should shortly and precisely describe the behavior of a test. If you have tests which differ only slightly in their behavior but still test different things you could separate the different names with capital characters. That makes it easier to read the code.

The test classes should be named exactly like the classes which contain the production code but with the suffix `Test`. This enables you to easily identify test classes within the file system or a package viewer because in most cases files are sorted alphabetically and you will find your test class right next to your original class\(^{19}\). When recognizing that there are very many lines of test code for a single class the functional range of your class under test

---

\(^{18}\) JUnit 4.x does not differentiate between failures and errors anymore.

\(^{19}\) When test code and production code is put in the same directory. Most IDEs also support both types of code in the same logical package but in different directories.
3. Unit tests in JUnit

is too big. You should then consider extracting some parts of your CUT into a separate
class or use fixtures in your tests to remove code duplication.

The default for naming methods within the test classes is to let them begin with the prefix
*test*. The first reason for doing so is that the methods cannot accidentally be mistaken for
production code. The second and more important reason is that JUnit automatically
considers all methods in a class which start with the prefix *test* as test methods and puts
them into the same test suite. Test suites are required in JUnit to group several tests. Surely
the test suites can be created manually. When no test suite is specified JUnit automatically
creates a default suite by using the Java Reflection API\(^{20}\) (*java.lang.reflect*) to scan
your class for test-methods. After this procedure this group of tests can be executed and
run by the *TestRunner*.

### 3.4.3 Testing void methods

Methods that have no return value are not directly testable because one cannot compare an
expected value against an actual, returned value. But they must have at least a side effect
on one or more objects. Otherwise this method would just do nothing. A simple example is
a list. A list may have methods like *add()*.*delete()*,*isempty()* or *contains()*,*where
*add()* and *delete()* are void methods. If something is added to the empty list it must
have an influence on the *isempty()* method because the list is not empty any more. When
*contains()* is called with the just added element, it also has to behave differently and
return true or the position of the element in the list. These facts show that *add()* behaves
correctly.

### 3.4.4 Testing constructors

According to Rainsberger, "the simplest kind of constructor test uses exposed internal
states to verify that parameters passed to the constructor reached the appropriate
properties" ([Rainsb04], p. 37). He gives the following example:

```java
public void testInitializationParameter() {
    assertEquals(762, new Integer(762).intValue());
}
```

Here the *intValue()* acts as a get method that verifies that the object has been initialized
with the correct value. Furthermore he proposes the use of an *isValid()* method which is
not a test method, but resides in the CUT itself. It can verify that the initialized values of
the object are within a valid scope, for example a date value which is not allowed to be in
the past. Here is his example with a money transfer:

```java
public void testInitializationParameters() {
    BankDepositCommand command = new BankDepositCommand("123", Money.dollars(125, 50), today());
    assertTrue(command.isValid());
}
```

In reality the developers decide for themselves whether they test constructors or not. In the
majority of cases they skip the tests because they are only elemental and very easy. If one
is new to JUnit, constructors should be tested and after some experience they may be

\(^{20}\) A tutorial about Java's Reflection API is available at [URL: Reflection].
skipped. But if an error finds its way into a constructor which is not being tested it may be
difficult to identify. Therefore it depends on the personal skills of each developer to write
tests for constructors. A common paradigm is: "Test until fear turns to boredom" or "Test
only if the code is not too simple to break". The same rules of course apply to the next sub
chapter where testing get / set methods is being discussed.

Finally, Rainsberger gives some examples of mistakes often made by beginners ([Rainsb04], p. 40):

```java
public void testConstructorDoesNotAnswerNull() {
    assertNotNull(new Integer(762));
}

public void testConstructorAnswersRightType() {
    assertTrue(new Integer(762) instanceof Integer);
}
```

The first test against `null` is unnecessary because a constructor either creates an object or
throws an exception. In the second example a constructor can only return a type of its class
and no other type. These facts are assured by the Java language specification.

### 3.4.5 Testing get / set methods

Though testing is very useful in a variety of cases, not everything has to be tested. If an
object stores its values according to the principle of data hiding, there can be numerous
get / set methods. If these methods only exist because values are read or assigned they are
too simple to be worth a test.

```java
public class Money {
    private long cents;

    public long getCents() {
        return cents;
    }
}
```

Testing this example would truly make no sense because even a novice programmer would
make no mistake here (except typographical errors, but most modern IDEs contain a spell
check). But programmers are encouraged to write many tests and if the slightest work /
computations are done in one of the get / set methods they should write a test for it and
compare the result with the expected result.

### 3.4.6 Testing interfaces

Interfaces occur very often in code that is well planned and structured and they are widely
used when APIs are designed. Testing interfaces seems difficult at first because there is no
way to instantiate an interface. Furthermore, no specific implementation of the interface
should be tested - all future implementations should be tested that derive from a specific
interface. Eric George gives a good guideline for testing interfaces in [George06]. He
proposes three main rules:
3. Unit tests in JUnit

- "Write an abstract test for every interface and abstract class. An abstract test should have test cases that cannot be overridden. It should also have an abstract Factory Method for creating instances of the class to be tested.

- Write a concrete test for every implementation of the interface (or abstract class). The concrete test should extend the Abstract Test and override the factory method.

- Tests defining the functionality of the interface belong in the Abstract Test. Tests specific to an implementation belong in a Concrete Test."

He also gives an easy to understand example which could be used as a blueprint for writing tests for interfaces. He introduces an interface which implements basic statistical calculations:

```java
public interface StatPak {
    public void reset();
    public void addValue(double x);
    public double getN();
    public double getMean();
    public double getStdDev();
}
```

Now take a look at his proposal of the abstract test for this interface:

```java
import junit.framework.*;

public abstract TestStatPak extends TestCase {
    private StatPak statPak;

    public TestStatPak(String name) {
        super(name);
    }

    // This is final to prevent descendents from tampering with it
    // and screwing up these tests

    public final setUp() throws Exception {
        statPak = createStatPak();
        assertNotNull("Problem creating StatPak instance.", statPak);
    }

    // Factory Method. // Every test of a concrete implementation must override this
    // to return an instance of the actual implementation.

    public abstract StatPak createStatPak() throws Exception;

    public final void testMean() {
        statPak.addValue(2.0);
        statPak.addValue(3.0);
        statPak.addValue(4.0);
        statPak.addValue(2.0);
        statPak.addValue(4.0);
    }
```

24
assertEquals("Mean value of test data should be 3.0", 3.0, statPak.getMean());

public final void testStdDev() {
    
}

His comments on the code: "Notice that the tests as well as the setUp() method are final. The motivation for doing this is to prevent descendant test cases from overriding these methods. This ensures that the tests run are the same for every implementation of StatPak. If a class implements StatPak and fails some of these tests, then there is either a flaw in the implementation, or the tests were incorrect or incomplete. Remember, we are identifying the intended behavior of all StatPak implementations. These tests are going beyond what can be checked with the compiler to test functional compliance."

He then gives the test code for two concrete implementations of StatPak; SuperFastStatPak and SuperSlowStatPak. Only the code for one is given here because they differ only in the class / constructor name:

public class TestSuperFastStatPak extends TestStatPak {
    
    public TestSuperFastStatPak(String name) {
        super(name);
    }

    public StatPak createStatPak() throws Exception {
        return new SuperFastStatPak();
    }
}

His comments on this source code: "Notice that in each Concrete Test, we implement the createStatPak factory method by creating a StatPak of the particular type we are testing. We also added main methods so that we can run these tests. [...]"

With only a few lines of code we have reused the test code written for the interface. Plus, the more implementations we have, the more time we have saved. In fact, it now may be possible to write several implementations of StatPak without writing a single new test!

Depending on the interface we are testing, we may want to include additional tests in the concrete tests. These tests should be specific to the implementation. For example, a test to determine whether SuperFastStatPak is really as fast as it claims should included in the concrete test. Since SuperFastStatPak is multi threaded, we may also need to include some concurrency checking tests."

This short introduction for writing tests for interfaces may be useful for a first start\textsuperscript{21}. For the practical use of fixtures like setUp() or for details about test suites see chapter 6.1 (integration of techniques).

\textsuperscript{21} For further reading [Rainsb04] p.48-53 is recommended.
3.4.7 Testing error handling

The testing of error handling of production code is also important in addition to 'normal' tests. JUnit test cases shall pass if an expected error is thrown and produce a successful unit test.

Example (taken from [Hamil04], p. 37):

```java
LibraryTest.java
public void testRemoveNonexistentBook() {
    try {
        library.removeBook( "Nonexistent" );
        fail( "Expected exception not thrown" );
    } catch (Exception e) {} 
}
```

In this example `fail()` is called if the expected exception is not thrown to signal that there is an error within the source code. `fail()` is equivalent to the assert `assertTrue(false)` but it may be better readable and thus understandable. When the exception is thrown the test passes.

Let us compare this example to another given by Rainsberger ([Rainsb04] p. 57):

```java
public void testConstructorDiesWithNull() throws Exception {
    try {
        Fraction oneOverZero = new Fraction(1, 0);
        fail("Created fraction 1/0! That's undefined!");
    } catch (IllegalArgumentException expected) {
        assertEquals("denominator", expected.getMessage());
    }
}
```

Compared to the first example this code looks clearer, mainly because it makes better use of variables like "expected" to communicate the programmer's stream of thought. Also there is an (optional) test for the expected error message. After this example Rainsberger gives these recipe-like recommendations for testing exceptions:

1. "Identify the code that might throw the exception and place it in a try block.

2. After invoking the method that might throw an exception, place a fail() statement to indicate "If we got here, then the exception we expected was not thrown."

3. Add a catch block for the expected exception.

4. Inside the catch block for the expected exception, verify that the exception object’s properties are the ones you expect, if desired.

5. Declare that the test method throws Exception. This makes the code more resistant to change. Someone may change the method under test so that it declares it might throw additional checked exceptions. That change likely does not affect your test, so it ought not to cause your test to stop compiling."

If unexpected exceptions from the method under test make their way up to the JUnit framework you do not have to worry about them because JUnit reports them as an error instead of a failure. In most cases unexpected exceptions occur because there is something wrong either with the implementation itself or the surrounding environment.
4. Advanced techniques

4.1 Isolated testing with reduced functionality objects

An important rule when writing unit tests is that a single test case should be as isolated as possible when it comes to dependencies of other objects or external resources. But this approach is sometimes very difficult or even impossible because tests cannot ignore its own properties. The tests have to be complete and the interaction with other objects has to be thoroughly tested as well as the coupling with external resources. However, that is not always possible because in OOP – and especially in Java – classes rely on superior or several other objects. But the coupling with objects from the Java core API is not the major problem. It can be assumed that the Java core API has been thoroughly tested.

A problem occurs when multiple modules are implemented and these modules differ in their amount of work. It is likely that some modules are almost finished when other modules are still being designed. When it comes to unit testing, objects essential for the module under test must be imported at the beginning of the test. Otherwise the tests cannot run at all or with very limited functionality. To deal with this problem a technique which introduces new objects with simulated functionality is used. In the literature there are some naming inconsistencies concerning the different types of reduced functionality objects. Therefore, the most popular terms are being defined as the following:

- A (method) stub is a piece of source code which acts as a placeholder for functionality which has been planned but not implemented yet. Later in the line of development a stub is replaced by the real implementation\(^{23}\). A stub can also contain partial programming logic. Massol defines a stub as follows: "A stub is a portion of code that is inserted at running time in place of the real code, in order to isolate calling code from the real implementation. The intent is to replace a complex behavior with a simpler one that allows independent testing of some portion of the real code." [Massol04]

- A dummy (object), referred to as object A, can be used to satisfy parameter lists of another object B. Object B needs object A in its parameter list, otherwise it cannot be instantiated for testing purposes. Object A contains no implementation and is useless after the test completed successfully. If object B wants to call methods of object, this try should fail and the corresponding test should signal an error\(^{24}\).

- A fake (object) always returns a fixed value. A fake method for example can be used to test the behavior when a non-existing book is removed from a library program. This method would always return no for testing purposes so that the exception handling can be examined. A fake object is an object which has only fake methods in its implementation.

\(^{22}\) For example the generic object java.lang.Object.

\(^{23}\) The term 'stub' occurs frequently in computer related literature and has similar meanings; it has a special meaning in software development and should not be mixed up with other meanings like stubs in computer-memory or networks.

\(^{24}\) More information of dummy object is available under [URL: xUnitPatterns]
• A spy (object) is a fake object which has been given the additional option to record in what way it was called by an external source. Data is recorded and it can be checked if the method calls occurred as expected.

• A mock (object) is a sophisticated simulation of a real world object but it does not contain any implementational details. It can contain methods which react in such a way the developer wants them to. Massol defines a mock as follows: "... is an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests. [Massol04]"

However, the mock object approach and its usefulness has to be explained a little bit more detailed than the definition of Massol states. The following citation is found throughout the mock object community:

"Once," said the Mock Turtle at last, with a deep sigh, “I was a real Turtle.”

(Alice In Wonderland, Lewis Carroll)

At first it seems unusual to compare a citation from a children's novel to an advanced testing methodology. But it can be argued, that a mock object is a version of a real object which has been stripped down of its real purpose. But the given citation is a bit imprecise in the context because mock objects are normally used to simulate a version of a real world object which is not (yet) available at a given time. A mock object is set up with methods, parameters and optionally the desired order of method execution by the tester. In between usages of mock objects they can use a verify() method to confirm that their setup was fulfilled. An advantage of this approach is that the principle of data hiding and encapsulation of the mock object is preserved. The mock object does not need to expose its internal implementation. Another benefit is that failures in the chain of events is instantly recognized. Thereby defects in the source code can be tracked more efficiently. Without the mock object technique the exact position of a failure is sometimes not easy to track. This property of mock objects is called fail-fast. This is a trait a spy object for example cannot offer.

When mock objects are used unit tests are able to focus on testing the behavior of the class under test (CUT) because no outside components are used which may be in an unfinished state of implementation. Additionally mock objects can help in the design of entities outside the class under test. For example when it becomes apparent that certain properties of an external resource are needed to test the CUT, this can lead to a discovery of an interface declaration which can then be made available to developers of the external source to assist their work [Mackinnon00]. Examples and the general practical usage of mock object can be found in chapter 6.1.
4. Advanced techniques

4.2 Inheritance

Inheritance is the widespread way in OOP to reuse existing classes to save developing time and also to group similar objects by introducing an class hierarchy. Existing methods can be overridden and thereby altered or expanded in their properties. When testing the relationship between superclasses and subclasses a few things must be kept in mind.

An advantage of inheritance in OOP is that existing positive properties can be taken over to the subclass but this also includes the acquirement of negative properties. A selection of only the positive properties is not possible. If one does want only specific qualities or features to be available in the subclass the concerning method has to be overridden.

According to [Weyuker88] there are three important test axioms when testing modular software systems:

- The anti-extensionality axiom says that a test suite which tests an implementation according to its specification does not necessarily test another implementation created from the same specification adequately.
- The anti-decomposition axiom says that the test coverage for a module under test does not necessarily apply to the module which calls this module.
- The anti-composition axiom says that test suites which are adequate for single segments of a module are not necessarily adequate for the module as a whole.

According to [Link05] these axioms have an impact on how tests of a class hierarchy should be composed. Links states that one has to test unchanged methods of a subclass if they are directly or indirectly calling overridden methods. Further he states that tests suites of a superclass do not satisfy the testing overridden methods of a subclass because the implementational details differ. It cannot be assumed that one test suite will suffice for both classes although it is a good practice that overridden subclasses should always leave the original features intact and only expand them.

According to Ebner there are some more problems which can hinder the successful implementation of tests [Ebner04]:

- "deep (hierarchy of subclasses) and wide (usage in many classes) inheritance,
- inheritance weakens encapsulation because subclasses can get direct access to superclass elements (hence, contract of superclass could be violated by subclass),
- abuse by using as macro expansion mechanism and
- as a model of hierarchy where no sharing or using is done."

The direct testing of interfaces in Java can only be used to test the software specification because interfaces are abstract and cannot be instantiated25. But is not a critical problem because often there is no possibility to create a generic test case class for all possible implementations of an interface. More advisable are tests for each implementation. This assures a greater test coverage and less defects in the source code.

25 Further details about testing interfaces can be found in chapter 3.4.6.
4. Advanced techniques

The testing of abstract classes does not pose a significant problem when the test hierarchies of the corresponding abstract classes are also abstract. Additionally it must be payed attention that abstract test cases are not automatically assigned to a test suite [Link05]. If an abstract class is frequently accessed by tests it is suggested that a concrete implementation of an abstract base class is conducted [Meffert06].

4.3 Polymorphism

Polymorphism "is the ability of objects belonging to different types [or classes] to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior" [URL: Wiki_3]. Polymorphism at running time is called dynamic polymorphism. The signature of a message sent to a polymorphic object determines which method is actually called [Link05]. Normally, polymorphism in OOP improves the possibilities a developer has to create computer software. For example the accumulation of methods within a class to react to an input in multiple ways.

When it comes to testing some problems might occur. When there are several choices within the program it is more complicated to understand what is actually done, thus, the control flow gets more complicated. Furthermore, in a client-server architecture where the server offers services using polymorphic objects errors may occur. For example: A client is using a wrong message signature. The manifestation may be a totally different or unexpected behavior which is delivered back to the client. This fault can happen easily if parameter lists differ only in one primitive data type. Therefore, it is advised not only to verify the server code when it is modified with unit tests, but to check every interaction with the client as well.

If unit tests have been designed since the beginning of a project it should not be very difficult to adapt changed code when alterations are made to the original code. In fact the unit tests could help to understand polymorphic behavior better because they can act as a documentation for the source code and the more complex interactions of polymorphic objects. According to Ebner, more "common mistakes are ignoring responsibility, independent revision of its definition and usage, contract inconsistency, not provided method, method misuse ..." [Ebner04].
5. The Daidalos II Project

5.1 Introducing the Daidalos II Project

Daidalos is the abbreviation for "Designing Advanced network Interfaces for the Delivery and Administration of Location independent, Optimised personal Services". Daidalos II is the second phase of the Integrated Project Daidalos [URL: Daidalos1]. It is scheduled from November 2003 to December 2008. The Daidalos logo is shown in figure 7.

Today the users of terminal equipment such as cell phones, satellite phones, smart phones, PDAs, mobile computers or pagers\textsuperscript{26} are confronted with a vast number of service networks to which these devices connect. Even today there are so many networks available that the end user is confused by all the names, standards and procedures involved in connecting to these networks. The vision of Daidalos is that networks become transparent to the end user so that he is able to use his personal services independently at his location when at least one network is available. The underlying technology automatically connects him to one of the heterogeneous networks, resulting in a better manageable communication infrastructure for all participants. For this goal a pure IPv6 infrastructure will be used because it provides sufficient address space for the numerous devices which may connect to the network. The networks which are mentioned within the Daidalos project are:

- IEEE 802.11 (Wireless LAN), IEEE 802.15 (Wireless Personal Area Networks) and IEEE 802.16 (Broadband Wireless Metropolitan Area Networks)
- GSM / GPRS / UMTS / EDGE
- satellite networks
- Bluetooth
- DVB-S /-T /-H
- Beyond 3G (B3G)
- sensor networks

The project features five key concepts which are:

- Mobility Management; Authentication Authorization and Accounting (AAA); Resource Management, QoS and Security (MARQS)

\textsuperscript{26} Just to name a few; today there are also MP3 players with a Wireless LAN interface available.
5. The Daidalos II Project

- Virtual Identity (VID)
- Ubiquitous and Seamless Pervasiveness (USP)
- Seamless Integration of Broadcast (SIB)
- Federation, to allow service providers offering and receiving profitable services in a dynamic environment

The project is further divided into technical working packages (WP) to structure the different network and service layers which have to be researched and implemented. **WP1** is mainly an organizational package to ensure consistency in architecture and technical direction. **WP2** addresses the integration of heterogeneous networks using an IP infrastructure. The goal is to implement an architecture which is flexible, scalable, robust and optimized. **WP3**'s main goal is to create a user-centered pervasive platform which manages the services of Daidalos. Concerns like privacy and security are also addressed in this working package. **WP4** is intended as a validation and verification package to evaluate the work of work packages one to four. This includes feedback from expert groups and end-users. Qualitative and quantitative analyses are planned to prove the concepts of Daidalos II.

The following scenario example should make the vision of Daidalos quite obvious [D111].

Imagine the fictive person 'Bart' on a normal working day under the assumption that in the near future, cars, buildings, large streets and every other place in a city are connected to the Internet. The scenario starts in the morning when Bart is watching a newscast at his home. The newscast is automatically shown on the right monitor as he changes the rooms. Bart receives a voice call from his boss Hector on his PDA. When he answers the call, the newscast is automatically put on hold for later resumption. Bart switches the audio-only call from his PDA to video conferencing mode on his television screen in his living quarters. During this call he simultaneously receives additional data from Hector regarding 'Rosalyn' whom Bart is asked to pick up from the airport. While still in conversation with Hector over his PDA, Bart walks to his car to drive to the airport.

As the car recognizes Bart's approach via an identification chip integrated into his electronic device (or his clothing) the on-board electronic system is started with Bart's personal system preferences. When Bart enters the car he is able to resume the video conference with Hector as the car has the necessary equipment. The location data of the airport is automatically received over the active data link by the on-board navigation system and processed with the help of a GPS system. The system also reserves a parking space at the destination. When the car is started Bart's call is switched to sound-only mode as Bart is driving. Bart may also resume his newscast which has been paused earlier.

Arriving at the reserved parking space at the airport, Bart's company is charged with the parking fee and the on-board computer receives an update about Rosalyn's flight arrival time. When Bart leaves the car and locks it, all services are automatically transferred to his PDA. This device is also able to exchange location information (Buddy Finder) with Rosalyn's PDA to guide both of them to a meeting point.

During Bart's trip to the airport another application of the Daidalos software could be the exchange of traffic information with other cars. Thereby car accidents could be vastly reduced. If there is an acute accident the Daidalos system may inform local rescue units and make them aware of the exact location where the accident occurred. The system could
5. The Daidalos II Project

further be used by the emergency vehicles to keep emergency lanes clear.

This example scenario is also available as streaming video on the Daidalos website and described in the publicly available deliverable D111. This deliverable also contains another scenario explaining the impact of Daidalos on a university campus network where students exchange information about courses and learning materials as easily as making an appointment for dinner.

The improvement of Daidalos is mainly the connectivity of the Daidalos software to all the different types of networks over a pure IPv6 infrastructure and the seamless data exchange between the client nodes. Today, many Internet routers are 'IPv6 aware' but the protocol is still in a migration and testing phase. Also it is required that the network density, for example for WLANs\(^27\) or Metropolitan Area Networks (MAN), is strongly improved so that the signal quality is constantly assured\(^28\). Modern cars are already equipped with onboard computers and it is very likely that ad-hoc car networks will be added in the near future.

A great challenge for such a project are the security precautions which have to be taken to ensure data integrity and privacy. The end-user must have the ultimate authority to control personal profiles and the information other users may obtain. All available data also has to conform with international data protection laws.

This project is supported by the European Union with a fund of € 13.8 million. The total budget of the project amounts to € 22.1 million [URL: Daidalo].

5.2 Details about the obtained source code

Figure 8 shows the Daidalos II source tree of WP4 which was obtained in May 2006\(^29\). The directory "DaidalosPrototype" contains the structure for a prototypical Daidalos binary build for previewing and demonstration purposes.

\(^{27}\) A metropolitan wide WLAN for example has been planned for San Francisco [URL: News_1].
\(^{28}\) The Daidalos software can be configured to use any available GSM network as a fallback option.
\(^{29}\) The source code was provided by Dr. M. Ebner, Institute for Informatics, University Göttingen.
The Daidalos II Project

The folder "impl" contains source code for displaying the Daidalos software on mobile devices and the functionality for a DPA-engine. A DPA is the Daidalos version of a PDA. The difference between a normal PDA and a DPA is that a user can obtain a "time limited ownership of (part of or multiple) devices" [Sarma04].

The folder "interfaces" holds interface declarations and classes for the display service and the DPA service.

"MMSP" is the abbreviation for Multimedia Service Provisioning / Platform. It is responsible for handling all multimedia related content, for example additional data transfer when utilizing a video / audio call or the transfer of charging information to other Daidalos internal components.

"msgui-wp4" contains an unknown server – client architecture written in C++. It was not able to figure out its purpose as it is poorly commented and the Daidalos documentation documents does not mention it.

OSGI is "an execution environment for dynamically deploying and managing software components that delivers services and applications to networked devices" [Bitzer06]. Daidalos uses this framework for various components. "OSGITest" contains these files. "PSP-Container" also holds files based on OSGi but with the additional help from
Knopflerfish, an OSGi based framework [URL: Knopflerfish]. For general information about the OSGi Alliance see the projects website [URL: OSGi].

The Pervasive Service Platform ("PSP") is a key Daidalos component. It is responsible for managing pervasive services which play a central role in the Daidalos project. Subscription based services are supported, as well as ad-hoc based services. Additionally it offers third party providers a standardized interface to key PSP components, ensuring compatibility and simplifying software upgrades [Daidalos06]. The interface automatically assures data integrity and security for example when sensitive user data is exchanged.

The directory "ServiceBinder" contains useful information about its designation: "The Service Binder is a mechanism that simplifies development in the OSGi services platform as it automates service dependency management. [...] Service dependency management logic is, in general, complex and error prone. The Service Binder solves this problem by extracting service dependency management logic from the bundles and moving it into an execution environment that is deployed inside the framework as a standard bundle."

"SLP" is the abbreviation for Service Location Protocol [URL: RFC2608]. Daidalos adopts SLP and uses it to (un-)register services in a dynamic network environment.

Finally, there are three additional directories: "UIManagement" contains source code for the user interface management, like underlying databases, etc. "VAS" holds implementations for Value Added (broadcast) Services, like real time audio / video content or entertainment products. "Visualiser" is responsible for the graphic display of the Daidalos control software.

A short analysis of the composition of the source code is shown in figures 9 and 10. The complete CVS checkout contains approximately 20,000 files. A large number of the files is required for the CVS versioning system. Furthermore, the source code contains mainly Java related files (.java and .jar file extension). XML files are generally needed for automated building processes of the source code.

The outcome of the analysis of the disk space usage shows a more distributed diagram in figure 9. The CVS files require only a minimal amount of disk space. Also, the space consumption of XML files was so low that they have not been included in the diagram. Large quantities of bytes are used by Java source code files and Java archives, as expected. The main use of text files is for documentation purposes.

30 Figure 9 displays the real disk space usage, not including file system overhead.
6. Applicability of techniques to Daidalos II

6.1 Integration of techniques

In this chapter the previously presented techniques of unit testing will be exercised on the basis of the Daidalos II WP4 source code when possible. Since the Daidalos II is a highly integrated project with many dependencies and the source code in a state of development, specific code fragments will act as a demonstration for the different techniques.

The class DPAServiceInfo is a service class which is responsible for storing information about a service provider used on a DPA. The first approach to unit testing would be to test the correct object initialization and correct storage of internal parameters with assert methods.

Excerpt of DPAServiceInfo.java:

```java
import java.net.URL;

// @author Jason Finnegan (TSSG)
public class DPAServiceInfo {

    String serviceName;
    ...
    Float cost;
    URL serviceURL;
    int state; //Bundle ACTIVE / INSTALLED / UNINSTALLED etc
    boolean isLocal;

    public DPAServiceInfo(String name){
        serviceName = name;
    }
    public void setServiceName(String newserviceName) {
        serviceName = newserviceName;
    }
    public String getServiceName() {
        return serviceName;
    }
    public void setState(int state) {
        this.state = state;
    }
    public int getState() {
        return state;
    }
    public void setLocal(boolean isLocal) {
        this.isLocal = isLocal;
    }
    public boolean isLocal() {
        return isLocal;
    }
    ...
}
```
The corresponding unit test class `DPAServiceInfoTest.java`:

```java
import junit.framework.TestCase;

public class DPAServiceInfoTest extends TestCase {

    public void testsetState() {
        DPAServiceInfo isp = new DPAServiceInfo("object1");
        assertNotNull(isp);
        assertEquals("object1", isp.getServiceName());
        isp.setState(1);
        assertEquals(1, isp.getState());
    }

    public void testsetServiceName() {
        DPAServiceInfo isp = new DPAServiceInfo("object1");
        assertNotNull(isp);
        assertEquals("object1", isp.getServiceName());
        isp.setServiceName("Internet Service Provider");
        assertEquals("Internet Service Provider", isp.getServiceName());
    }

    public void testsetLocal() {
        DPAServiceInfo isp = new DPAServiceInfo("object1");
        assertNotNull(isp);
        assertEquals("object1", isp.getServiceName());
        isp.setLocal(true);
        assertTrue(isp.isLocal());
    }
}
```

In every test method it is tested that the objects equals the object which was requested by invoking `assertEquals`. Thereby, the objects constructor is tested. It follows a test of a concrete method. The test methods include tests for the state of the service (`setState / getState`), the service name (`setServiceName / getServiceName`) and a method which returns a boolean variable if the service is local (`setLocal / isLocal`). These are simple get / set methods but even when testing these there occurs some code duplication when creating the objects to work with. With the slight modification of moving the duplicated code into a `fixture` the readability of the code can be improved and the code duplication can be removed in the concerning test methods:

```java
private DPAServiceInfo isp;

    protected void setUp() {
        isp = new DPAServiceInfo("object1");
    }
```

The `fixture` `setUp()` is executed before every test method and therefore the object initialization in can be removed in every test method.

As an example to test correct behavior classes the Class `ContactList` is tested. Its primary function is to lookup a name from the virtual identity (VID) which is stored for every Daidalos user. For the lookup a SIP URI is required. This is done by using the Java class `Vector` which is available from the Java API. It stores the different profiles of the users. The name which was looked up is then written to the console output using a `toXMLString()` method.
However, the ContactList is based on another class called EndPoint. This class is only used to supply the basic strings type (of contact), name and URI which are required for a VID profile. Therefore, it is not given here. Since the class is static and contains hard-coded material such as contact identities, it suggests that it is still under development and not yet completed. But this is not every important to show the possibility to test this class. It also shows effectively that no main-method is required to test the concerning class. The class contains some more identities but they have been left out to improve readability.

The class ContactList.java:

```java
import java.util.Vector;

public class ContactList {
    public static String getVIDname(String sipURI){
        String name = "";
        Vector contactlist = new Vector();
        EndPoint contact = new EndPoint();
        
        contact.name="boss";
        contact.URI = "hector@daidalos.org";
        contactlist.add(contact);
        contact = new EndPoint();
        
        contact.name="wife";
        contact.URI = "jennyPDA@daidalos.org";
        contactlist.add(contact);
        
        for (int i=0; i < contactlist.size(); i++){
            EndPoint c = (EndPoint) contactlist.get(i);
            System.out.println("Checking Contactlist Endpoint: "+i+"" +c.getName()+" with URI: "+c.getURI());
            if (c.getURI().toLowerCase().equals(sipURI.toLowerCase())){
                name = c.getName();
                System.out.println("Found matching URI: "+c.toXMLString());
            }
        } 
        c = new EndPoint();
        System.out.println("returning name: "+name+"for SIPURI: "+sipURI);
        return name;
    }
}
```

The corresponding ContactListTest.java:

```java
public class ContactListTest extends TestCase {
    public void testgetVIDname(){
        String teststring1 = "hector@daidalos.org";
        String teststring2 = "jennyPDA@daidalos.org";
        String name = ContactList.getVIDname(teststring1);
        assertEquals("boss", name);
        String name2 = ContactList.getVIDname(teststring2);
        assertEquals("wife", name2);
    }
}
```

This is the typical approach for testing the correct behavior of the class under test.
Unfortunately, the obtained Daidalos II source code does not include exception handling yet. The basic exception classes are in fact available, but there are not yet thrown by any of the service classes. This may be the case due to the internationalization of the Daidalos project. It is most likely that error messages will be presented in different languages. Therefore, it is probable that the internationalization features are not yet complete. But because the exception handling is considered very important, the available exception classes will be used to illustrate the basic testing of exceptions. The following exception classes are taken from the Daidalos II security package.

Consider a UnauthenticatedUserException which is thrown when a user tries to log into a Daidalos service with the wrong password. Further consider a AccessDeniedException which may occur if a user tries to directly access services which he is not registered for. The test for such a behavior may look like this:

```java
public void testUserAccess throws Exception {
    user.setName("Bart");
    user.setPassword("Blank");
    user.registeredService("GPS", "BuddyFinder", "VideoCall");
    try {
        network.manager.useservice("GPS");
        fail("UnauthenticatedUserException expected");
    } catch (UnauthenticatedUserException expected) {} 
    user.setPassword("realPass");
    assertTrue(network.manager.useservice("GPS");
    try {
        network.manager.useservice("ParkingReservertion");
        fail("AccessDeniedException expected");
    } catch (AccessDeniedException expected) {} 
    assertEquals("Error Message", expected.getMessage());
    user.addRegisteredService("ParkingReservertion");
    assertTrue(network.manager.useservice("ParkingReservertion");
}
```

This approach is typical for testing exceptions. A fail() statement lets the test pass if the expected exception occurs. The user "Bart" was deliberately equipped with a wrong password to force the expected exception to occur. Important is the correct continuation of the program logic after an exception occurred. This can be assured by additional assert statements. It is also possible to examine the exception further by using an assertEquals() on the caught object like it is shown directly after the second catch statement. Further, it is not advisable to use any return statement within a catch clause. The return statement causes that a current method is quit instantly and additional tests or assertions would excluded. The throws Exception keyword in the method declaration makes sure that unexpected exceptions are passed up to the JUnit framework.

JUnit 3.8.x contains the class ExceptionTestCase. This method is rather complex and unintuitive in its usage. Tests cases have to extend ExceptionTestCase and must be used without a catch-try block. When such test cases are used in a test suite, every test method has to be manually added to the suite. This is very annoying if many exceptions have to be tested. Throughout the testing community the use of ExceptionTestCase is discouraged and it is no longer included in version 4 of JUnit.
In Daidalos there are many different groups of developers and each of them is responsible for a particular piece of the source code. When a group decides to use unit testing it is recommended that they combine their tests to a test suite. The number of test cases is not limited. When the integration of the different parts of the source code continue it is also possible to combine test suites of several group to superior test suite. Thereby, the state of the integration of the source code can be monitored using the complete set of test cases. The dependencies of source code which spans across groups can be tested. One condition is that each group creates a test suite which covers all test cases which were created by the group. Since JUnit automatically puts all tests cases within a test class into a test suite, it is recommended that this default behavior is kept. However, assigning the tests manually is also possible for example for categorized test suites. Test suites from test case classes can then be combined to a package-wide superclass:

```java
public class AllTests {

    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }

    public static Test suite( ) {
        TestSuite suite= new TestSuite("All JUnit Tests");
        suite.addTest(new TestSuite(DPAServiceInfoTest.class));
        suite.addTest(new TestSuite(ContactListTest.class));
        suite.addTest(new TestSuite(DPAServiceInfoTestWithFixture.class));
        ...
        return suite;
    }
}
```

To create test suite that contains every unit test has also simple psychological advantages. If many test cases have been created and they all succeed when a package wide test suite is run, this can have a positive effect on the confidence of the own work.

Example for test suite hierarchies within different packages:

- eu.ist.daidalos.pervasive.security → AllSecurityTests
- eu.ist..daidalos.pervasive.security.accesscontrol → AllAccesscontrolTests
- eu.ist.daidalos.pervasive.security.accesscontrol.authentication → AllAuthTests
Mock Objects

With the help of extensions to JUnit the manual creation of mock objects is not necessary. There are numerous tools available which can help in creating mock objects\(^{31}\). The basic features of the extensions are very similar. That is the reason why EasyMock is used as a reference example to the applicability\(^{32}\). EasyMock is a class library and it uses a record state to set up the expectations of an object. After the record state is completed the EasyMock framework notifies the programming environment that an expected behavior was not recognized and detailed error messages are shown. Afterwards the results can be verified using a \texttt{verify()} method call on the mock control object.

Supported is the set up of expected return values and exceptions, method call count checking and the checking of the execution order of methods. Installing EasyMock is an easy task: The class library is delivered as a .jar file which can be easily integrated into an IDE. In Eclipse for example, the class library has to be added to the project's build path. Further, EasyMock needs a Java interface where the concerning methods are declared. An example could be the privacy policy negotiation located in the package \texttt{eu.ist.daidalos.pervasive.framework.privacy.negotiator}. Its purpose is to register or unregister different privacy policies for a specific customer of the Daidalos service platform.

Excerpt of \texttt{PrivacyPolicyNegotiation.java}:

```java
public interface PrivacyPolicyNegotiation extends Negotiator {
    public void setPrivacyPolicy(PrivacyPolicy policy) throws PrivacyPolicyNegotiationBundleContextException;
    public void unsetPrivacyPolicy();
    public boolean negotiate(Role role) throws PrivacyPolicyAgreementFaultException,
            InvalidPrivacyPolicyStatementListException,
            NotAnOwnerException;
    void setPeerCustomer(Identifier pc);
    Identifier getPeerCustomer();
    public void register();
    public void unregister();
    public void breakNegotiation();
}
```

If a class under test has to use these external methods which may not yet have an implementation, a mock object can be set up that simulates behavior. A class under test may look like this:

```java
public class HandlePrivacyPolicy {
    public void addHandler(PrivacyPolicyNegotiation handler);
    public boolean acceptPolicy(PrivacyPolicy policy);
    public boolean removePolicy(PrivacyPolicy policy);
    public boolean cancelPolicyNegotiation(PrivacyPolicy policy);
}
```

\(^{31}\) See appendix B for an overview of available tools.
\(^{32}\) Another reason to use EasyMock is because the development is still continuing and releases are up to date.
In the JUnit test case the mock objects and its controls can then be set up using the previously discussed techniques:

The class HandlePrivacyPolicyTest.java (example 1):

```java
import junit.framework.TestCase;
import org.easymock.MockControl;

public class HandlePrivacyPolicyTest extends TestCase {

private HandlePrivacyPolicy policy;
private MockControl control;
private PrivacyPolicyNegotiation mock;

protected void setUp() {
    control =
        MockControl.createControl(PrivacyPolicyNegotiation.class);
    mock = (PrivacyPolicyNegotiation)control.getMock();
    policy = new HandlePrivacyPolicy();
    policy.addHandler(mock);
    PrivacyPolicy strictPrivacyPolicy;

    public void testacceptPolicy() {
        // behavior specification will be put here
        control.replay();
        policy.acceptPolicy(strictPrivacyPolicy);
    }
}
```

The procedure to get a mock object consists of four steps:

- Getting a MockControl object for the interface which should be simulated as done in line 1 of the setUp() method.
- Getting a mock object from the MockControl object as done in line 2 of the setUp() method.
- Specifying a behavior for the mock object, which was not yet done in the testacceptPolicy() method. This part of setting up a mock object is also called 'record state'.
- Activation of the mock object by invoking the replay() method of the control object.

If the above testacceptPolicy() is run the mock object will throw an error because if the acceptPolicy method issues a call to a method of the original interface this is an unexpected behavior as it was not set up before.

As the setting up of a mock object is always the same, the setUp() fixture will not be included in the examples from now. Now, the mock object has been set up properly and is ready to receive instructions of the desired behavior.

An alternative object which also verifies the order of method execution can also be requested. This type of mock is called a strict mock. It can be requested by using MockControl.createStrictControl().
Test with adding behavior to a mock object (example 2):

```java
public void testAcceptPolicy() {
    mock.negotiate(role);
    mock.setPrivacyPolicy(strictPrivacyPolicy);
    control.replay();
    policy.acceptPolicy(strictPrivacyPolicy);
}
```

If a policy is accepted it is expected that a call to `negotiate()` and `setPrivacyPolicy()` precede this acceptance. Therefore, they are set in the record state of the mock object. If the test does something unexpected, like requesting another policy as shown in example 3, the mock object gives a detailed feedback about what happened:

Test which behaves unexpected (example 3):

```java
public void testAcceptPolicy() {
    mock.negotiate(role);
    mock.setPrivacyPolicy(strictPrivacyPolicy);
    control.replay();
    policy.acceptPolicy(loosePrivacyPolicy);
}
```

The output gives detailed feedback about the error (example 4):

```
junit.framework.AssertionFailedError:
Unexpected method call: setPrivacyPolicy(loosePrivacyPolicy):
    setPrivacyPolicy(loosePrivacyPolicy): expected: 0, actual: 1
    setPrivacyPolicy(strictPrivacyPolicy): expected: 1, actual: 0
...```

To verify that a specified behavior is actually used, the self-verification technique is used. The mock object is able to verify that expected method calls have actually been used by calling the `verify()` method on the mock object. Otherwise tests could pass even if no method is called on the mock object.

The usage of the `verify()` method is always recommended to ensure the expected behavior (example 5):

```java
public void testAcceptPolicy() {
    mock.negotiate(role);
    mock.setPrivacyPolicy(strictPrivacyPolicy);
    control.replay();
    policy.acceptPolicy(strictPrivacyPolicy);
    control.verify();
}
```

An additional feature is the specification of the number of method calls. Often it is important that specific methods are called only a defined number of times. For example it would make no sense to call the `setPrivacyPolicy()` method more then once. If this behavior is observed it is likely that the class under test contains an error. The most simple approach is to tell the mock object how many method calls it would expect by repeating the statement when the mock object is in a record state. To avoid code duplication the method `setVoidCallable(int times)` is provided by the library. It has to be called immediately after the concerning method as shown in example 6. If the requirements are not met an exception similar to example 4 is displayed.
6. Applicability of techniques to Daidalos II

Setting the explicit number of method calls (example 6):

```java
public void testAcceptPolicy() {
    mock.negotiate(role);
    mock.setPrivacyPolicy(strictPrivacyPolicy);
    control.setVoidCallable(1);
    control.replay();
    policy.acceptPolicy(strictPrivacyPolicy);
    control.verify();
}
```

Sometimes it is not applicable that a specific number of calls occurs. Thereby, it is supported that the number of method calls is between two variables. This is accomplished by using the control method with the signature `setVoidCallable(int minCount, int maxCount)`. If one is not sure about the value of the variables, for example if the `getPeerCustomer()` method is called at least once but if a network error occurs it may be called several times, EasyMock also supports this by introducing Range Objects. Range objects are called on the control object as usual and support the use of exactly one call, one or more calls or zero or more (unlimited) calls.

Further, a specific return value of a method can be inflicted on a method. The EasyMock library supports this property by providing the following control methods:

- `setReturnValue([type] value)` and
- `setReturnValue([type] value, int times)`

Specific return values can be defined on the mock object. For example trying to get a loose privacy policy fails but results in a successful test (example 7):

```java
public void testAcceptPolicy() {
    mock.negotiate(role);
    control.setReturnValue(false);
    mock.setPrivacyPolicy(strictPrivacyPolicy);
    control.replay();
    assertFalse(policy.acceptPolicy(loosePrivacyPolicy));
    control.verify();
}
```

A useful feature, especially for working with exceptions is the short form of some commands on the mock object. Normally, to define an expected exception the method with the signature `setThrowable(Throwable throwable)` is used. In the majority of cases an exception occurs when a specific return value is expected. Example 8 shows both approaches. However, the short version works only for return types other than void.
6. Applicability of techniques to Daidalos II

Example 8:

```java
public void testacceptPolicy() {
    mock.negotiate(role);
    control.setReturnValue(false);
    mock.setThrowable(new PrivacyPolicyAgreementFaultException());
    control.replay();
    assertFalse(policy.acceptPolicy(loosePrivacyPolicy));
    control.verify();
}

public void testacceptPolicy() {
    mock.negotiate(role);
    control.expectAndThrow(mock.negotiate(false), new 
        PrivacyPolicyAgreementFaultException());
    control.replay();
    assertFalse(policy.acceptPolicy(loosePrivacyPolicy));
    control.verify();
}
```

The presented usage of mock objects can act as a blueprint for the concept of mock objects. Other libraries available differ slightly but the basic principles are the same. Further, the record and replay features with additional verification is considered to be demonstrative. A risk which occurs when using mock objects is that the mock objects can contain errors within themselves. This can easily occur when return types of methods are mixed up. But the chance that this occurs is relatively small because of the verification and well designed error messages.

6.2 Automation

"In Daidalos I developers had to create their own stub / mock objects for external code."\textsuperscript{33}

This method is very time consuming because sometimes the specification of the resources one has to rely on is still incomplete.

The creation of mock objects can be automated to save time and work during development cycles. It is suggested that developers create mock objects for themselves using one of the available automation tools. This could also enable other developers to easily access previously created mock objects when they are made available on a shared platform. It is recommended that one automation tool for mock objects is selected so that the mock objects share a common syntax. Tools like MockMaker and MockCreator are candidates for such an approach.

Test suites can be automatically created with the help of the JUnit addons, rMock and GSBase when each source code package contains the structure presented in this document.

Test cases can be easily created automatically using modern IDEs like Eclipse or JBuilder. An additional interesting tool for this task is JUnitDoclet. It uses Javadoc comments to gather informations about a class. If the source code has appropriate Javadoc comments JUnitDoclet is able to create a skeleton of test cases and test suites.

\textsuperscript{33} Personal communication with Dr. M. Ebner, Georg-August-University Göttingen.
As the desired test coverage of a system should be 100%\textsuperscript{34} it may be useful to confirm how many of the source code is actually covered by unit tests. With the help of JCoverage a code coverage analysis can be conducted. The analysis is presented with detailed information about the source code.

Finally, defective unit tests can be detected by Jester by using code instrumentation and a mutation approach. As mutations alter the source code, Jester should only be allowed to work on a copy of the source code.

### 6.3 Feasibility

The feasibility of applying unit tests to the Daidalos project is generally considered possible. However, it is difficult to apply unit tests when the project is in a state of development and unit tests were not created from the beginning of the project. The unique architecture of Daidalos with all its service containers to simplify the integration of third party products makes it difficult to apply unit tests to every part of the project.

The unit test approach is reasonable but it is not the final solution to remove every defect in source code. Functional tests and integration tests are still necessary because unit tests cannot uncover every dependency between interacting objects.

Mock objects are a powerful approach when certain packages are not yet complete but some parties involved already require unit tests.

Certain pitfalls may occur when trying to introduce new techniques in a project which is already progressing. For example the inflexibility to change the underlying Java SDK version may require to used past versions of testing and automation tools. Useful features of those tools may have been introduced since then which could not be used within the project.

However, unit testing in Daidalos II can lead to more stable classes and source code which communicates its properties and functions clearly.

\textsuperscript{34} A 100% test coverage may not be achieved at the end of the development, but the aim towards it can further improve the percentage.
7. Conclusions

7.1 Summary and discussion

The past chapters covered component testing techniques with the help of the xUnit model. Before the Daidalos II project was introduced, common testing vocabulary and general software testing techniques were presented. Their advantages and disadvantages in the context of a modern software development process were introduced. Test driven development and its origin in extreme programming was shown.

JUnit, the reference implementation of the xUnit model was presented in detail. The requirements of a testing framework were given and the internal structure of JUnit was discussed as well as the usage of the JUnit framework. Whenever possible, references to continuative literature were given. Testing methodology was presented with the help of practical examples from the literature.

The advantages of using different types of objects with reduced functionality were presented. The problems which arise from properties of object-oriented languages like inheritance and polymorphism were also explained.

An overview of EU Project Daidalos II and its vision to connect mobile devices in a pervasive way consolidated under the Daidalos service platform was given and discussed. Daidalos has the aim to interact with numerous types of networks. The technical details were presented as well as detailed scenario descriptions of the possible future functionality. An analysis of the components of the work package 4 source code completes the Daidalos II project description.

The principles of unit tests were then explained with the help of the Daidalos II source code where it was possible. Specific code fragments were chosen to demonstrate advantages of unit testing within software projects. Attention was also given to automation, which can be useful in large software projects to save development time. The automation tool easyMock which helps creating mock objects in such a large project as Daidalos was presented. Final thoughts on the feasibility of the external evaluation of the testing within the Daidalos II project as it was in May 2006 completes this bachelor thesis.

Discussion

At the beginning of the work on this bachelor thesis the author was only familiar with the very basic principles of testing software. As the work continued the great importance of testing software was recognized as well as the different problems. Before the studies of testing concepts, the approach of Extreme Programming was considered a heavyweight technique which requires lots of time and is difficult to learn, perhaps also because of the unusual name. But as the studies continued, the possible impact of Extreme Programming and test driven development was recognized. It is in fact a lightweight approach whose principles are not difficult at all. Maybe a bit unfamiliar but possibly only because the author has grown up with traditional testing techniques. The test driven approach, which is an essential part of Extreme Programming, is able to push the development of testing techniques in the right direction. Not only because tests act as a documentation for source
Conclusions

code, but because the removal of defects in the early stages of a software development process is cost-effective for every company which wants to produce software for the open market. The speculation is that it takes some time for the TDD approach to become omnipresent in the field of software tests. Since it arose only about six years ago today's software developers may still think of the traditional way as the better one. But universities have started to integrate the concepts of XP and TDD into their lectures and practical courses and today's students are aware of the potential of the new techniques. Some companies have recognized the potential of TDD as well and are using it as presented in chapter 2. But in this context also carefulness has to be practiced. A developer should not be forced to use a specific form of testing or developing software. This could lead to the refusal of new approaches.

Unit testing is considered a reasonable procedure. The argument that the extra work of writing unit tests slows developers down is flawed. When developers are writing source code they are very familiar with the internal structure of their own code. By verifying the modules by using unit tests developers can uncover defects they might not have thought of. The trust in their own product is heightened. Additionally, when it comes to presenting the development results or integrating the source code into a larger context, unit test can act as a very good demonstration of the abilities of the developer and thus the source code. Further, the maintenance of a software product can be improved by using unit tests. If an error is discovered in an integration or system test and the source code is changed, existing unit tests can uncover unwanted side effects when modifying existing source code.

Advanced techniques in the context of unit testing have been developed by several parties which can be a great help especially in the context of different versions of mock objects. Unfortunately, not every technique can be presented within the limited scope of a bachelor thesis. But in the given literature additional topics are discussed. For example testing persistent objects, threads, distributed applications and Java Enterprise applications.

The Daidalos project is a very interesting and ambitious project within its area of application. However, the concept of unit testing has not totally reached this project. The overall unit test coverage is approximately five percent or lower (as the date of May 2006). As Daidalos uses the architecture of the OSGi model to make it easier for third party providers to integrate their own products into Daidalos, it complicates the applicability of unit tests. Further studies should assess whether a testing approach starting at the beginning of future EU projects is more effective.

However, this document also wants to give an overview of possible approaches to unit testing. These techniques could be integrated at any time into an existing project when they are properly introduced to the people who are involved.
7. Conclusions

7.2 Perspective to JUnit 4

Although JUnit version 4 is not applicable to the Daidalos II project because the Java API cannot be changed during the development, it is available since February 2006. JUnit 4.x requires the Java 2 Platform Edition 5.0 [URL: Java5]. The improvements are presented in this chapter. Java 5, which was published in May 2006, will be used in the future together with JUnit 4 only. But JUnit 4 is backward compatible with tests written for previous versions of JUnit. Old test cases can be run using the JUnit4TestAdapter which can be found in the package junit.framework.

Perhaps the most important new feature in JUnit 4 is the support for Annotations. These new language constructs were introduced in Java 5 to address several inconveniences, for example the sometimes required 'side files'. Annotations allow to place custom metadata within the source code. An Annotation is marked with preceding '@'-character. JUnit uses this concept for example to declare that a method is actually a test method:

```
@Test public void EmptyBuffer() {...}
```

But Annotations have also been introduced for test fixtures. The Annotations for setUp() and tearDown() are Before and After. So the programmer can choose his own method naming for (de-)initializations. This also addresses the problem that unwanted side effects could occur if inherited setUp()-methods are not explicitly called.

New are the Annotations BeforeClass and AfterClass. Methods flagged like this can be used to do jobs which only have to be executed once for all test cases in a class. An example could be the initialization of a database connection.

Another useful Annotation is @ignore which can be used to exclude test cases from their execution. This approach may be especially useful when some tests are written before the source code is. The function of a method may be already clear due to its naming, but it is not implemented yet:

```
@Ignore("This method is not implemented yet...");
```

The string directly after the Annotation is called a parameter. The Annotation @Test possesses a parameter which can be used to test checked exceptions. Using this approach testing checked exceptions gets a lot easier than in the previous versions of JUnit where a try / catch block and a fail() assert had to be used:

```
@Test(expected=ArrayOutOfBoundsException.class)
```

Also new is testing with timeouts. The parameter timeout is used to specify the amount of time in milliseconds the test has to complete. If the given time limit is exceeded the tests fails. It is now also possible to use 'static imports' in JUnit 4.

"The static import construct allows unqualified access to static members without inheriting from the type containing the static members. Instead, the program imports the members, either individually [...] or en masse [with the * operator]" [URL: Java5_1]. This means for JUnit, that static imports can be used to easily integrate custom assertions into the source code.

Another important change is the introduction of parameterized tests which can for

35 'Side files' are files which have to be kept alongside and parallel to the source code
7. Conclusions

example reduce code duplication. With this approach, data can be passed to test methods by using the `@RunWith` Annotation. Other changes in JUnit 4 are the handling of test suites, the discontinuation of the differentiation between failures and errors, the doubling of the namespaces\(^{36}\) `junit.*` and `org.junit.*` and the `assertEquals()` implementation has been changed to support the comparison between arrays\(^{37}\). For current changes visit www.junit.org or consult the documentation which is included in every JUnit release package. All other new features and enhancements regarding Java 2 version 5, such as Autoboxing or variable-length argument lists, can be found under [URL: Java5_News].

\(^{36}\) The temporary doubling of the namespaces was done for compatibility reasons.

\(^{37}\) Multi-dimensional arrays are not processed correctly by `assertEquals` (JUnit 4.1).
Appendix

A. JUnit setup instructions

The latest version of JUnit is available on the project's website [URL: JUnit]. However, one can be in need of getting an earlier version. For example if the current development of a project is based on an earlier Java SDK. The current version of JUnit (4.1) claims to be backward compatible, but tests developed with an earlier version have to be slightly rewritten. The fact is that JUnit 4.0 requires Java Enterprise Edition SDK 5.0 and does not run with previous versions of Java. All versions of JUnit can be downloaded from JUnit's project site on sourceforge.net [URL: JUnitProj].

JUnit comes as a .zip file and needs to be unzipped to its location. On Windows it is important that the complete directory tree where JUnit is unzipped does not contain any spaces. Otherwise some tests can fail. On Linux and Unix derivatives it is recommended to put JUnit in /opt. After doing so the Java class path variable has to be changed so that it includes the JUnit path. On Windows this is done by editing the environment variable CLASSPATH under “System Properties → Advanced”. On Unix and Linux systems type in a shell:

```
# export CLASSPATH=/opt/junit3.8.2
```

If you want to preserve your existing CLASSPATH you should check it first, for example with

```
# set | grep CLASSPATH
```

and then append the JUnit path to your existing path.

Self-Testing with JUnit's integrated components

JUnit 3.8.2 – the version this document uses – comes with a series of example tests and self tests which are located in the junit/samples and junit/tests subdirectory. If one wants to quickly have a look at source code examples these tests are worth a look at because they are well documented and structured by the authors of JUnit. These tests are executed via a TestRunner which implements the main()-method. There are three different test runners available:

- `junt.textui.TestRunner` – simply writes the result of a test to the standard output (stdout). It displays a dot for a successful unit test and an 'E' for an error or an 'F' for a failure
- `junit.awtui.TestRunner` – a simple AWT-based graphical tool for displaying test results
- `junit.swingui.TestRunner` – a more sophisticated Swing based graphical tool

Type the following commands into a shell to execute the appropriate test runner:

```
38 For example the ClassLoaderTest.testJarClassLoading() test
```
A. JUnit setup instructions

Windows:

```bash
> java -cp junit.jar:. junit.[ textui | awtui | swingui ].TestRunner \ 
  junit.samples.AllTests
```

Unix:

```bash
# java -cp junit.jar:. junit.[ textui | awtui | swingui ].TestRunner \ 
  junit.samples.AllTests
```

Self-Testing JUnit with Eclipse

Running or executing JUnit from the command line can be useful if you use a text editor like Emacs or an editor without JUnit support. However, more and more developers are using an Integrated Development Environment (IDE) to develop their applications. These IDEs can also be useful when large scale enterprise applications are developed because it is much easier to manage large code repositories. For example Eclipse 3.1 comes with a full JUnit support\(^\text{39}\) and many other IDEs also have a good JUnit support because of its popularity. The following example will show how the JUnit samples are run from within the eclipse environment. However, one has to follow the steps from 1.1 – Setting up JUnit because Eclipse does not contain all the samples and the latest pre JUnit 4 version (3.8.2) which is used in this document.

1. First you have to create a new project by selecting File → New → Project

2. Select Java Project from the list

3. The next screen lets you enter a project name. Additionally you have to specify JUnit's source directory so that Eclipse can map the existing files to an Eclipse

\(^{39}\) This is no surprise because Eric Gamma is a key member of the Eclipse team.
A. JUnit setup instructions

project as shown in figure 12.

4. When pressing the next-button Eclipse shows the source tree of the given directory and allows you to specify some custom options. Be sure that the 'Libraries' – tab contains JUnit's library – junit.jar. Maybe you have to add it via the button 'add Jars...' (the file is located in JUnit's root directory).

5. Click Finish. Eclipse creates the project in your workspace and compiles the project. If Eclipse detects errors or warnings while checking the project these are shown in the 'Problems'-view.

Now each test can be run from within Eclipse. AllTests.java is a container for running different test suites which itself contain the unit tests. To see if JUnit is programmed well and the installation was successful it is recommended to run this file first. To execute these tests select AllTests.java in the package explorer (figure 14) and select 'Run as → JUnit test' from the context menu.

The integrated JUnit GUI should then show a 'green bar' meaning that all tests ran successfully. Some other details like execution time, number of tests run, number of errors and failure trace complete the feedback of the tests (figure 13).
A. JUnit setup instructions

Figure 14: The package explorer showing the JUnit source tree

Figure 15: The new JUnit Test Case creation window
This small appendix presents a short overview about some famous JUnit extension which were either recommended in the given literature or throughout the web. Where it was possible the extensions are assigned to a category to make the decision easier which extension is really needed. The descriptions were mainly taken from the original web sites. Some tools are under heavy development, so the list of extensions are given here without any version numbers. As always it is recommend to use the latest (stable) version. Another overview is available on [URL: JUnit_Ex]. However, this index may be partially outdated.

### Generic extensions

- **GSBase Testing classes**
  Collection of classes that are helpful when writing JUnit test cases. It includes a `RecursiveTestSuite` (which searches a directory structure for subclasses of `TestCase`), an `OrderedTestSuite` (for controlling the order of execution of the tests), an `EventCatcher`, an `EqualsTester` (testing equality of objects) and some basic extensions to `TestCase`.  

- **JUnit-addons**
  JUnit-addons is a collection of classes to facilitate the use of JUnit.  

- **Log4Unit**
  Log4Unit is a JUnit extension combining JUnit with Log4J. The intention is to create test protocols for JUnit. JUnit is asymmetrical in the sense that it focusses on the documentation of test failures and errors. Successful execution of a test case is not further documented. In order to obtain a test protocol that documents the initial settings, the test case execution and its results, a logging component is required. Log4J as the current de facto standard is selected for this extension.  
  [http://www.openfuture.de/Log4Unit](http://www.openfuture.de/Log4Unit)

### Automation tools

- **JesTer**
  It finds code that is not covered by tests. JesTer makes some change to your code, runs your tests, and if the tests pass JesTer displays a message saying what it changed. JesTer is different than code coverage tools, because it can find code that is executed by the running of tests but not actually tested. However, JesTer is not meant as a replacement for code coverage tools, merely as a complementary approach.  
  [http://jester.sourceforge.net](http://jester.sourceforge.net)
JUnit extensions

- **JUnitDoclet**
  JUnitDoclet generates skeletons of TestCases based on your application source code. And it supports you to reorganize tests during refactoring.
  http://www.junitdoclet.org

- **JCoverage**
  JCoverage is a free code-coverage tool for that allows to measure the effectiveness of Java tests and how much of a software program's code has been tested. JCoverage identifies how many times each line of code in your application has been executed and you can see which parts of your software remain untested. After instrumenting your code and running your tests, a report is generated allowing you to view information coverage figures from a project level right down to the individual line of code.
  http://www.jcoverage.com

Extensions for mock objects

- **jMock**
  jMock is a lightweight library for testing Java code using mock objects.
  http://jmock.codehaus.org

- **EasyMock**
  EasyMock provides Mock Objects for interfaces in JUnit tests by generating them on the fly using Java's proxy mechanism. Due to EasyMock's unique style of recording expectations, most refactorings will not affect the Mock Objects.
  http://www.easymock.org

- **MockMaker**
  MockMaker is a program for creating source code for mock object classes. Given an interface, it writes the source code for a mock object class that implements the interface and allows instances of that class to have expectations set about how many times a method is called, what parameters each method is called with, and to pre-set return values for methods. In many cases (possibly most cases), the classes produced by MockMaker are exactly what you want a mock class to do.
  http://mockmaker.sourceforge.net

- **MockCreator**
  The main idea of MockCreator development was to reduce knowledge of real code logic required to write a test. When a number of tests reaches a few hundred, it's often becomes too complex to support them since minor changes in real code often require to alter a bunch of unit tests. This is because test code often relays too much on, say, order of calls to setters (which supposed to be independent) or exact value of parameter (when actually it can be slightly more or less, like current time).
  http://mockcreator.sourceforge.net

- **rMock**
  rMock is a Java mock object framework to use with jUnit. rMock has support for a
setup-modify-run-verify workflow when writing jUnit tests. It integrates better with IDE refactoring support and allows designing classes and interfaces in a true test-first fashion. rMock has the flexibility and power of jMock, while the recording of mock calls is influenced by EasyMock.
http://rmock.sourceforge.net

Extensions for graphical user interfaces

- **Abbot**
  Abbot is a scripted Java GUI testing framework that supports recording and playback of semantic user events, as well as component state evaluation for both unit and functional testing. http://abbot.sourceforge.net

- **JFCUnit**
  An extension to the JUnit framework that enables you to execute unit tests against code that presents a Swing based interface.
  https://sourceforge.net/projects/jfcunit

- **Marathon**
  Marathon is a GUI-test tool that allows you to play and record scripts against a java swing ui. It's written in Java, and uses python as its scripting language - the emphasis being on an extremely simple, readable syntax that customers/testers/analysts feel comfortable with. It is full fledged python, so it is also extremely powerful and customizable for developer-types.
  http://marathonman.sourceforge.net
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>Authentication, Authorization and Accounting</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWT</td>
<td>Abstract Windowing Toolkit</td>
</tr>
<tr>
<td>B3G</td>
<td>Beyond Third Generation technology</td>
</tr>
<tr>
<td>BDUF</td>
<td>Big Design Up Front</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUT</td>
<td>Class Under Test</td>
</tr>
<tr>
<td>DPA</td>
<td>Daidalos Personal Assistant</td>
</tr>
<tr>
<td>DVB-T/-H/-S</td>
<td>Digital Video Broadcasting (terrestrial -T, handhelds: -H, satellite: -S)</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile Communications</td>
</tr>
<tr>
<td>IP (v6)</td>
<td>Internet Protocol (Version 6)</td>
</tr>
<tr>
<td>MAN</td>
<td>Metropolitan Area Network</td>
</tr>
<tr>
<td>MMSP</td>
<td>Multimedia Service Provisioning / Platform</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>OSGi</td>
<td>Open Services Gateway initiative</td>
</tr>
<tr>
<td>OUT</td>
<td>Object Under Test</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>PSP</td>
<td>Pervasive Service Platform</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>TDD</td>
<td>Test Driven Development</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications System</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VID</td>
<td>Virtual IDentity</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network</td>
</tr>
<tr>
<td>WP</td>
<td>Working package</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming</td>
</tr>
</tbody>
</table>
Bibliography


Online Resources

[URL: Agile_Manifesto]
agilemanifesto.org

[URL: AgileAlliance]
www.agilealliance.org/

[URL: Daidalos]
www.ist-daidalos.org/

[URL: Daidalos1]
www.ist-daidalos.org/daten/publications/EU-leaflet/EU-project_Daidalos_II_Summary.pdf

[URL: Java5_1]
java.sun.com/j2se/1.5.0/docs/guide/language/static-import.html

[URL: Java5_News]
java.sun.com/j2se/1.5.0/docs/relnotes/features.html

[URL: Java5]
java.sun.com/javaee/technologies/javae5.jsp

[URL: JavaDocAssert]
junit.sourceforge.net/javadoc/junit/framework/Assert.html

[URL: JUnit_Ex]
www.junit.org/news/extension/index.htm

[URL: JUnit]
www.junit.org/index.htm

[URL: JUnitProj]
prdownloads.sourceforge.net/junit/

[URL: Knopflerfish]
www.knopflerfish.org

[URL: OSGi]
www.osgi.org

[URL: Realtime]
www.real-time.org

[URL: Reflection]
java.sun.com/docs/books/tutorial/reflect
Online Resources

[URL: RFC2608]
tools.ietf.org/html/rfc2608

[URL: Royce70]

[URL: Spiral]
www.sce.carleton.ca/faculty/ajila/4106-5006/Spiral%20Model%20Boehm.pdf

[URL: TestDriven]
www.testdriven.com

[URL: TestingCat]
www.testing.com/writings/short-catalog.pdf

[URL: TMG_Unde]
www.tmg.informatik.uni-goettingen.de

[URL: Waterfall04]
www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=110

[URL: Wiki_1]
en.wikipedia.org/wiki/Development_stage

[URL: Wiki_2]
en.wikipedia.org/wiki/Waterfall_model

[URL: Wiki_3]
en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming

[URL: XP_Dead]
martinfowler.com/articles/designDead.html

[URL: XP]
xprogramming.com

[URL: xUnitPatterns]
xunitpatterns.com/Dummy%20Object.html