Development and Implementation of an Automated Test Environment for a Data Processing System

Master's Thesis

Institute for Software Technology
Graz University of Technology

Andreas Mitterer
andreas.mitterer@student.tugraz.at

Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Zusammenfassung

Diese Diplomarbeit behandelt die Entwicklung eines Testautomatisierungsprogramms und die Änderung von bestehenden Arbeitsprozessen bei der Softwareentwicklung und Testtätigkeit. Sie entstand in Zusammenarbeit mit der Firma RunningBall Sports Information GmbH. Der praktische Teil wurde in Kooperation mit dieser Firma umgesetzt. Das Kerngeschäft des Unternehmens besteht aus der Echtzeit-Datenübertragung von Sportereignissen über das Internet. Der operative Standpunkt von RunningBall befindet sich in Graz. Die Firma ist ein junges Unternehmen und es gibt nicht viele Softwareprodukte, die mit umfassenden Tests ausgestattet sind. Daher bietet sich hier das perfekte Umfeld um ein Testautomatisierungsprogramm zu entwickeln.

Den Zugang zu dem Wissensgebiet bilden Nachforschungen über den theoretischen Hintergrund von Softwaretests, Testautomatisierung und Metriken. Viele Werkzeuge, welche zu einer automatisierten Testumgebung beitragen, sind bereits verfügbar; die geeignetsten wurden ausgewählt und untersucht. Die gesammelte Erfahrung und ausgewählte Werkzeuge wurden in der Folge in einem Praxisprojekt angewendet.

Teil eins dieser Diplomarbeit behandelt den theoretischen Hintergrund von Softwaretests - Testbereiche und -methoden werden erläutert. Danach folgt ein umfassendes Kapitel über Metriken. Metriken stellen ein unverzichtbares Instrument für das Testen und vor allem zur Ermittlung des Fortschritts und der Qualität von Tests dar. Das Kapitel über Testautomatisierung endet mit einem Abschnitt über die Kriterien, wann ein Test beendet ist und ausreichend getestet wurde. Im nächsten Kapitel werden Werkzeuge für eine Continuous Integration Umgebung beschrieben. Werden diese Werkzeuge entsprechend ihres Verwendungszweckes eingesetzt, können sie in eine leistungsfähige Continuous Integration Umgebung zusammengefügt werden.

Der zweite Teil gliedert sich in drei Hauptkapitel. Das erste Kapitel erläutert den Projekthintergrund. Das umfasst eine Beschreibung der Firma RunningBall, einen Abschnitt über Fachbegriffe aus diesem Anwendungsgebiet und den Status Quo bezüglich Testaktivitäten bei der Firma. Weiters werden die Projektzieldefinition erklärt und ein Firmenprofil von RunningBall erstellt. Das zweite Kapitel zeigt die Evolutionsschritte des Testautomatisierungsprogramms und schildert die Hintergründe, warum das Projekt in diesen Phasen abgewickelt wurde. Das dritte Kapitel beinhaltet die Resultate und beleuchtet diese von unterschiedlichen Standpunkten: quantifizierbare Ergebnisse, beobachtete Veränderungen des Entwicklungsprozesses bei RunningBall und Rückmeldung der vom Testautomatisierungsprogramm betroffenen Personen.

Abstract

This diploma thesis deals with the implementation of a test automation program, and the change of existing work processes of software development and test activities. It came into life in cooperation with the company RunningBall Sports Information GmbH. The practical part was implemented at the company. The company focuses on sport live data transferred over the internet, and its operational site is located in Graz. RunningBall and its software development department are young, and there are not many applications that benefit from an extensive test coverage. So this is the perfect environment to implement an automated test tool.

The approach to the knowledge field is to do research on the theoretical background of software testing, test automation and metrics at the beginning. There are many tools that contribute to an automated test environment. The most suitable ones are selected and investigated. The gained knowledge as well as the chosen tool set are then applied to a real-life project.

This document is divided into two parts. The first part of this thesis deals with the theoretical background of software testing. Test scopes and test methods are discussed. After that follows a comprehensive chapter about metrics. Metrics are vital for testing, and most of all for assessing the progress and quality of tests. At the end of the chapter dealing with test automation there is a section dealing with the criteria when a test is over and if it has been tested enough. Later on tools for a good continuous integration environment are listed. Using these tools in a way that meets their purpose they can be combined into a continuous integration environment.

In part two there are three main chapters. The first one presents the project background. This gives a description of the company RunningBall, a section about special terms used in this application area, and the status quo at the company regarding testing. Also, the project goals will be defined and explained, and a company profile of RunningBall is provided. The second chapter shows the evolution of the Automated Test Environment project and the reasons why the described project phases have been chosen. The third chapter contains the outcome of the project and outlines it from different perspectives: quantifiable results, observed changes in the development process at RunningBall, and feedback from people affected by the Automated Test Environment.

Acknowledgements

My gratitude goes to DI (FH) Christian Ringhofer who came up with the idea of implementing a test automation tool, guided me through the project with essential hints and helped me out whenever I got stuck in implementation issues. Aidan Mc Gettrick reviewed this thesis as a native English speaker. Thank you for that. Furthermore I would like to thank all developers, testers, the project manager and everybody else at RunningBall that made this project possible and who contributed towards it.

I would like to express my thanks to Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany for being my mentor during my master studies and supervising this diploma thesis. I went to our meetings with a head full of problems, and left them inspired and motivated for the upcoming tasks.

A special thanks goes to my parents Maria and Johann Mitterer. You gave me the chance to attend the Graz University of Technology and always provided me with a place of recreation whenever I needed it.

Most of all I would like to thank my wife Anke. Your support was and is essential and can be spelled out with just four letters.

Contents

Chapter 1

Introduction

In a world of modern software development the number of requested features increases and the complexity of software products rises. Besides that the opportunities of tools and libraries used by software developers have got more powerful in recent years. Fast changes are provided by third party tools and demanded from software developers in many fields of expertise. The only factor that remains constant in software development is the human being. He has to keep up and be able to compete in this environment.

This diploma thesis addresses this fact in many aspects. The challenges of changing an existing work process from scratch and developing a software tool that drives that change have to be addressed.

The reader of this work can expect detailed descriptions in the important areas of this thesis. Furthermore links and hints for additional reading are provided whenever needed. The main goals of this thesis are to improve software quality and a better working together of all persons involved in the development process.

Part I

Theoretical Background

Chapter 2

Software Test

According to the Oxford Dictionary [19, p. 332] a test is “something done to discover a person’s or thing’s qualities or abilities”.

The basic process of software testing can be described in three steps [15, p. 36]:

- Test preparation
- Test execution
- Test evaluation

In the following sections software tests are explained by their scope and by their method. Additionally test metrics are described that are used to assess the progress and the results of the executed tests.

2.1 Test Scopes

The scope of a test refers to a certain abstraction level. The number of abstraction levels is a measure of the complexity of a software product. Software tests can be set up on different abstraction levels as seen in Figure 2.1.

A program is formed by several program units which unite to a subsystem. A system consists of several subsystems.

2.1.1 Program Test

The test strategy reflects the development strategy of a program. If top-down-development is used then top-down-testing is recommended. This is mostly the case if a program is put together from previously existing modules. Top-down-development strategy means that at first only the façade of a program will exist without any functionality behind it. After creating a skeleton of the program fully functional program parts will be implemented

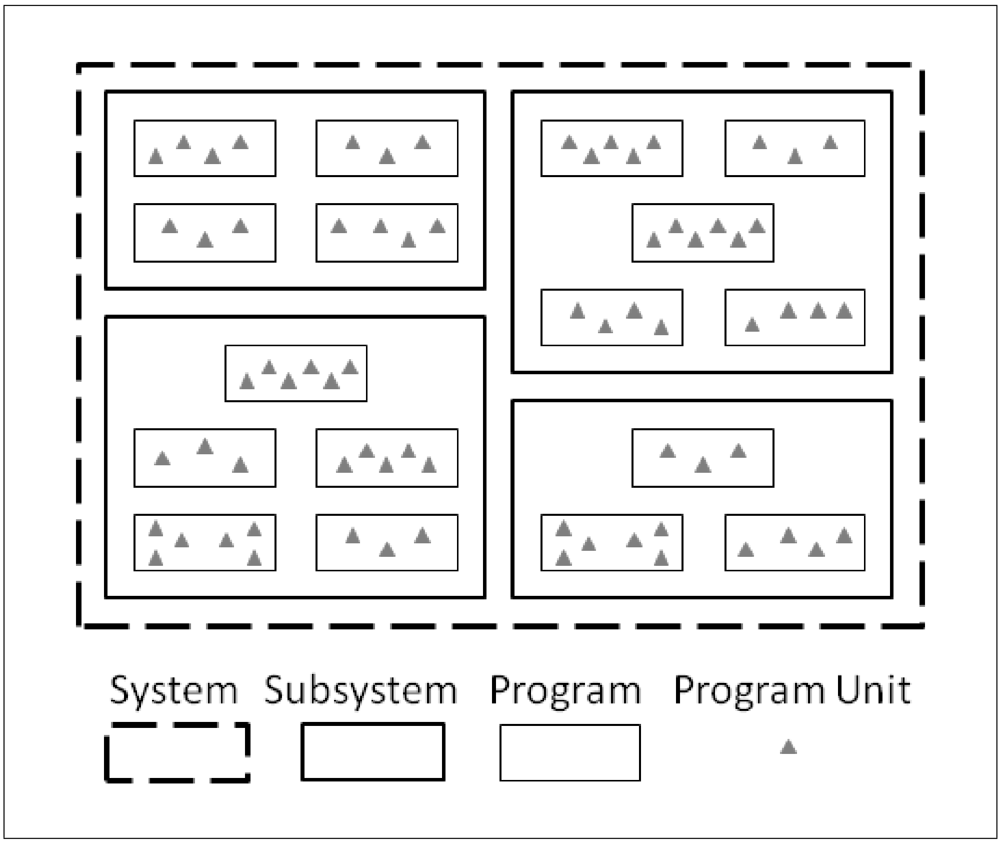


Figure 2.1: Test Scopes of a Hierarchical Software Project [15]

and tested step by step. Development and test accompany each other from the root of the call hierarchy down to the leaves of the call tree of the program.

A big advantage of that strategy is that the interfaces between software and test as well as the interfaces between program units, are clearly defined from the beginning on. This means little effort for the specification and implementation of test adapters. All that remains to be done is the creation of test data and the connection of the adapters with the tested program interfaces. The disadvantage is that there is often no functionality behind the program façade. This requires the implementation of simulated program behavior for the test to be executed.

When using bottom-up-development the implementation starts at the leaves of the call tree. These are small program units or functions. Later on bigger units are put together until the root of the call tree is reached. In this case no simulated functionality for testing purpose is needed. When development and test move up one level of abstraction all lower program parts are already implemented and tested. For the lowest program entity unit tests are recommended.

As a general rule it can be stated that for iterative development processes the top-down strategy is more promising than the bottom-up strategy [15, p. 75].

2.1.2 Program System Test

To test a system of programs the perspective of top-down or bottom-up testing is inadequate. Program systems are specified by their communication channels and protocols. There is no pure tree structure visible. It is more common to talk about program system chains or subsystem chains. Depending on the application, sensible network paths have to be chosen for the tests.

Test data can be infused into a program system using the communication protocols between separate programs. This results in well defined starting points for test adapters that send and receive test data.

The test is chasing development and its strategy. Implemented components are tested after they are available. Testing and development have to be coordinated in a predefined time schedule. This practice is called *availability strategy* because testing must stick to the availability of the tested programs. The disadvantage of that procedure is that you must be better organized and well prepared.

Often development is directed by changing priorities. These can be customer requirements or the examination of the highest risks. It means for the test that all software parts that contribute to a certain function are integrated and tested. This is called thread testing because an entire system function is distributed to several programs in a system. All program parts that represent that function are called thread.

A criterion for test prioritisation is often combined from more than one factor. For finding test priorities it is important to consider the whole context of a software product. The experience of the developers as well as the complexity of several program parts play an important role. Perceptibility of failures, severity and the probability of a failure should be taken into consideration as well [9, p. 14].

2.1.3 The Right Test Scope at the Right Time

A *module test* executes a test of a single program unit. For doing this a specialized test tool is needed that uses the program unit's input interface to send data into it. The output interface of the program unit under test is monitored to read the resulting data. After the test execution the obtained results are evaluated and assessed.

The so called *system test* or alpha test takes place when the entire software system has to be tested. At the development company a realistic simulation of the expected production environment is set up for the test execution. The next test step is beta testing which means the distribution of the software to some selected customers. The customers are asked to use the program in production like conditions and give feedback about the program's behavior.

There are two types of *integration tests*. In the first some units of one program are put together and tested. By doing this the correctness of one program can be verified. The second integration test uses more than one program unit combined in a system. Programs can also be integrated into this test one by one. With that procedure the localization of errors is simplified.

2.2 Test Methods

A test method should be chosen depending on the intended scope and abstraction level of the test to be implemented. A good guideline for many possible test targets can be found in the often cited ISO 9126 which addresses software quality criteria. Since this document is aimed to describe the introduction of a test automation tool to prove functional correctness, only the aspects of functionality of ISO 9126 are touched here. See Figure 2.2, taken from [36] and [20].

2.2.1 Functional Tests

In functional testing the overall complexity of an entire program is split into smaller and less complicated so called functions. It is a good practice to do this because it eases discussion, communication and documentation of a

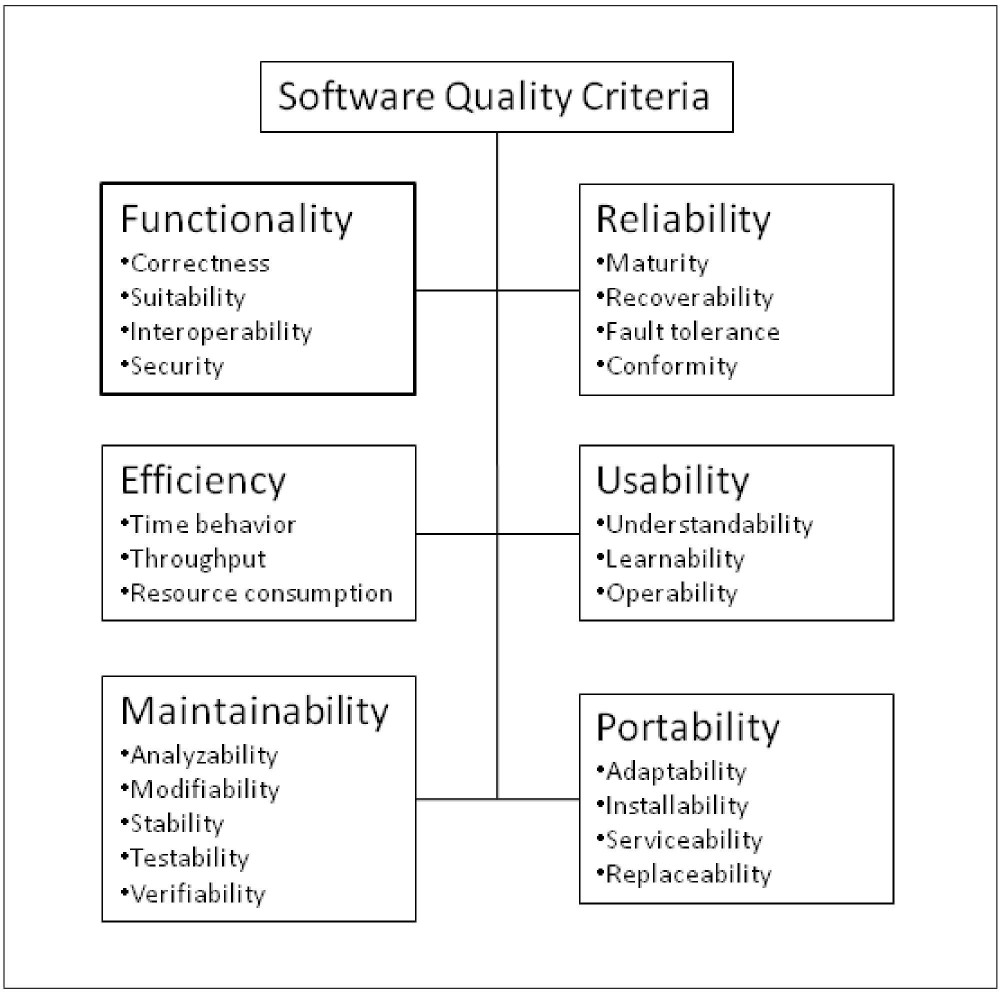


Figure 2.2: ISO 9126 Software Quality Criteria

single test case [25, p. 35].
Functional testing can prove the following quality criteria [4]:

- Correctness
- Suitability
- Interoperability
- Functional security

Functional testing ensures that a software program does what it is intended to do. What a software should do is defined on the basis of requirements and specifications, the knowledge of the tester and implicit requirements of the customer. Implicit requirements are requirements that arise out of the context of other requirements.

The size of the functional test changes with the complexity of the tested software. If unit tests are executed the focus lies upon the functionality of a single component. During integration tests interfaces are tested to ensure that the software has been put together successfully. When doing system tests the end-to-end functionality of an entire system is in the forefront.

Functional testing is also called black-box testing because it is based on the specifications of a program and not on the internals of the developed code [33]. It is a testing technique that has a very wide area of application. Thinking about functional tests already during the formulation of the specifications can enrich the quality of the specifications which will consequently ease the understanding of the software's context and size.

Test for Correctness

The criteria for test correctness can be deducted from the program requirements. In the requirements a lot of the desired behavior of the software is described. Each of the statements is a starting point for a correctness test case. Therefore it is rather easy to generate test cases if there is a well-written requirement document.

Functional tests for correctness can be very detailed (such as the provoking of a special error message in a complicated context) or only aimed at some very top-level use cases like the filling out of a registration form or a login process. Test for correctness touches many areas in software development. Calculations, graphical user interface representation, data accessibility and data correctness are only some of them. The test cases are written quite easily and there can be a lot of them [4].

If there is only an imprecise or even no requirement document the expertise of the tester is the basis for generating useful correctness test cases. In this case the tester has a much bigger responsibility for the success of the project since the project targets are not defined that clearly in the requirements document. Therefore the tester has to investigate suitable situations, usages of the program in production and he has to think of adequate test cases all by himself. After asking experienced developers, analysts and users of similar programs the tester needs to have common sense and creativity as well to generate well-directed test cases.

Test for Suitability

To execute tests for suitability the planned or expected application of a software product must be known. As the name indicates, this method tests if the set of functions are suitable for the specified tasks. Normally suitability tests are based on use cases and usage scenarios to judge if the software complies with the requirements of the users [4].

The plausibility of the use cases is of great importance for the effectiveness of such tests. The suitability can be proven if the use cases mirror the real interaction of the users with the system. No conclusion about suitability can be given if the use cases do not represent the real user behavior. Given the situation that there are no use cases available the tester has to gather knowledge about the expected utilization of the software and generate use cases.

It is not sufficient to do automated testing. There are concepts like keyword driven testing that support test automation and the suitability of tests. Still it is necessary to collect detailed information about the purpose of the tested system to implement effective and well-directed tests [36].

The in-depth knowledge of the tester is crucial for suitability tests. In most cases the requirements are not sufficient to define suitability test scenarios. It happens very often that shipped programs do not fulfill the expectations of their users. Programs consume too much hard-disk space, are too slow and too complicated to use for the purpose of the user. This is why it is a good idea to include some designated future users into the test process.

Test for Interoperability

Interoperability means the program's ability to communicate. With an interoperability test it is shown that the software to be tested works as expected in several environments. These environments comprise of hardware, software, middle ware, operating systems, and other connected applications or network architectures. Portability and compatibility are other terms that describe interoperability. The degree of interoperability of a software is defined by the number of necessary configuration parameters that have to be adapted when changing the context of a software's runtime environment. Interoperability is better if the number of configuration parameters is lower.

A further feature of a software regarding interoperability is the ability of a software to recognize its system requirements automatically. Requirements are system parameters, environment variables or platform singularities. The higher the manual configuration effort is to get a software running the lower is its interoperability. The keyword plug-and-play is a synonym for very high interoperability [4].

Interoperability tests are mandatory for a software that shares an environment with other software. Commercial off-the-shelf (COTS) software has to undergo severe compatibility tests. These tests also contain tests about data transfer between different environments and circumstances.

Test for Functional Security

Security tests are divided into functional and technical security tests. The purpose of this test is to assure that access to certain areas is granted for authorized users only and that access is denied for unauthorized users. If it is possible to break into a system without authorization a technical security problem exists. If an authorized user detects that he has the rights to perform steps that a user with his role should not be able to perform a functional security problem exists.

A security concept affects the whole software system. This is why performing security tests requires a detailed knowledge of the software being tested. Only by doing that, can all possible access points to a software can be identified and tested. Functional security tests are limited to all known legal access points because all other access points are tested by technical security tests.

One aspect is to test if an authorized user can use all functions and access all data that a user with his role should be able to access. The other aspect is to test that all of the content not allowed is inaccessible for that user. To create such test scenarios the tester must understand the security concept and the security standards of the software. That means that user accounts with different privileges representing the different user roles must be created for the test. To ensure a suitable coverage of security tests they have to be considered in the test concept [4].

2.2.2 Combinatorial Testing

Software requirements are written in natural language. This is the most flexible and understandable way to communicate information. On the other hand it is hard to analyze natural language automatically and derive test input from it. This is why the first step for combinatorial testing is manually going through the specification, and structuring it into a set of properties and attributes. This leads to the main advantage of doing combinatorial testing, namely the separation of the analysis of the problem (which is done manually) and the synthesis of test cases (usually automatically) [33].

Combinatorial testing is well suited for tool support. The main three techniques of combinatorial testing are described in the following sections.

Category-Partition Testing

The category-partition method tries to identify input spaces by separating possible values into complete test cases. This gives a fast estimate about the number of test cases. Three steps have to be performed for category-partition testing [33]:

1. Independently testable features have to be found. *Categories* are introduced which contain the elementary *parameter characteristics*. The goal is that the generated test cases can later be combined in any order and do not influence each other. This means that one test case must not be the precondition of the next one.

2. The next step is to select values isolated from other parameter characteristics. The distribution of this classification of values is called partitioning the categories into *choices*. This step gives the method its name.

3. In the last step some limitations have to be introduced to generate useful test cases. The values are restricted by forbidden value ranges and also useless combinations of values are eliminated.

The obtained categories, choices and constraints are then put into a test case generation tool to automatically generate realistic test case sequences.

Pairwise Combination Testing

Pairwise combination testing tries to find pairs of values and combines them into test cases. This is a stricter form of the category-based testing described in Section 2.2.2. The number of test cases is relatively small.

A tester who has performed the category-based method will soon be stunned by the large number of test cases that it produces. The number is calculated as the product of the number of classes for each parameter and therefore grows exponentially because each allowed combination is generated into a test case. Pairwise combination on the other hand tries to find pairs and triplets of combinations which reduces the number of test cases dramatically. The test case count does not grow exponentially but logarithmically which saves a lot of effort and increases efficiency.

Since it is very exhausting to perform the generation of all reasonable combinations manually, automation is used. To simplify the automatic generation again limitations are imposed to avoid pointless combinations. These combinations in pairs are avoided by the so-called omit instruction [33].

Catalogue-Based Testing

A catalog is a group of attributes that contains values of one type. By sorting demanded features and grouping them it eases the step of specification analysis. To create catalogs, needs experience which comes with a longer examination of a particular area. Using that experience a test engineer can define types of a variable that embody logical inputs, outputs and status of the computation. If an integer variable is used, a catalog would define the following cases [33, p. 194]:

1. The element immediately preceding the lower bound of the interval
2. The lower bound of the interval
3. A non-boundary element within the interval
4. The upper bound of the interval
5. The element immediately following the upper bound

It is easy to see that in this catalog the cases 1 and 5 identify erroneous conditions, 2 and 4 the boundary conditions and 3 the normal conditions.

Elementary items of the specification are identified into sets in step one. These basic sets are preconditions, postconditions, variables, operations and definitions. This excerpt will be rather informal at first as it is written in informal language.

Secondly a set of test case specifications is generated from preconditions, postconditions and definitions. So called *validated preconditions* are defined that consist of simple boolean expressions. *Assumed preconditions* should be avoided. They are only applicable if a condition is a logical disjunction of more elementary conditions. Postconditions are treated like validated preconditions i.e. with possible values true (if the condition is met) or false (if the condition does not apply). Definitions set the values for the variables. They are again treated like validated preconditions with values that meet the specification and values that don't.

In the third and last step the values defined in the catalog are transferred into the specification. All values from the catalog are connected to cases in the specification. For each catalog entry, a list of types of elements that can occur in a specification is generated. For example, if we have an entry for boolean variables two test cases are necessary - one resulting in true an one resulting in false. A good trick is to use wrong values for testing the input behavior of a system and correct values to test its output behavior.

2.2.3 Structural Testing

Compared to functional testing discussed in Section 2.2.1 structural testing is not aimed at the program specifications but at the program itself.

Function testing checks if a function is fulfilled, structural testing checks that as many control flow elements as possible are tested. Control flow elements are statements, paths, branches and conditions. Program parts not executed can result from the inherent difference between specification and implementation of a program. A big influence on that gap has to do with the experience and understanding of the developer, as well as the chosen programming language and the program environment. An aspect that arises with structural testing is coverage. Coverage means the part of a program that has been executed by the test, expressed as a percentage value. With coverage we have a measure of test accuracy and also a measure about test progress.

At first glance structural testing is closely connected to the source code for doing module or unit tests. On the other hand there are some reasons why it is furthermore good to look at structural testing as detached from the code [4]:

- The tester thinks that the tests are only useful for the developers and not for themselves.
- We exclude opportunities that arise if we perform structural testing in other areas.

The most elementary structural testing technique is *statement testing*. Statements are represented by the nodes in the control flow graph. The technique demands that each statement is executed at least once by the test. The result is the statement coverage $C_{Statement}$ [33, p. 215].

$$C_{Statement} = \frac{\text{number of executed statements}}{\text{number of statements}} \tag{2.1}$$

It is obvious that if $C_{Statement} = 1$ then the coverage has reached its maximum value.

To demonstrate the three structural testing methods I will include an example which is based on Armin Beer's lecture notes [10].

```java
public void coverMe(int a, int b) {
  System.out.println("A");
  if (a < 1) {
    System.out.println("B");
  }
  System.out.println("C");
  if (b < 2) {
    System.out.println("D");
  }
  System.out.println("E");
}
```

Listing 2.1: A simple Java Program

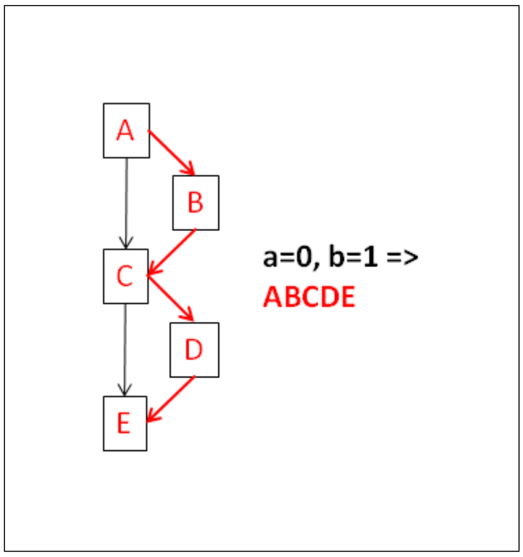


Figure 2.3: Statement Coverage

To fulfill maximum statement coverage we need exactly one test case as shown in Figure 2.3.

As we can see in the example control flow graph in Figure 2.3 it is possible that the same statement can be called by different branches. Therefore just the testing of all statements is insufficient. There are not all branches covered in the tests that can cause the execution of one statement. *Branch testing* goes one step further than statement testing. Branch testing requires all possible paths of a program being executed by at least one test case. A branch stands for an edge in the control flow graph. If we test for branch coverage we ensure that each statement is called by each possible preceding statement. In other words this means that all edges that lead to one node should be executed by the test.

The branch coverage C_{Branch} is defined as follows [33, p. 217]:

$$C_{Branch} = \frac{\text{number of executed branches}}{\text{number of branches}} \tag{2.2}$$

Figure 2.4 shows the example from above with test cases that cover all branches.

A path is one possible way through the control flow graph of a program from start to end. Therefore *path testing* is the technique that demands the generation of test cases that execute each individual possible path through a program. C_{Path} is defined as [33, p. 222]:

$$C_{Path} = \frac{\text{number of executed paths}}{\text{number of paths}} \tag{2.3}$$

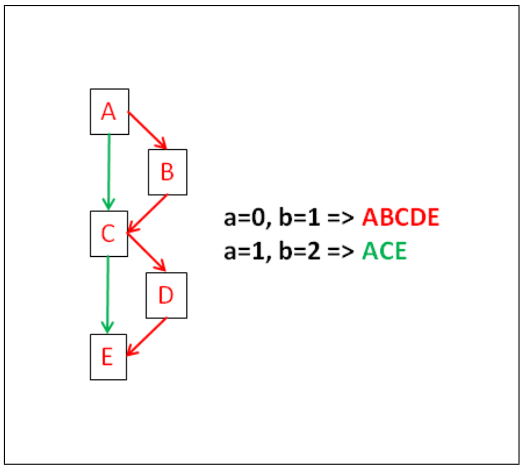


Figure 2.4: Branch Coverage

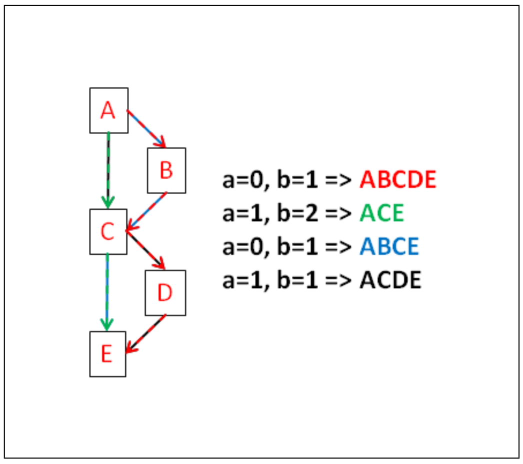


Figure 2.5: Path Coverage

Figure 2.5 shows our example with test cases that cover all possible paths.

In theory path testing sounds tempting but in praxis it is almost never applicable. This is because if a program contains for- or while-loops, the number of paths and therefore the number of test cases to cover them is limitless. So path testing can only be applied to a program if it has no loops in it. One possibility to avoid this knock-out criterion of path testing is to divide the program's execution path into sub-paths.

2.2.4 Data Flow Testing

Data flow testing means to test how one syntactic element affects the computation of another. On contrary to control flow testing discussed in Section

2.2.3 data flow testing does not focus on all possible execution paths of a program but on the data used in the program. If a wrongly computed variable is not used later on, this does not lead to a program error. Control flow testing ignores that aspect and produces a lot of test failures although the computed data may never be used.

DU Pairs and DU Paths

A program variable has two important facets: Its definition, meaning the assignment of a value and its uses. Therefore it is important to track these two aspects of data flow information. For example if we look at the statement ++var we can see that it contains a usage of the variable var and then a definition. ++var is the same as writing var = var + 1. At first var + 1 is computed (usage) and then it is assigned to var again (definition) [33].

For a detailed data flow description it is suitable to use definition-use pairs (*DU pairs*). These are written as variable name plus <definition line number, usage line number>. To obtain a DU pair there must be at least one path without definitions on its way from definition to usage (the value of the variable must not be changed from definition to usage). Such a path is called definition-use path (*DU path*). Since a definition is a write operation and a usage is a read operation on the variable it is possible to have multiple uses on a DU path because uses do not change the value of the variable.

To obtain information about DU pairs and DU paths it is necessary to perform a static code analysis. Definitions and uses that only exist during the runtime of the program are not taken into consideration.

DU Test Definitions

Another perspective is that an error that exists in a computation and is assigned to a variable first shows in the usage of the variable. From this it follows, that definition errors are not revealed until the use of a variable is evaluated in a test. The all DU pairs coverage $C_{DUpairs}$ is defined as [33, p. 239]:

$$C_{DUpairs} = \frac{\text{number of exercised DU pairs}}{\text{number of DU pairs}} \tag{2.4}$$

To extend the test accuracy it is important to know that one DU pair can be executed by several different DU paths. Therefore a metric has to be introduced as well. $C_{DUpaths}$ is defined [33, p. 240]:

$$C_{DUpaths} = \frac{\text{number of exercised simple DU paths}}{\text{number of simple DU paths}} \tag{2.5}$$

Chapter 3

Metrics as a Measure for Software Quality

Software development is essentially chaotic and so is the derived science of testing. Metrics are needed to make processes transparent, controllable and measurable. With their help it is possible to simplify the planning of software development and testing.

3.1 Step Size of Metrics

To assess a development process and to derive measurement values from it the level of the metric should be defined. For example, to measure the quality of software tests one could use a five step scale that displays the test effectiveness or event percentage values to define the test coverage of an entire program. Sometimes a concrete graduation is very evident, sometimes different other systems could be used.

3.1.1 Nominal Scale

The simplest possibility of division is the nominal classification. We need predefined categories that fulfill two properties [24]:

- At first these categories should be mutually exclusive which means that one measured value can only be exactly in one category and not in any other.
- Secondly the set of all possible categories must contain all appearing possible values. If we encounter a value that does not have a matching category we have to create a new category.

Example:
Sitting at the curb of a street we would like to classify all passing persons

according to their nationality. We define 'German' and 'Italian' as possible categories. These are exactly definable and mutually exclusive (Property one is fulfilled).
Germans and Italians walk by at the test point and can be sorted exactly into 'Germans' and 'Italians'. An Austrian passes. That means that property two is not met. The possible categories have to be extended with 'Austrian' and the evaluation can be continued.

For the nominal scale order and ranking of the categories is arbitrary. There is neither information about importance nor their relation among each other.

3.1.2 Ordinal Scale

In addition to the properties of the nominal scale we demand the categories to have an order. By doing this we obtain comparability. Cars for example can be subdivided by their size (micro-car, sub-compact, compact car, medium-sized car, luxury class). Logical asymmetry (if A $>$ B then also B $<$ A) and transitivity (if A $>$ B and B $>$ C then A $>$ C) are valid. The step size from one category to the next is not defined exactly though. This is why one should only apply comparison operators to the measuring value and no mathematical operations such as addition or subtraction. These could lead to wrong interpretations.

3.1.3 Interval and Ratio Scales

The interval scale specifies a unit which defines exactly the distance between measuring points. This scale unit should be defined as common standard which is traceable and reproducible. Making these prerequisites it is possible to apply mathematical operations to measuring values, and the data becomes evaluable and interpretable.

If there is an absolute and not random zero-point on an interval scale it is called ratio scale. This is the most precise arrangement, and as a result all mathematical operations are applicable, even division and multiplication.

To give an example we have a look at the temperature scales of Fahrenheit and Celsius. Both have different zero points which are defined at random and are not related to the point of absolute zero. Therefore the scales of Fahrenheit and Celsius are interval scales. That is a problem and because of that the scale of Kelvin was introduced. It is related to the point of absolute zero temperature and therefore a ratio scale.

Summing up, it can be said that the system of scales is hierarchical. Each higher scale includes the same properties as the one below and some additional properties. The higher the type of scale the more powerful and significant are its interpretation possibilities of measuring values. A higher scale type can always be simplified to a lower one (for example for a better understanding and transparency) but not vice versa.

3.2 Software Quality Metrics

Software quality metrics can be divided into three categories: product metrics, project metrics and in-process quality metrics. Product metrics describe the product itself in terms of design features, size, complexity, performance and quality level.

Project metrics accompany the implementation of a project and deal with the number of developers, cost, time schedules and productivity during the project runtime.

In this chapter in-process metrics are described. They are used to document and improve development as well as maintenance. Among them are the defect density during automated tests, the defect arrival pattern during automated tests and the defect removal effectiveness (DRE).

In-process metrics start their work in the area where a lot of errors happen: in the software development process. These metrics are not defined precisely, and their usage varies with the structure and size of the development project.

3.2.1 Defect Density during Automated Tests

When doing automated test runs, it is important to produce as many errors as possible. The defect density during testing can be interpreted in several ways though. Finding many errors and fixing them does not automatically result in a good product. It can also indicate too intensive and too precise or wrong test cases.

A good starting point is to observe and document the defect density per kilo lines of code (KLOC) of a product over more than one release cycles in one and the same development department. In doing so we can nearly exclude all external factors that would distort the result. Two things are important:

- If the defect density stagnates over a longer time period or if it even declines compared to a former release we should pose the question: Have the test cases deteriorated for the current release?
  - If no, then the product quality increases.
  - If yes, then additional and better tests have to be created.
- I the defect density rises significantly compared to the last release then we should ask: Did the test effectiveness improve?

- If the answer is no, then the product quality has become worse. The only way out is to test more and more intense. This will initially boost the defect density as well but on the long term help to improve the quality of the product.
- If yes, then the product quality has remained the same or has become even better.

3.2.2 Defect Arrival Patterns during Automated Tests

A defect density is just an average value for one entire test-run. More precise analysis can be generated if we try to find similarities between frequent errors. By grouping these errors it is possible to find defect patterns. An example is the duration between defects. In the end we would like to achieve a stable error ratio on a low level or to keep the times between erroneous behavior called mean time to failure (MTTF) as long as possible.

As a time unit for the investigation of defect arrival patterns normally a week is taken, sometimes also a month. These three slightly different aspects should be taken into consideration simultaneously:

- Reported errors during the test phase per time unit: These are all errors, which consist of real errors and non real errors.[1]
- The pattern in the arrival of real errors: these are the important ones.
- Extra work because of backlogs: It is of no use to document all errors, if there are insufficient resources in the development team to fix them.

3.2.3 Defect Removal Effectiveness

The defect removal effectiveness (DRE) is defined as [24, p. 103]:

$$DRE = \frac{\text{Defects removed during a development phase}}{\text{defects latent in the product}} \times 100\% \tag{3.1}$$

Since we do not know the total latent defects in the product this value is approximated as:

$$\text{defects removed during the phase} + \text{defects found later} \tag{3.2}$$

This value can be computed at different times. The early defect removal effectiveness is calculated before the code integration and the phase defect removal effectiveness is calculated for one particular project phase. The higher the value is the better is the development process because less errors are carried forward from one project phase to the next.

[1]A real error is a program related error. On contrary a non real error is a program error that has emerged because of the wrong usage of a program, a test tool or a wrong test case.

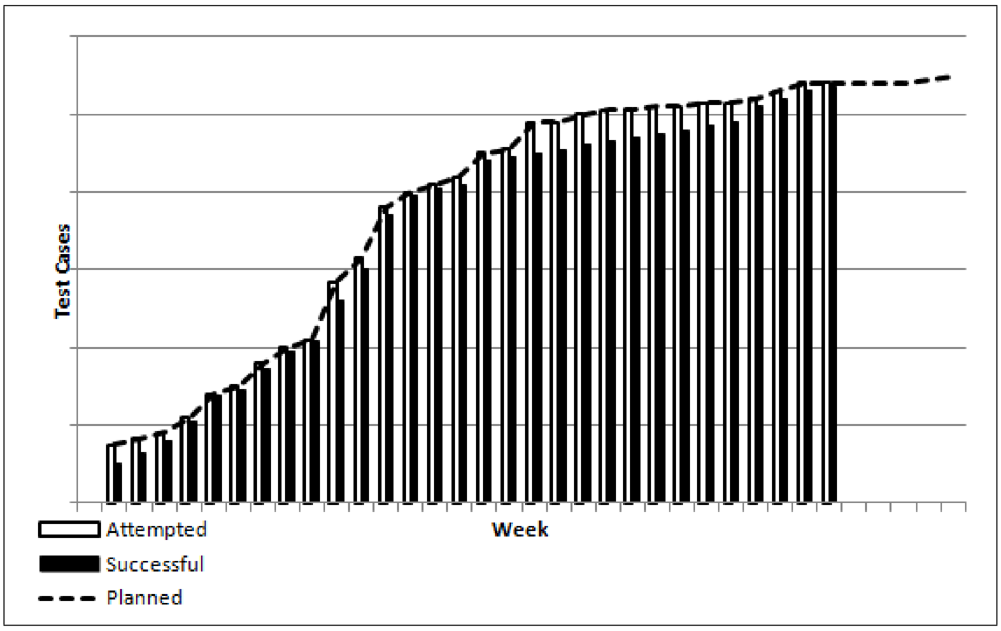


Figure 3.1: Test Progress Curve, modified from [24]

3.3 Process Metrics for Software Tests

In this section the most important process metrics for the management of software tests are described.

3.3.1 Test Progress Curve

The number of test cases is displayed in a graph over a time axis. The shape of the resulting chart often has the outline of an S - that is why the test progress curve often is called *S-curve*. To make this metric sensible we need this information in it:

- Planned test progress (number of planned successful test cases)
- Number of executed test cases per week
- Number of successful executed test cases per week

The goal of this metric is to compare the progress of the real test to the progress of the planned test. By doing this it is possible to recognize significant deviations and to react on them. Interpretations about the development progress related to the development schedule can be made as well. Figure 3.1 shows an example of a test progress curve.

There is also the option to combine more than one test progress curve of different releases into one chart and compare them as shown in Figure 3.2.

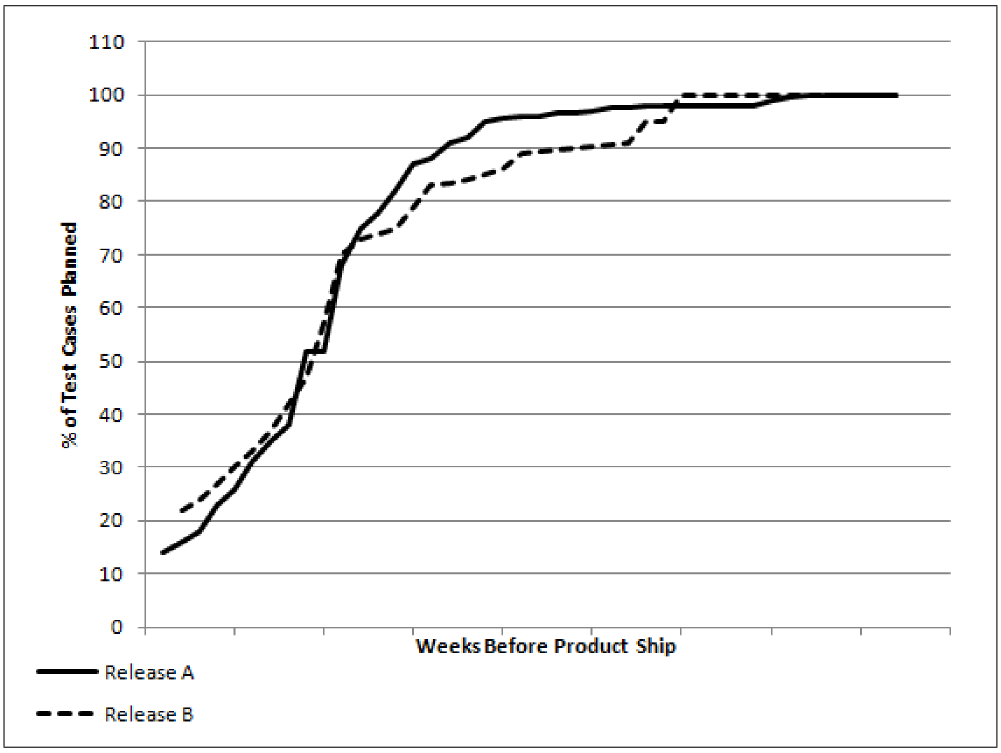


Figure 3.2: Comparison of Test Progress Curves of different Releases, modified from [24]

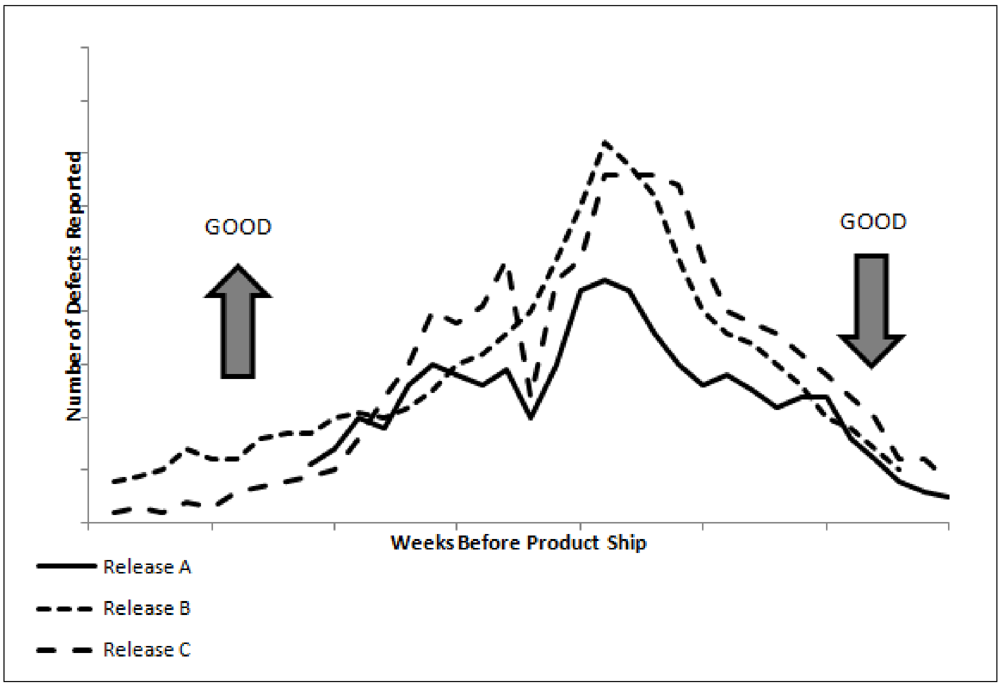


Figure 3.3: Number of reported Defects over Time, modified from [24]

3.3.2 Time Elapsed of found Errors

It is recommended to collect found errors per time unit and per development phase. These instructions should be considered:

- Data with a comparable time basis has to be used, for example data of some former releases.
- The unit of the X-axis is weeks before product ship.
- The unit of the Y-axis is the number of found errors. Figure 3.3 shows an example of how to plot defects over time for different releases.

3.3.3 Testing Defect Backlog over Time

The testing defect backlog is represented by the cumulated number of the differences between reported errors minus fixed errors. If this list is quite long before the product shipping there is the possibility that errors that are already on the list are reported again by customers. This is why the backlog should be kept as short as possible to avoid duplicate error entries. To achieve a high product quality as fast as possible errors with the highest priority have to be fixed first. This approach is recommended:

- If there is a test plan then the test progress should enforce an early rise in the S-curve.
- One should keep an eye on error messages and the causing problems should be analyzed. This should not be done automatically because an experienced tester can derive important knowledge about the overall state of the system out of its error messages.
- The defect backlog should be monitored strictly.

3.3.4 Product Size over Time

Lines of code (see Section 3.4.1) or other product size metrics are well-suited indicators for the development team. Product size rises during each development phase and increases with the number of implemented features. Well-directed refactoring can bring down product size from time to time. Towards the end of a project sometimes features with lowered priority are removed from the product. Some product size metrics are lines of code, function points or even the memory usage of a program.

3.3.5 CPU Usage Trend over Time

For systems with the demand for high availability and high stability the CPU usage is an important indicator. CPU usage represents the load of the used hardware. Stress tests[2] should start during the component test phase and should be performed up to the system test phase. Prior to this it is important to define the targets of the planned CPU usage. Doing stress tests significant limitations of a product can be found out like the maximum number of concurrent users.

3.3.6 System Crashes and System Hangs

This metric is connected tightly to the above mentioned CPU usage in Section 3.3.5. It is represented by the number of unplanned initial program loads (IPLs). To detect problems of that kind the system has to be put under high utilization to generate a high CPU load. If defects of that kind are found and fixed the system's stability improves over time. If an unplanned IPL is found, a classification and codification is recommended [24, p. 290]:

- 001 Hardware problem (unplanned)
- 002 Software problem (unplanned)
- 003 Other problem (unplanned)
- 004 Load fix (planned)

[2]A stress test is a technique to detect the hardware limits and the program's efficiency

3.3.7 Mean Time to Unplanned Initial Program Load (IPL)

It is useful to test for mean time to unplanned initial program load (MTI) not before the system test phase. At that stage the program is finally completed sufficiently [24, p. 291].

$$\text{Weekly MTI} = \sum_{i=1}^{n} W_i * \left(\frac{H_i}{I_i + 1} \right) \tag{3.3}$$

where
n = Number of weeks that testing has been performed
H = Total of weekly CPU run hours
W = Weighting factor
I = Number of weekly (unique) unplanned IPLs (due to software failures)

3.3.8 Showstopper

Severe errors that render a program unusable are called show-stoppers. According to IBM researchers all defects must be put on the show-stopper-list that impede the development or the testing process or even prevent a customer from working with a product [3]. The usage of that metric normally starts during the component test phase. All entries on the show-stopper-list have top priority and must be solved before a release.

3.4 Code Complexity Metrics

All metrics that have been discussed so far treat source code as a black box. The exact connections between design, implementation and quality have not been touched. Complexity metrics are closely connected to source code and therefore provide the possibility for the developers to improve the quality of their work directly.

3.4.1 Lines of Code

The name of this metric misleads to the perspective that it only measures the number of written source code lines. Indeed it is a bit more complicated because there are a number of definitions for lines of code. There are more than one for the same programming language and again different ones for different programming languages. These are the possibilities of counting LOC [22]:

- Only executable lines
- Executable lines plus data definitions
- Executable lines, data definitions and comments

Table 3.1: Curvilinear Relationship between Defect Rate and Module Size

Maximum Source Lines of Modules	Average Defect per KLOC
63	1,5
100	1,4
158	0,9
251	0,5
398	1,1
630	1,9
1000	1,3
>1000	1,4

- Executable lines, data definitions, comments and command language
- Lines as physical lines on the input screen (depending on screen size)
- Lines that are terminated by a logical separator like *;*

Due to the many different definitions of lines of code this metric can be hard to use. To obtain usable and comparable data there has to be a standard of how to count the lines.

If one wants to compare lines of code of two different products that are implemented in different programming languages it is recommended to normalize the lines of code at first. The reference language for that is Assembler. Conversion factors to Assembler for lines of code exist for nearly all common programming languages. By doing this the counted values are almost comparable.

Example:
For the release of a software product it is rather simple following the definition of lines of code to declare its quality level: 'The product has 50 KLOC. The latent defect rate of that product is 2.0 defects per KLOC within the next four years.'

One other perspective is trying to find a link between lines of code and defect density. The logical conclusion is that there must be a higher defect rate for more lines of code because the complexity is higher.

Early studies at the beginning of the eighties have found out the opposite. The more lines of code a product had the lower was the defect density in it [2]. Later investigations concluded that the characteristics is curved. Very small program modules have a higher defect density and so do very big ones. In average sized programs the error density is lower [51]. The relation between lines of code and program module size can be seen in Table 3.1 [24, p. 313].

3.4.2 Halstead's Software Science

Maurice Halstead coined the term *Token* which represents a unit with which a developer can combine programs of bigger size. The primitive measures of Halstead are [18]:

n_1....Number of different operators in a program
n_2....Number of different operands in a program
N_1....Number of usages of operators
N_2....Number of usages of operands

Based on that primitive units Halstead developed a number of equations. Vocabulary (n)

$$n = n_1 + n_2 \tag{3.4}$$

Length (N)

$$N = N_1 + N_2 = n_1 * log_2(n_1) + n_2 * log_2(n_2) \tag{3.5}$$

Volume (V)

$$V = N * log_2(n) = N * log_2(n_1 + n_2) \tag{3.6}$$

Level (L)

$$L = \frac{V*}{V} = \left(\frac{2}{n_1}\right) * \left(\frac{n_2}{N_2}\right) \tag{3.7}$$

Difficulty (D) = inverse of level

$$D = \frac{V}{V*} \tag{3.8}$$

Effort (E)

$$E = \frac{V}{L} \tag{3.9}$$

Faults (B)

$$B = \frac{V}{S*} \tag{3.10}$$

V* means the minimum volume of an included function to fulfill the purpose of the entire program and S* is the average number of mental decisions between errors (S* is 3000 according to Halstead [18]).

3.4.3 Cyclomatic Complexity

Cyclomatic complexity was introduced by Thomas McCabe to measure the testability and the understandability of a program [30]. The execution paths of a program are drawn into a graph. The formula is:

$$M = V(G) = e - n + 2p \tag{3.11}$$

$V(G)$....Cyclomatic number of G
e....Number of edges
n....Number of nodes
p....Number of unconnected parts of the graph

This metric displays the number of binary decision points. Experience has shown that a cyclomatic complexity of 10 should not be exceeded. However if the number is exceeded one should think of refactoring the affected program part or even arrange a re-design.

3.4.4 Syntactic Constructs

This is an extension to the cyclomatic complexity and watches closely which loop construct has the highest defect probability. In the research work of Lo came to light that *while* loops caused the most problems for programmers [27]. The number of while loops was reduced in a program and subsequently the defect rate decreased.

3.5 Metrics in Object-Oriented Projects

Nowadays a lot of software projects are realized with object-oriented (OO) programming languages. Therefore it is necessary to introduce specialized metrics that aim for OO projects. The basic understanding of object orientation will not be discussed in that section. All other metrics can be applied to OO projects plus the following specialized ones. Rules of thumb are listed in Table 3.2.

3.6 Metrics for Extreme Programming Projects

Related to extreme programming there are three preeminent metrics that help with the planning and controlling of a project. Until a product can be declared finished there are some phases called *release*. A release typically lasts one up to two months. Each release consists of more than one *iteration* which can last one to two weeks.

The detailed planning is only done for the current iteration at its beginning. At a planning meeting so called *stories* are written that represent one feature or one task. These are relatively small work assignments. Development takes place in form of *pair programming*. If a story content is defined exactly enough, all of the three values discussed in the following sections are written down on to the story sheet [6].

Table 3.2: Object-Oriented Metrics and Rules of Thumb according to Mark Lorenz, modified from [28]

	Metric	**Rules of Thumb and Comments**
1.	Average Method Size (LOC)	Depends on language, <24 LOC for C++
2.	Average Number of Methods per Class	<20
3.	Average Number of Instance Variables per Class	<6
4.	Class Hierarchy Nesting Level (Depth of Inheritance Tree, DIT)	<6
5.	Number of Subsystem/Subsystem Relationships	Should be less than the number in metric 6
6.	Number of Class/Class Relationships in each Subsystem	Should be relatively high.
7.	Instance Variable Usage	If groups of methods use separated sets of variables, the class should be split in subclasses
8.	Average Number of Comment Lines (per Method)	>1
9.	Number of Problem Reports per Class	Should be as low as possible.
10.	Number of Times Class Is Reused	If a class is not reused it might be redesigned
11.	Number of Classes and Methods Thrown Away	Should be relatively high in following the idea of an evolutionary design process.

3.6.1 Priority

For each story a priority is defined at the planning meeting together with the customer. The scale of a priority is arbitrary but should remain the same standard for one project to ensure comparability among the different releases. As an example we could define a priority scale from 1 to 10. 1 means highest priority of a story. A priority of one does not only mean exceptional importance but also that this story has to be developed before others with lower priority.

Extreme programming demands that priority determines also the implementation sequence. By doing that we ensure that the most important program parts are completed in the beginning and that the development progress is not impeded by topics of low importance.

3.6.2 Risk

At the end of a planning meeting the written stories are distributed to development pairs. The pair that works on a story assigns a risk value following their assessment to that story before they begin their implementation work. The risk value embodies the probability that the story can be implemented at all. The scale for risk - like the priority scale - can be chosen at project start. For example if we use a scale from 1 to 10, 10 would mean the highest risk.

3.6.3 Effort

Again at the beginning of the work for a story the development pair has to give an estimation of effort. The unit for effort can be hours, but we can also use a virtual unit as long as all team members stick to the same agreement. At the end of an iteration the sum of the efforts of all finished stories in that iteration is calculated. This sum represents the project speed for that iteration.

During a project the different project speeds can be compared for each iteration in each release. A past project speed value can then be used to estimate the amount of stories that can be implemented for future iterations.

Chapter 4

Test Automation

Test automation is the execution of otherwise manual test activities by a machine. It can be applied to all operations that assess software quality during different stages of the development process. Test automation is limited by the fact that it can only perform the manual operations of a tester, and not the creative and intelligent dimension of that role.

There is a number of standards that describe processes connected to software development or even software quality (e.g. ISO 9126). Test automation is a fairly young area compared to other established engineering sciences. This is a reason for the lack of generally used standards.

It is not true that software producing companies do not want to implement all the same standards for test automation. On the one hand they choose to use certain development and quality standards company-wide but it is also a tolerated practice to ignore standards for some projects because they are different. The "Not Invented Here" - syndrome (NIH) is and remains a steady companion in many software development projects [26]. In test automation the technical concepts do not fulfill general norms as they are often subject to commercial manufacturers or open-source-communities.

According to Mauro Pezzè and Michal Young [33] test automation should be seen as a support for testers to get rid of time consuming simple and often repeated activities. The aim is to make the best use of human resources. Introducing automation the testers gain time for more creative test activities. A side effect is also that people are more motivated to work, if work is not a repetition of always the same steps.

People tend to make errors and therefore are not well suited to perform regression tests because the results of such a test may differ [20]. Besides that human resources are limited and expensive. Often business needs to run tests faster and more frequently, for example in the case of an emergency hot-fix.

Depending on the type of testing activity the matching document should be edited. IEEE 829 gives an overview which test documents exist and how they interact [13, p. 189] as seen in Figure 4.1.

4.1 Use Case Definition

One way to generate data that is fed into automated tests is to define use cases. The first step is to translate informal use cases from the software's requirements. There are two ways to write use cases [12]:

- *Casual* use cases are written in normal language following no numbering criterion and no predefined structure.
- *Fully dressed* use cases follow a fixed procedure. There exists a template that contains the actors (stakeholders), the context and the executed steps.

As the number and complexity of use cases grow it is a good idea to think about a way to keep track of them. Use cases should be linked together. A possibility is to split a very large use case into sub use cases and create one master use cases that refers to them.

An important point is the style of a use case. Depending on the nature of the project a use case style has to be selected. Details on each style can be found in Alistair Cockburn's book on use cases [12, p. 132]. The five project situations are:

1. Supporting the generation of requirements.
2. Modeling the business process.
3. Estimating system requirements.
4. Writing functional requirements on a short, high-pressure project.
5. Generating detailed functional requirements at the beginning of a larger project.

One really valuable aspect of use case testing is the separation between *actor* and *action* [13]. An actor is a user and embodies the user's privileges and its role. This is the perspective seen from a real person that uses the software and not from a technical perspective. An action is a step performed by the user when working with the software. It is possible to derive test cases directly from use cases.

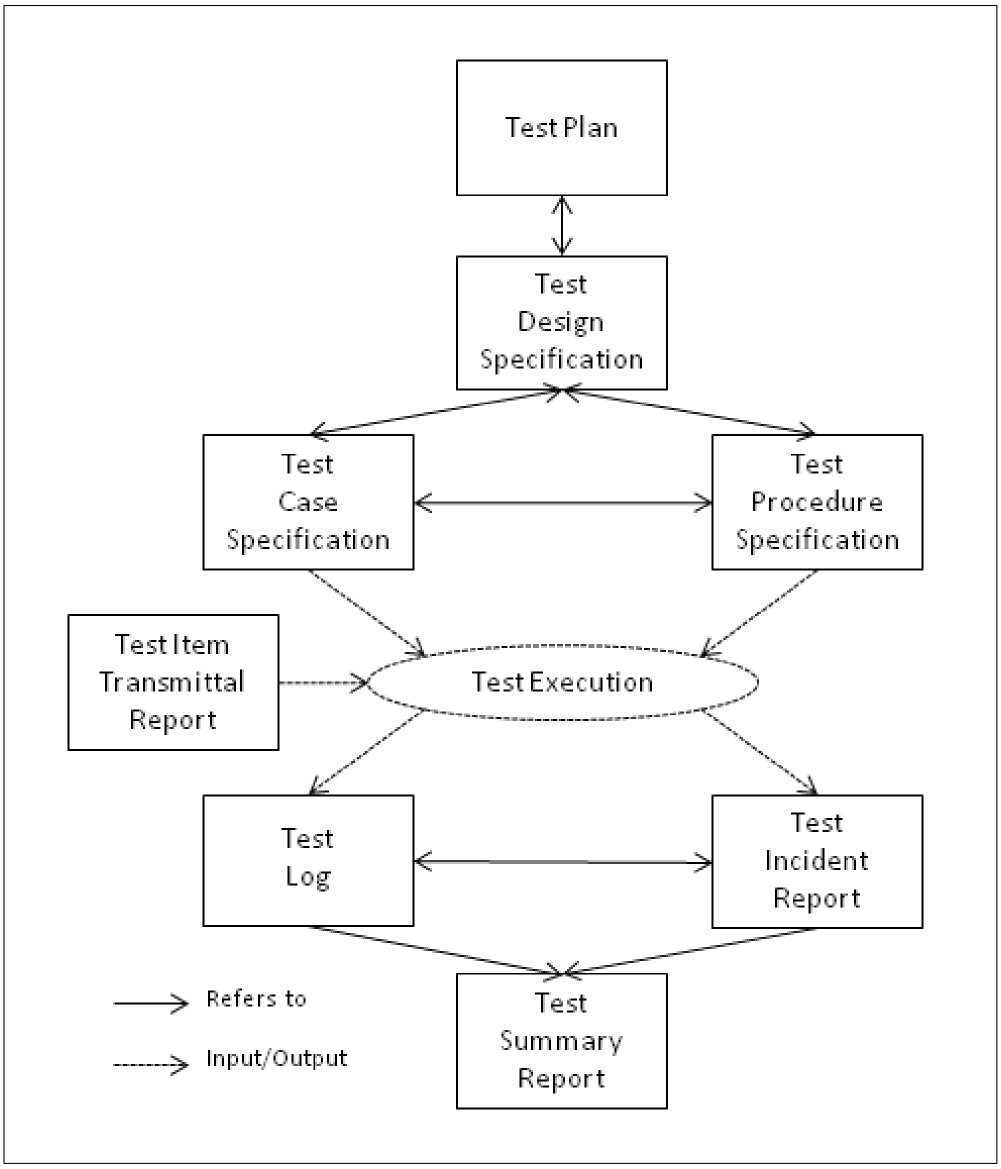


Figure 4.1: The IEEE 829 Test Documents

4.2 Test Data Description

When test data should be generated automatically one possibility is to use a formal specification language. The extended Backus-Naur Form (eBNF) is a good method to define test data. Valid and invalid inputs can be generated automatically from such a specification [36]. Listing 4.1 is an example of how such a definition would look like for a simple user input form.

```
address = name, newline , living_address , newline , city ;
name = surname , ",␣", firstname ;
living_address = street , "␣", number;
city = city-name, ",␣", area code;
```

Listing 4.1: Extended Backus-Naur Form

Another good practice is to use XML as descriptive language. XML has the advantage that it has a well-defined standard provided by W3C [48] and it is widely spread. XML is simple to read and edit manually, and simple to parse automatically. It also has mature mechanisms for definition and validation which is good to detect errors already during the parsing of an XML document.

4.3 Automatic Test Execution

Automated tests are repeatable and therefore perfectly suitable for regression tests. One should think about the background of automated tests for the correct development phase. Depending on the situation there are several types of automated testing. When using sequential development models it is good to do occasional testing. For projects that use any form of an agile development technique and continuous integration the focus should be put on permanent test execution.

Permanent automatic test execution is important to give developers fast feedback about their code. If the practice of *continuous build* is used it is inevitable to test each build. Continuous build means that a build is triggered after every code contribution of each developer. The main goal of this type of test execution is not to test for new features but it represents a regression test, namely to test that all functionality covered by tests is still working after a code change.

Frequent automatic test execution embodies tests that are too complicated or too time-consuming to be executed after each build. The interval of the test execution can be a few days from one test to the other. In this category we have large unit test suites as well as automated integration or system tests. A prerequisite for automatic integration testing is that the deployment of the software modules is automated as well. It should be taken into consideration that complex automatic test frameworks are complex to maintain. It is therefore sometimes better to execute them not on a daily basis during development but after each development iteration [36].

Occasional execution is done if the testing effort is high. All available test cases should be executed at least once for each release. If it is not possible for whatever reason to execute all defined test cases the most important ones should be selected for execution using risk analysis.

4.3.1 Development of an Automatic Test Framework

According to Seidl [36] the development of an automatic test framework is to be considered as a development operation that demands an expert's know-how. A member of a test framework development team has to be a good developer, has to be familiar with the object that will be tested and has to have some understanding of software testing. A structured approach is mandatory. One should choose between these methods:

1. *Bottom-up*: The development begins at the bottom of the layered architecture. This part is represented by the interfaces that connect to the object to be tested. After that the higher levels like data processing, test control and test evaluation are implemented.

2. *Top-down*: The starting point of this method is the administration and processing of test data, and test sequences and therefore the layer of test execution.

It is furthermore good practice to divide the entire test framework into modules. Each module can then be developed by a different person if necessary. That results in a speed-up in development because all modules can be developed simultaneously. A test framework consists of modules such as test data administration and supply, interfaces to the tested object, test execution, test evaluation and test interpretation.

4.3.2 Automatic Test for Functionality

Depending on the nature of the tested software there are a number of aspects to be considered which are described in this subsection.

A common type of software architecture is a client-server-system. A lot of data is held at a central server program that is also responsible to implement the major part of the system's functionality. A *fat client* is a client with a lot of functionality whereas a *thin client* is a client with very little functionality. The first embodies a significant part of the system's behavior and the second is often only seen as a means of user interaction or input mask.

An important aspect for automated tests of such systems is the point if it is useful to test the graphical user interface (GUI) automatically. Testing

GUIs automatically is very error-prone and difficult. GUIs can change over time and depend very much on the user's environment such as the operating system or the screen resolution. If there is not much functionality in the GUI and the layer below can be accessed quite easily by test adapters it is recommended to inspect the graphical user interface manually and perform automated tests of the functional layer below it [36].

An aspect worth mentioning is the presence of concurrency in client-server-systems. The first approach here is to simulate *multiple clients on one test system*. That means to start several test clients on one physical computer. This method is often limited by the fact that a client is intended to be used only once per computer and only once per user. Therefore it can happen, that making such a client testable in a concurrent way is very difficult.

The second way to produce concurrency is to simulate *several clients on multiple physical machines*. The challenge of such a set-up is the manageability of several distributed clients and their control.

The most common concept is to use *several virtual test environments on one physical device*. This has the advantage that one defined test configuration can be easily transferred to a new virtual instance. Many development and test teams underestimate the administrative effort for such a project. Activities such as configuration changes and software updates have to be performed on each virtual machine. It is a good idea to think about an automation of such operations as well to keep the effort as low as possible.

Defined input-output-pairs are a good way to prove the correct behavior of a tested system. This method is applicable for all tested objects where it is possible to assign defined output data to defined input data. The advantage is that this is easy to be implemented and the test coverage is guaranteed. The downsides are that the test case number can be very high, and that the target system and the original system have to have the same data format.

4.3.3 Capture and Replay

The capture and replay technique is well-suited for testing graphical user interfaces. Sometimes recording the steps performed by a person when using a program is the easiest way to generate repeatable test cases. The advantage of that method is that once recorded the test sequence can be replayed automatically and does not consume human resources [33].

It is to mention that this method is limited by changes to the tested software. The recorded test steps will not work anymore if the software has been changed in a significant way. The number of development cycles and therefore changes to the user interface of the software must be rather low so that capture and replay pays off.

There is a possibility though to make capture and replay more resistant to changes in the graphical user interface. This is applicable if there is a certain logic, a server, data processing or data persistence behind the graphical user interface program (which in many cases is). One can perform the desired user steps and afterwards generate test cases out of the persisted data. Then those test cases can be used to perform automated test steps using the interfaces that are below the graphical user interface layer. To control the interfaces adapters are needed. The effort needed to convert data and to implement the adapters is compensated by the improved robustness of the test environment against changes in the graphical user interface.

4.4 When to Stop Testing

Gerald M. Weinberg explains the banana principle in his book [50]. According to it, a little boy comes home from school and his mother asks him what he had learned that day. The boy answers that he had learned how to spell the word *banana* but that he did not learn when to stop.

It is not easy to give certain numbers when software testing is finished or a software has been tested enough. There are too many variables in software environment to do that. A common practice is to declare the tests finished. In the following sections a set of useful test end criteria is described [13].

4.4.1 Coverage Goals

The term coverage means the amount of tested software parts compared to the amount of available software. The level of coverage definition can be made up from code coverage, functional coverage, use case coverage, system coverage and so on. The coverage percentage of a tested program part can be connected to the estimated risk of failure of that part. The opinions on coverage are controversial. Glenford Myers states that coverage is counterproductive [32]. He thinks that people that want to reach a coverage goal are likely to design their test cases in a way that the goal is reached as fast as possible (using so-called *quick-wins*). This often means that the test cases are not well-directed and only test the program parts that are the easiest to test. The more complex and hard to test program parts are not tested.

4.4.2 Defect Discovery Rate

The defect discovery criterion suggests to stop testing when the number of discovered defects during a defined time period (normally a week) falls below a certain threshold. There are some factors that can distort the accuracy of that practice. These can be testers on holiday or new test cases that are not that effective. Again as mentioned above it gets very clear at this point that following only one test ending criterion is insufficient. Also the pure number of discovered defects is not enough to determine the stopping of the test. There still can be serious errors in the software that prevent it from being shipped.

4.4.3 Marginal Cost

If we look at a manufacturing company that produces consumer goods the term marginal cost is used to describe the cost of adding one unit of production. Normally the higher the output of the production plant is, the lower the marginal cost is. In software engineering things look different. The first test cases are quite easy to write and the less possibilities are left, the more effort has to be put into testing them. If we come to the point that finding a new test case is more expensive, than delaying the release of the software product would cost, it is reasonable to stop testing.

4.4.4 Team Consensus

As the name indicates this test stopping aspect means that ending of the test is decided by the development team. The team states that the software is tested enough and that all major bugs are fixed. It is time to ship the software now and gather eventual other information about possible defects out of the production environment.

4.4.5 Ship It!

This is the criterion that is executed mostly in practice. The boss tells you to ship the software at a fixed date. The decision for that date is driven by factors like being the first to market. The first to market often has to take the risk of having a product that is not working correctly but that risk is compensated by the prospect of entering a new market as the first one and gaining a lot of customers.

Testers and software developers tend to see their work pessimistically. This means they only see the possible errors and cannot decide whether the product is good enough for the market in an objective way. Many of those people are perfectionists and want to develop the perfect piece of software. This has the disadvantage that the work is never finished within a reasonable time span. Therefore sometimes it is very useful that the management of a company sets goals and demands the meeting of a deadline.

Chapter 5

Test Environment and Continuous Integration Tools

A decent set of tools supports a well-structured development and testing process. In this chapter one possible combination of tools and its interaction is described. The tool-set should contain tools for source code management, automated build execution, static code analysis, unit tests, project module dependency management and storage. Furthermore the tools should be available for free. Since the Automated Test Environment is implemented in Java the tools presented here are best suitable for Java projects. The question of choosing the best source code editor is not discussed here, but the usage of Eclipse [46] is recommended.

5.1 Subversion

Keeping track of different versions of an evolving software project is essential in software development. A source code management system should be used in every case, even if there is only one developer working on a project [11]. A software that is developed to automate testing is in its roots nothing other than a normal software project. Therefore it has to follow the same conventions as the software that is tested by the test software. Keeping track of the test software, test plans, test logs and test cases can be done in a source code management system as well [33].

Subversion or in short *svn* is an open source project of the Apache Software Foundation [45]. It is a server software for source code management with basically the same functionality as cvs [14] or bazaar [5]. Subversion has some handy enhancements compared to cvs [31]. Before the Subversion project [44] joined the Apache Software Foundation in November 2009 it was a project hosted by Tigris.org [47].

5.1.1 Subversion Folder Structure

The root folder of a project in Subversion is called *repository*. The repository contains three sub-folders:

- trunk: In the trunk folder all current development source code is managed. Everything that has not been released yet is kept here. It is the current working directory where in normal development work all contributors store their code.
- tags: The tags folder contains one folder for each version of the software that has been released. A tag folder serves as a backup of exactly the same code version that is or was running in a production environment and therefore must not be changed.
- branches: Sometimes a developer is in the unpleasant situation to make a bug-fix for an older released version of a software that is still in production. The code from the trunk cannot be used for that because it contains changed or new functionality that is incompatible to the older version. In that case a sub-folder of the branches folder is generated with the content of the tags folder matching the version of the running software to be fixed. In the branches folder then the bug-fix is applied and a hot-fix version is released. One other possible scenario besides the bug-fix is that changed functionality of the trunk development path is needed in an older software version as a patch. The same procedure as for bug-fixing is applied here.

5.1.2 Software Version Nomenclature

A good choice of a version nomenclature for an average sized development project of up to 100.000 lines of code and a few developers is described here. The version string contains three numbers separated by a dot (x.y.z). For each release at least one of the numbers is changed. This is what the variables stand for:

- x is the major version that gets incremented only for major architectural and functional changes. A higher number is not compatible to the lower major version number.
- y is the minor version number that gets changed whenever a release with new features is made.
- z is the bug-fix version number that gets increased in case that no new features are added and only bugs are fixed in a release.

To mark the version of the currently developed and changing source code the appendix -*SNAPSHOT* is added to the version. The source code of this version is kept in the trunk-*svn*-folder. A software version that is denoted as not changing but not ready for release (e.g. because it is not yet tested) is called a release candidate. A release candidate adds *-RC* to the software version. It is important to test software against a not-changing version to make the found defects reproducible. For one release there can be more than one release candidates. This means we can have *-RC1*, *-RC2*, *-RC3* and so on for one and the same release version. It is good practice to make one release candidate per development iteration. An old software version that needs to be bug-fixed or patched is called hot-fix and adds *-hf* to the version number.

Example:
The version 2.1.4-RC2 means the second major version with first minor version and bug-fix version four - second release candidate.

5.2 Maven

Maven is a tool intended to facilitate build management and can be compared to Ant in some ways [42]. The biggest difference is the way Maven treats project dependencies. The dependencies are organized hierarchically. The project meta-data defined in Maven's build configuration file is called project object model (POM). Therefore each Maven project has to have a file called pom.xml (similar to Ant's build.xml) [43]. The following folders are the most important in a Maven project:

- src/main/java: The main folder for Java source code.
- src/main/resources: A folder for all resources such as xml or property files.
- src/test/java: This is where all unit test classes are put.
- src/test/resources: In this folder all resources that are used for unit tests are found.
- target: The produced artifacts (also known as distributable) are generated into that folder.

5.2.1 Maven Dependencies

The definition of a required external library is called dependency. A ready built distributable of a project is called an artifact. A repository is a location where Maven looks for artifacts that are defined as dependency in the pom.xml file. Project dependencies are inherited.

Example:

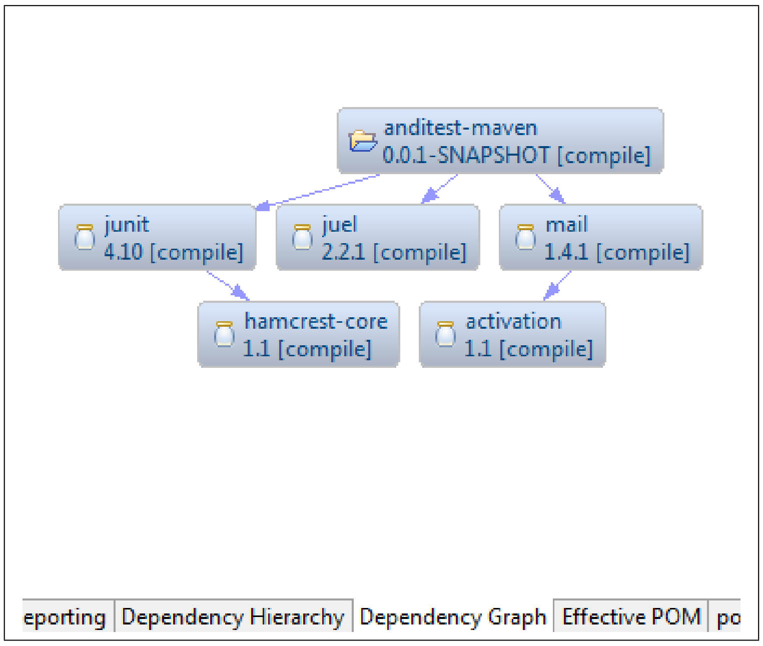


Figure 5.1: Maven Dependencies in a small Project

The developer wants to start implementing JUnit tests in his project called project A. Project A defines a dependency to project B because project A uses functionality of project B. Project B already contains a dependency to the JUnit artifact because in project B there are JUnit tests. For the developer this means that JUnit is also available in project A because it has inherited the dependency to JUnit from project B. There is no need for the developer to define a separate dependency to JUnit in project A because JUnit is already available.

Dependency inheritance works on multiple levels. This keeps the size of the pom.xml file very low. In a real-life project the dependency hierarchy looks as shown in Figure 5.1 for a small project, for a bigger project dependency hierarchy illustration see Figure 5.2.

5.2.2 Maven Repository Handling

Maven organizes the sources of artifacts of dependencies on multiple levels. At first there is a local repository on the machine where the build is executed. Normally this is located in the user-home directory. This is the first place where Maven looks for a required artifact. If the artifact (or the correct version of it) is not found in the local repository, Maven looks in other repositories that are defined in the pom.xml file. The artifact is then downloaded to the local repository to speed up future builds.

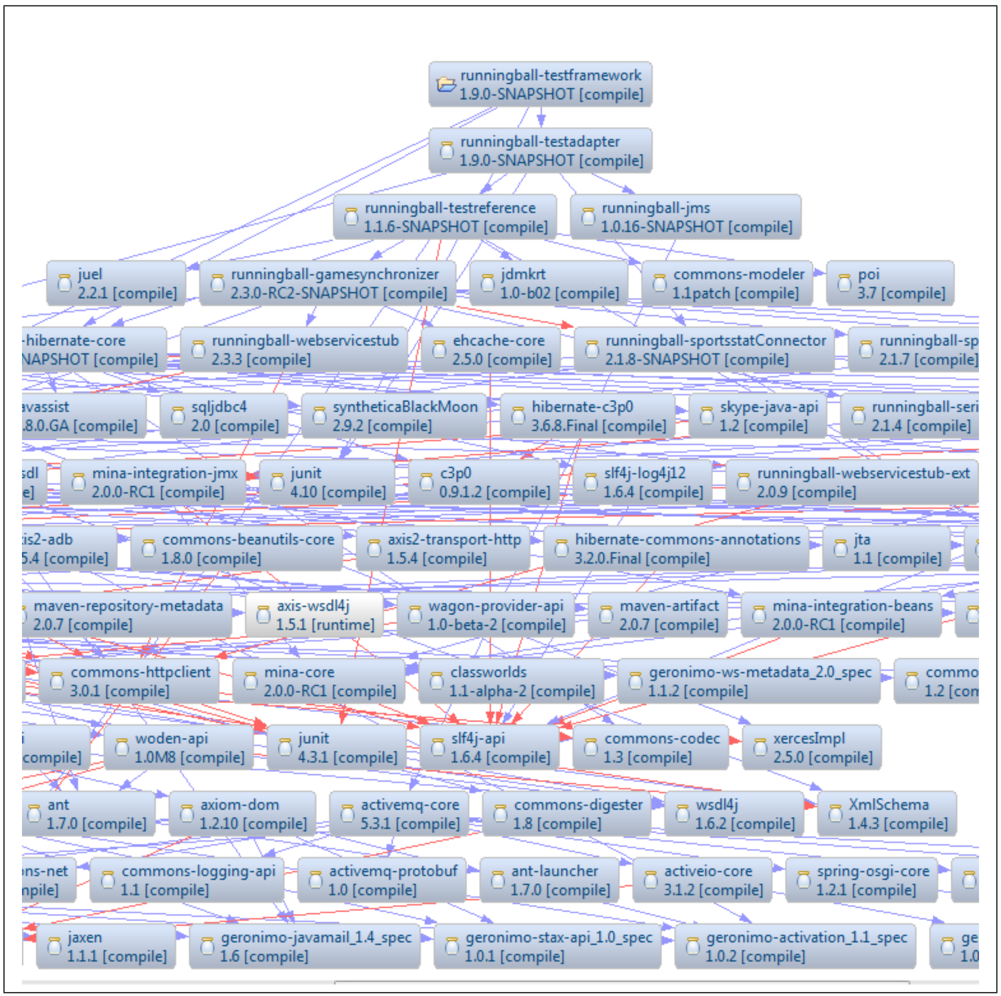

runningball-testframework 1.9.0-SNAPSHOT [compile]
runningball-testadapter 1.9.0-SNAPSHOT [compile]
runningball-testreference 1.1.6-SNAPSHOT [compile]
runningball-jms 1.0.16-SNAPSHOT [compile]
juel 2.2.1 [compile]
runningball-gamesynchronizer 2.3.0-RC2-SNAPSHOT [compile]
jdmkrt 1.0-b02 [compile]
commons-modeler 1.1patch [compile]
poi 3.7 [compile]
-hibernate-core NAPSHOT [compile]
runningball-webservicestub 2.3.3 [compile]
ehcache-core 2.5.0 [compile]
runningball-sportsstatConnector 2.1.8-SNAPSHOT [compile]
runningball-sp 2.1.7 [compile]
avassist .8.0.GA [compile]
sqljdbc4 2.0 [compile]
syntheticaBlackMoon 2.9.2 [compile]
hibernate-c3p0 3.6.8.Final [compile]
skype-java-api 1.2 [compile]
runningball-seri 2.1.4 [compile]
sdl e]
mina-integration-jmx 2.0.0-RC1 [compile]
junit 4.10 [compile]
c3p0 0.9.1.2 [compile]
slf4j-log4j12 1.6.4 [compile]
runningball-webservicestub-ext 2.0.9 [compile]
is2-adb 5.4 [compile]
commons-beanutils-core 1.8.0 [compile]
axis2-transport-http 1.5.4 [compile]
hibernate-commons-annotations 3.2.0.Final [compile]
jta 1.1 [compile]
maven-repository-metadata 2.0.7 [compile]
axis-wsdl4j 1.5.1 [runtime]
wagon-provider-api 1.0-beta-2 [compile]
maven-artifact 2.0.7 [compile]
mina-integration-beans 2.0.0-RC1 [compile]
commons-httpclient 3.0.1 [compile]
mina-core 2.0.0-RC1 [compile]
classworlds 1.1-alpha-2 [compile]
geronimo-ws-metadata_2.0_spec 1.1.2 [compile]
commo 1.2 [com
i compile]
woden-api 1.0M8 [compile]
junit 4.3.1 [compile]
slf4j-api 1.6.4 [compile]
commons-codec 1.3 [compile]
xercesImpl 2.5.0 [compile]
ant 1.7.0 [compile]
axiom-dom 1.2.10 [compile]
activemq-core 5.3.1 [compile]
commons-digester 1.8 [compile]
wsdl4j 1.6.2 [compile]
XmlSchema 1.4.3 [compile]
ns-net npile]
commons-logging-api 1.1 [compile]
activemq-protobuf 1.0 [compile]
ant-launcher 1.7.0 [compile]
activeio-core 3.1.2 [compile]
spring-osgi-core 1.2.1 [compile]
jaxen 1.1.1 [compile]
geronimo-javamail_1.4_spec 1.6 [compile]
geronimo-stax-api_1.0_spec 1.0.1 [compile]
geronimo-activation_1.1_spec 1.0.2 [compile]
ge 1.0

Figure 5.2: Extract of Maven Dependencies in a bigger Project

Many open source projects share their Maven artifacts in public repositories. This is sufficient and fine for some time. If there is more than one Maven project in one organization, and the organization does not want to upload and share its artifacts onto public repositories, the need for a company repository arises.

The requirements of a company repository are straightforward. We would like to store and retrieve artifacts during a Maven build, and have a basic web interface that displays the available artifacts with some meta information. Besides that it would be nice to upload artifacts manually and search for them on the web interface. A tool that supports all those operations is Artifactory [21]. Furthermore Artifactory operates as a proxy between the local build environment and the outside world. That means that it caches third-party artifacts that are available in a public repository outside the company network and by doing that, speeds up artifact retrieval and build times.

5.2.3 Maven Plugins

There is a large number of Maven plugins. Among them, there are plugins for xml validation, xml bean generation, test execution, artifact distribution, distributable generation and static source code analysis.

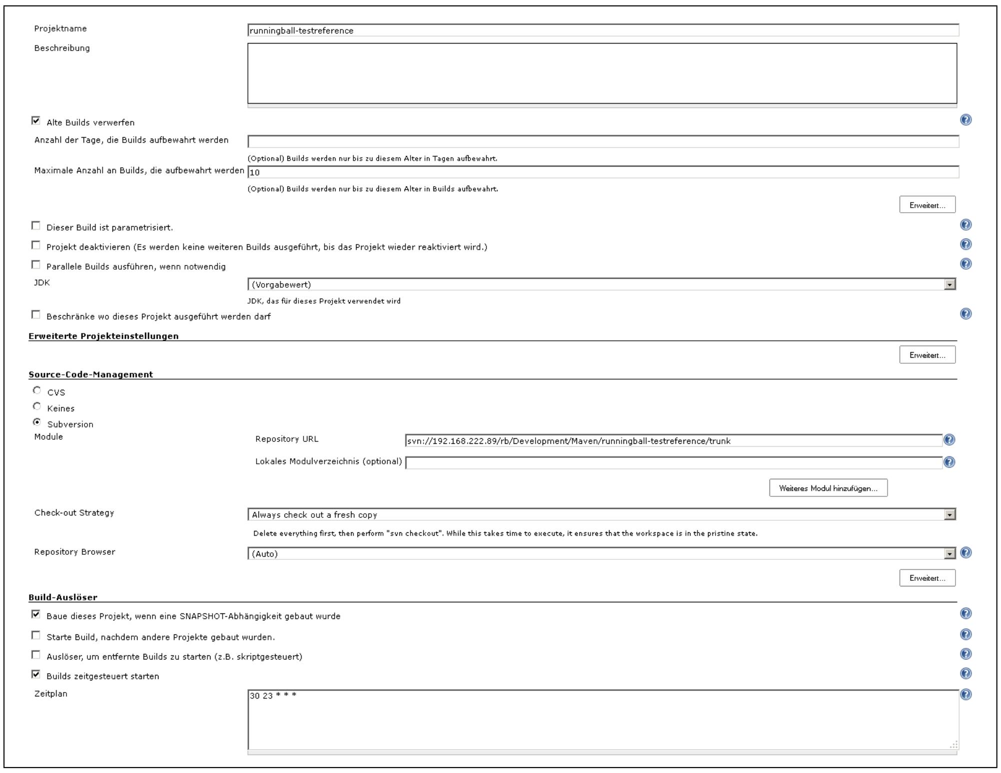
5.3 Jenkins

Jenkins formerly known as Hudson, is a tool to support continuous integration (CI) in software development. In its core Jenkins is a server software that can trigger pretty much anything that can execute something on a computer. The application area ranges from automated builds to automated data manipulation tasks and automated deployment. For each Jenkins task three steps are important [35]:

1. Determine if a new build is necessary.
2. Execute the build.
3. Process and visualize the collected data.

Configuration of Jenkins is simple but powerful. The most important configuration settings can be seen in Figure 5.3 and Figure 5.4.

Besides the mentioned configuration options Jenkins supports the possibility to grant different privileges to different users. For example a Jenkins project can be configured in a way that only the project responsible is allowed to perform releases.



Projektname: runningball-testreference

Beschreibung:

[✓] Alte Builds verwerfen

Anzahl der Tage, die Builds aufbewahrt werden:

(Optional) Builds werden nur bis zu diesem Alter in Tagen aufbewahrt.

Maximale Anzahl an Builds, die aufbewahrt werden: 10

(Optional) Builds werden nur bis zu diesem Alter in Builds aufbewahrt.

[ ] Dieser Build ist parametrisiert.

[ ] Projekt deaktivieren (Es werden keine weiteren Builds ausgeführt, bis das Projekt wieder reaktiviert wird.)

[ ] Parallele Builds ausführen, wenn notwendig

JDK: (Vorgabewert)

JDK, das für dieses Projekt verwendet wird

[ ] Beschränke wo dieses Projekt ausgeführt werden darf

Erweiterte Projekteinstellungen

Source-Code-Management

( ) CVS
( ) Keines
(●) Subversion

Module

Repository URL: svn://192.168.222.89/rb/Development/Maven/runningball-testreference/trunk

Lokales Modulverzeichnis (optional):

Check-out Strategy: Always check out a fresh copy

Delete everything first, then perform "svn checkout". While this takes time to execute, it ensures that the workspace is in the pristine state.

Repository Browser: (Auto)

Build-Auslöser

[✓] Baue dieses Projekt, wenn eine SNAPSHOT-Abhängigkeit gebaut wurde

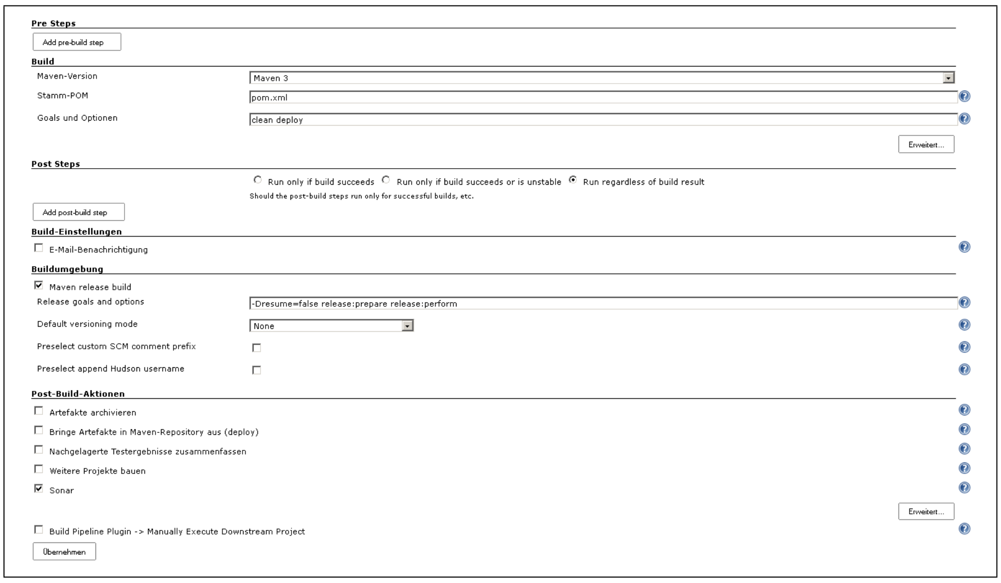
[ ] Starte Build, nachdem andere Projekte gebaut wurden.

[ ] Auslöser, um entfernte Builds zu starten (z.B. skriptgesteuert)

[✓] Builds zeitgesteuert starten

Zeitplan: 30 23 * * *

Figure 5.3: Jenkins Job Configuration Part 1



Pre Steps

Build

Maven-Version: Maven 3

Stamm-POM: pom.xml

Goals und Optionen: clean deploy

Post Steps

( ) Run only if build succeeds ( ) Run only if build succeeds or is unstable (●) Run regardless of build result

Should the post-build steps run only for successful builds, etc.

Build-Einstellungen

[ ] E-Mail-Benachrichtigung

Buildumgebung

[✓] Maven release build

Release goals and options: -Dresume=false release:prepare release:perform

Default versioning mode: None

Preselect custom SCM comment prefix: [ ]

Preselect append Hudson username: [ ]

Post-Build-Aktionen

[ ] Artefakte archivieren

[ ] Bringe Artefakte in Maven-Repository aus (deploy)

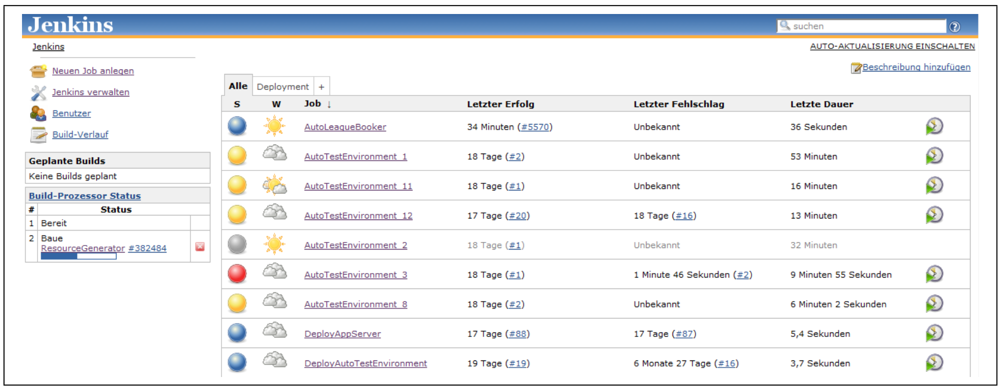
[ ] Nachgelagerte Testergebnisse zusammenfassen

[ ] Weitere Projekte bauen

[✓] Sonar

[ ] Build Pipeline Plugin -> Manually Execute Downstream Project

Figure 5.4: Jenkins Job Configuration Part 2



Geplante Builds
Keine Builds geplant

Build-Prozessor Status

#	Status
1	Bereit
2	Baue ResourceGenerator #382484

S	W	Job ↓	Letzter Erfolg	Letzter Fehlschlag	Letzte Dauer
●	☀	AutoLeagueBooker	34 Minuten (#5570)	Unbekannt	36 Sekunden
●	☁	AutoTestEnvironment_1	18 Tage (#2)	Unbekannt	53 Minuten
●	⛅	AutoTestEnvironment_11	18 Tage (#1)	Unbekannt	16 Minuten
●	☁	AutoTestEnvironment_12	17 Tage (#20)	18 Tage (#16)	13 Minuten
●	☀	AutoTestEnvironment_2	18 Tage (#1)	Unbekannt	32 Minuten
●	☁	AutoTestEnvironment_3	18 Tage (#1)	1 Minute 46 Sekunden (#2)	9 Minuten 55 Sekunden
●	☁	AutoTestEnvironment_8	18 Tage (#2)	Unbekannt	6 Minuten 2 Sekunden
●	☁	DeployAppServer	17 Tage (#88)	17 Tage (#87)	5,4 Sekunden
●	☁	DeployAutoTestEnvironment	19 Tage (#19)	6 Monate 27 Tage (#16)	3,7 Sekunden

Figure 5.5: Jenkins Job Overview

5.3.1 Jenkins as Build Automation Tool

On the Jenkins main page there is a list of all jobs that are known to Jenkins. The state at which each job is, is indicated by the color of the ball icon (see Figure 5.5):

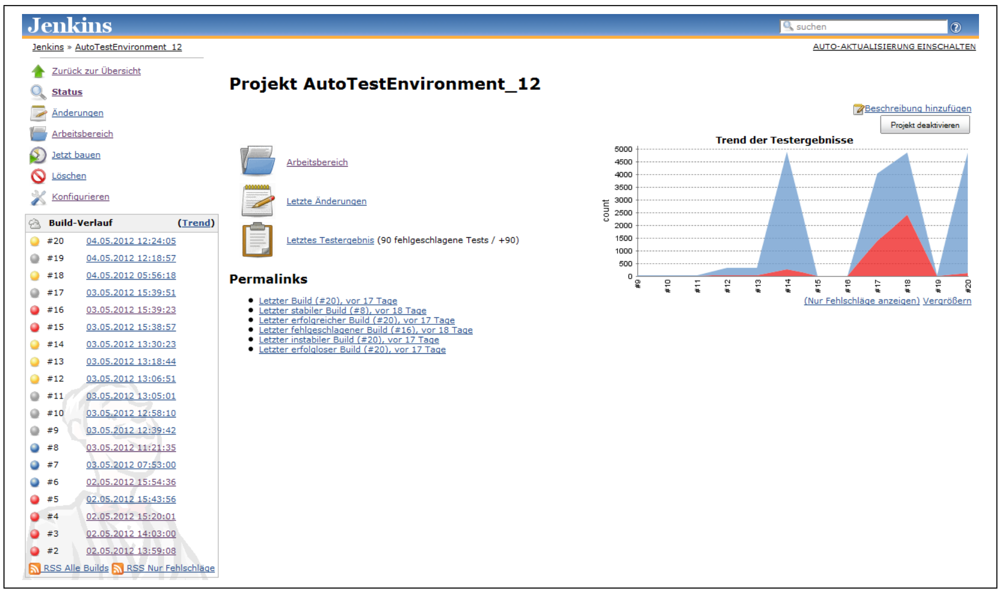
- *Blue* ball means build OK and stable.
- *Yellow* ball means build OK but test failures.
- *Red* ball means build failure.
- *Grey* ball means that this job is deactivated.

The last successful build time and the duration of the last successful build are displayed as well. Furthermore the tendency of the builds is visualized using weather icons (See Figure 5.5).

The execution result of a build or any other job that is configured for Jenkins is called *build*. The data generated by a build is called build artefact. One aspect why Jenkins is so well-suited for CI is that Jenkins keeps track of the build execution results. The user can configure how many artefacts are archived. So it is possible to monitor a project's progress. For example, the last build results as well as the number of executed Unit tests and their outcome is displayed in a chart as seen in Figure 5.6.

5.3.2 Jenkins as Deployment Automation Tool

One of the key factors of continuous integration and test automation is fast and easy deployment. Jenkins can be used to execute ant tasks that perform a deployment with certain parameters such as a version of a program to be deployed. Also the restoration of a database and its conversion into a test database can be achieved by using SQL scripts that are executed by a


Jenkins

suchen

Jenkins » AutoTestEnvironment_12

AUTO-AKTUALISIERUNG EINSCHALTEN

Zurück zur Übersicht
Status
Änderungen
Arbeitsbereich
Jetzt bauen
Löschen
Konfigurieren

Build-Verlauf (Trend)

🟡	#20	04.05.2012 12:24:05
⚪	#19	04.05.2012 12:18:57
🟡	#18	04.05.2012 05:56:18
⚪	#17	03.05.2012 15:39:51
🔴	#16	03.05.2012 15:39:23
🔴	#15	03.05.2012 15:38:57
🟡	#14	03.05.2012 13:30:23
🟡	#13	03.05.2012 13:18:44
🟡	#12	03.05.2012 13:06:51
⚪	#11	03.05.2012 13:05:01
⚪	#10	03.05.2012 12:58:10
⚪	#9	03.05.2012 12:39:42
🔵	#8	03.05.2012 11:21:35
🔵	#7	03.05.2012 07:53:00
🔵	#6	02.05.2012 15:54:36
🔴	#5	02.05.2012 15:43:56
🔴	#4	02.05.2012 15:20:01
🔴	#3	02.05.2012 14:03:00
🔴	#2	02.05.2012 13:59:08

RSS Alle Builds RSS Nur Fehlschläge

Projekt AutoTestEnvironment_12

Beschreibung hinzufügen

Projekt deaktivieren

Arbeitsbereich

Letzte Änderungen

Letztes Testergebnis (90 fehlgeschlagene Tests / +90)

Permalinks

- Letzter Build (#20), vor 17 Tage
- Letzter stabiler Build (#8), vor 18 Tage
- Letzter erfolgreicher Build (#20), vor 17 Tage
- Letzter fehlgeschlagener Build (#16), vor 18 Tage
- Letzter instabiler Build (#20), vor 17 Tage
- Letzter erfolgloser Build (#20), vor 17 Tage

(Nur Fehlschläge anzeigen) Vergrößern

Figure 5.6: Jenkins Job Progress

Jenkins-triggered ant task. A Jenkins job can be executed manually by the user or on a regular schedule. To summarize, Jenkins is a vital tool for test automation and continuous integration.

5.4 JUnit

JUnit [23] is a unit testing framework for Java. Its intention is to test small pieces of code uncoupled from its dependencies to other code units. Unit testing is not integration or system testing where the interaction of code classes is tested. In the test hierarchy, unit testing is the first step to avoid the most primitive coding errors in the most primitive code parts.

JUnit was developed by Kent Beck and Erich Gamma to accompany their method of test-driven-development (*TDD*) [8].

5.4.1 Annotations in JUnit

Using annotations to test methods the developer is able to control what is executed and when. The most important annotations for JUnit are [29]

- *@Test*: Marks a test method that is executed by JUnit.
- *@Before*: Denotes that the following method is executed before each test method, to perform test preparations.
- *@BeforeClass*: The method is executed only once before the test class is instantiated.

- *@After*: This is the counterpart of *@Before* and is executed after each test method.
- *@AfterClass*: This is the counterpart of *@BeforeClass* and is executed after all test methods of a test class have been called to perform a clean-up.

5.4.2 Assertions in JUnit

JUnit differs between two erroneous test result states: *failure* and *error*. A failure is the unsuccessful evaluation of an assertion (i.e. wrong data is returned by the tested method) and an error is a program error occurred during the test execution (for example an uncaught exception in the tested method).

JUnit provides the following assertion methods [8]:

- *assertEquals(...)* tests if the two given objects have the same values.
- *assertTrue(...)* tests if the given boolean is true.
- *assertFalse(...)* tests if the given boolean is false.
- *assertNull(...)* tests if the given object is null.
- *assertNotNull(...)* tests if the given object is not null.
- *assertSame(...)* tests if the two given objects refer to the same object.
- *assertNotSame(...)* tests if the two given objects refer to different objects.

It is important to say that the assertion methods always take the expected object as the first input parameter (if needed) and then the actual object retrieved from the test execution. There is also the possibility to provide tolerance values for the assertion of double values to avoid a misleading failure-test-result because of round-off errors.

5.4.3 Simulated Behavior using Mock-Objects

In the JUnit execution context we have only a very limited amount of connectors to other software parts. Furthermore we do not want the test to use real database connections or real web-service connections to guarantee the test's independence of data changes and to ensure that the test is executed as fast as possible. Therefore the need for simulated environments arises. Hence, the creation of so-called *Mock* objects [29]. A Mock object can be seen as a mechanism to decouple the test class from any other complex program parts that it depends on. Common modules that are replaced by Mock objects are persistence layer classes or any other connector classes that require external infrastructure [41].

The challenge of using Mock objects is that their number grows over time and that the interfaces implemented in the Mock objects are subject to change. This means a rather high maintenance for the administration of Mock objects. One way to solve this issue is the usage of a Mock object framework that can provide Mock objects automatically. If we would like to unit-test the functionality of a database persistence layer, one solution is to replace the required database server with an in-memory database such as H2 [17].

5.5 Sonar

Last but not least, an important component of the test and development environment is a tool for static code analysis. Static code analysis means the assessment of source code without executing it. A tool that performs static code analysis searches for predefined code patterns that bare the risk of failure.

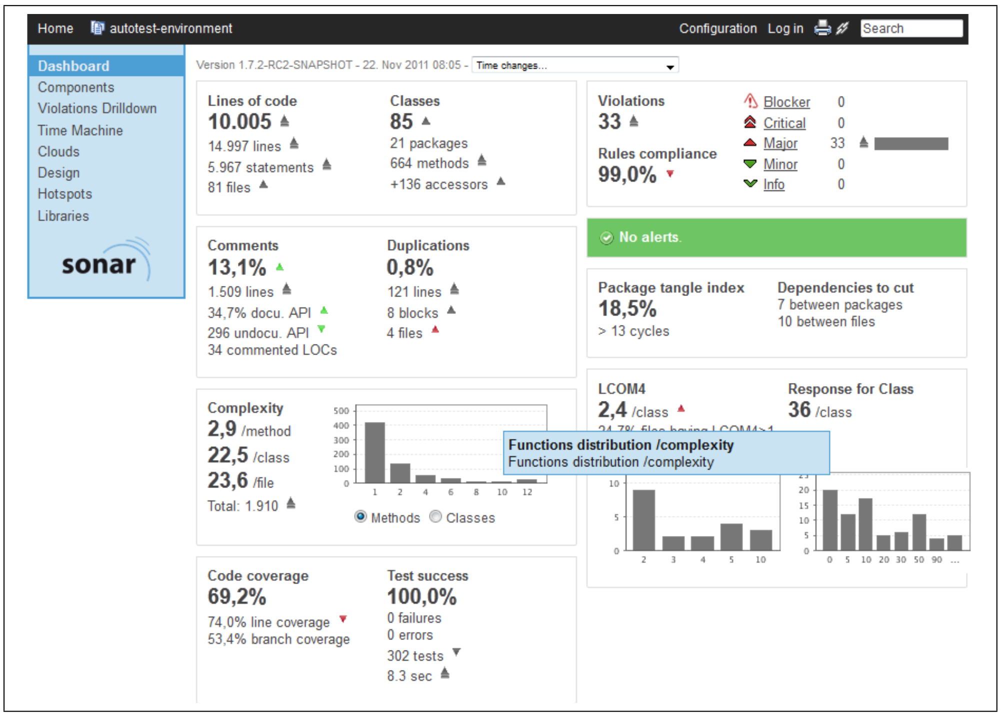
In Section 5.2 we talked about Maven which has a plugin called Maven Site Plugin [1]. This plugin is quite easy to configure and can display some of the basic project metrics. The biggest limitation is that with the Maven Site Plugin it is not possible to generate meta-information for more than one project.

Of the six main aspects of software quality as presented in Figure 2.2 Sonar addresses reliability, efficiency, maintainability and portability.

5.5.1 Overview

Sonar [37] is a platform that provides the following features:

- Produces static analysis project data during the Maven build process with Maven plugins such as PMD [40], Checkstyle [38] and Cobertura [39].
- Writes the data into a database.
- Displays the data on a web interface.
- Configures the applied analysis rules.


Home autotest-environment Configuration Log in Search

Dashboard
Components
Violations Drilldown
Time Machine
Clouds
Design
Hotspots
Libraries

Version 1.7.2-RC2-SNAPSHOT - 22. Nov 2011 08:05 - Time changes...

Lines of code
10.005
14.997 lines
5.967 statements
81 files

Classes
85
21 packages
664 methods
+136 accessors

Violations
33

Rules compliance
99,0%

Blocker	0
Critical	0
Major	33
Minor	0
Info	0

No alerts.

Comments
13,1%
1.509 lines
34,7% docu. API
296 undocu. API
34 commented LOCs

Duplications
0,8%
121 lines
8 blocks
4 files

Package tangle index
18,5%
> 13 cycles

Dependencies to cut
7 between packages
10 between files

Complexity
2,9 /method
22,5 /class
23,6 /file
Total: 1.910

Methods Classes

LCOM4
2,4 /class

Response for Class
36 /class

Functions distribution /complexity
Functions distribution /complexity

Code coverage
69,2%
74,0% line coverage
53,4% branch coverage

Test success
100,0%
0 failures
0 errors
302 tests
8.3 sec

Figure 5.7: Sonar Project Dashboard

5.5.2 Usage

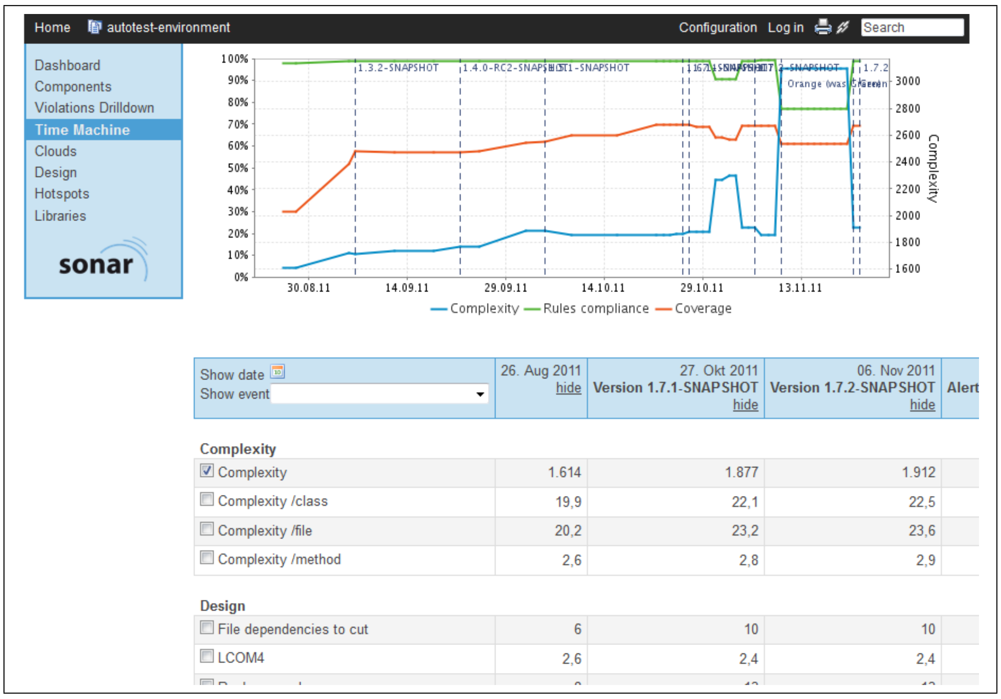
On the Sonar project dashboard you can see a summary of the most important project metrics. In Figure 5.7 project attributes representing the project's size, complexity and quality are shown.

In the Sonar Time Machine the development of a project can be seen (Figure 5.8). The progress in complexity, rules compliance and code coverage are visualized.

The code coverage can be displayed in an even more detailed way related to the analyzed source code as seen in Figure 5.9.

Red lines mean lines of code that are not covered by unit tests. If there is a number with green background to the left in one line, the number represents the number of times a line of code has been executed by a unit test. Yellow lines describe conditions that have not been tested entirely. A number of 2/8 for example means that only two out of eight possible conditions have been tested. According to Pezzè and Young [33] the most important static metrics can be grouped into two categories, code size and code complexity.

Another advantage of performing static code analysis is to embrace a coding standard. As Bath and McKay [4] point out in their book, a common coding standard among a team of developers has a lot of benefits. A code



Show date Show event	26. Aug 2011 hide	27. Okt 2011 Version 1.7.1-SNAPSHOT hide	06. Nov 2011 Version 1.7.2-SNAPSHOT hide	Alert
Complexity				
☑ Complexity	1.614	1.877	1.912	
☐ Complexity /class	19,9	22,1	22,5	
☐ Complexity /file	20,2	23,2	23,6	
☐ Complexity /method	2,6	2,8	2,9	
Design				
☐ File dependencies to cut	6	10	10	
☐ LCOM4	2,6	2,4	2,4	

Figure 5.8: Sonar Time Machine

sticking to a coding standard can be shared easier between developers, and improves maintainability and testability.

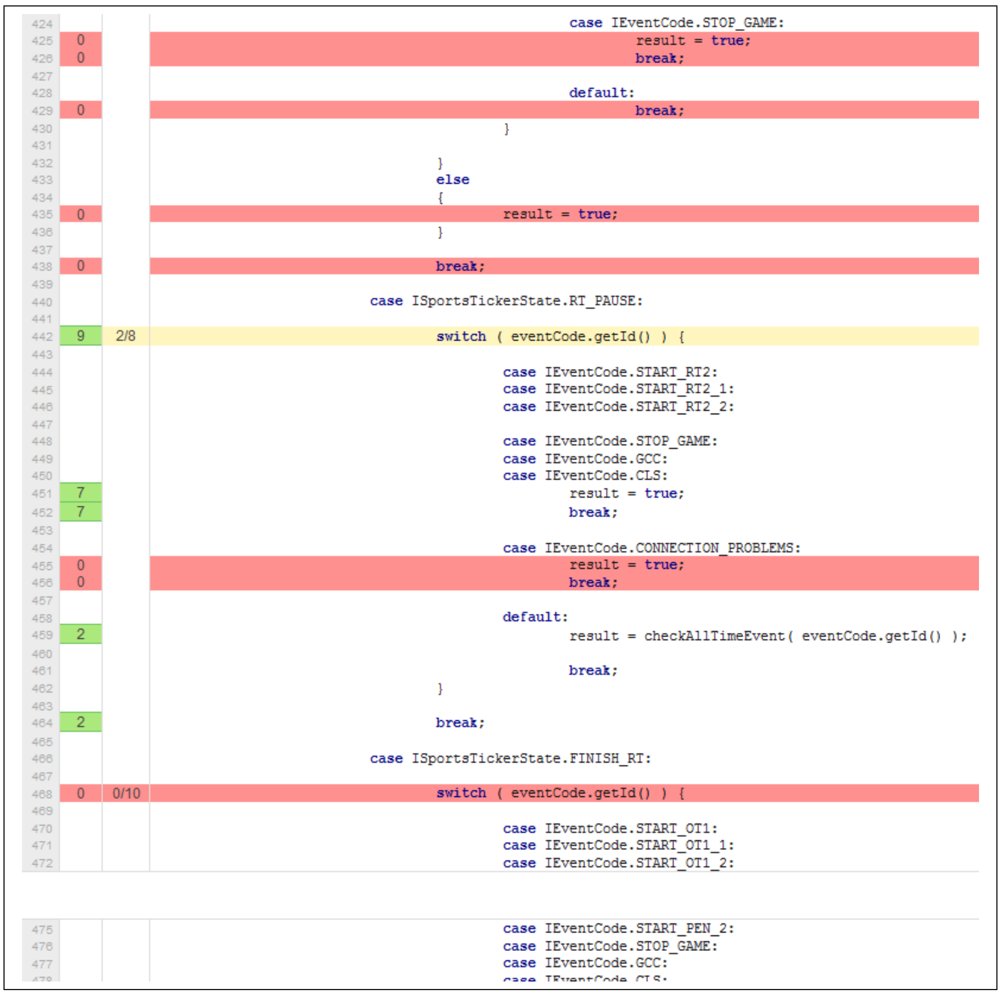
Sonar should be used to develop a coding standard in a development team. It should push developers to improve their coding habits. Whenever an error is found that can be related to a bad coding standard, a new rule detecting such an error has to be activated in Sonar to avoid future errors of the same kind. If the rule is activated, all source codes are checked for that rule and the error should never occur in a production environment again.

5.6 Bringing it all Together

Depending on his role in the development team each person has special wishes to the CI environment and different perspectives. The combination and usage of the five tools presented in the last few sections is explained in this section.

5.6.1 Developer

Figure 5.10 shows the CI environment from the developer's point of view. A developer works with an Integrated Development Environment (IDE) and produces program code that is built locally on the development machine



```
                        case IEventCode.STOP_GAME:
0                               result = true;
0                               break;

                        default:
0                               break;
                }

        }
        else
        {
0               result = true;
        }

0       break;

case ISportsTickerState.RT_PAUSE:

9   2/8 switch ( eventCode.getId() ) {

                case IEventCode.START_RT2:
                case IEventCode.START_RT2_1:
                case IEventCode.START_RT2_2:

                case IEventCode.STOP_GAME:
                case IEventCode.GCC:
                case IEventCode.CLS:
7                       result = true;
7                       break;

                case IEventCode.CONNECTION_PROBLEMS:
0                       result = true;
0                       break;

                default:
2                       result = checkAllTimeEvent( eventCode.getId() );

                        break;
        }

2       break;

case ISportsTickerState.FINISH_RT:

0   0/10 switch ( eventCode.getId() ) {

                case IEventCode.START_OT1:
                case IEventCode.START_OT1_1:
                case IEventCode.START_OT1_2:

                case IEventCode.START_PEN_2:
                case IEventCode.STOP_GAME:
                case IEventCode.GCC:
                case IEventCode.CLS:
```

Figure 5.9: Sonar Code Coverage

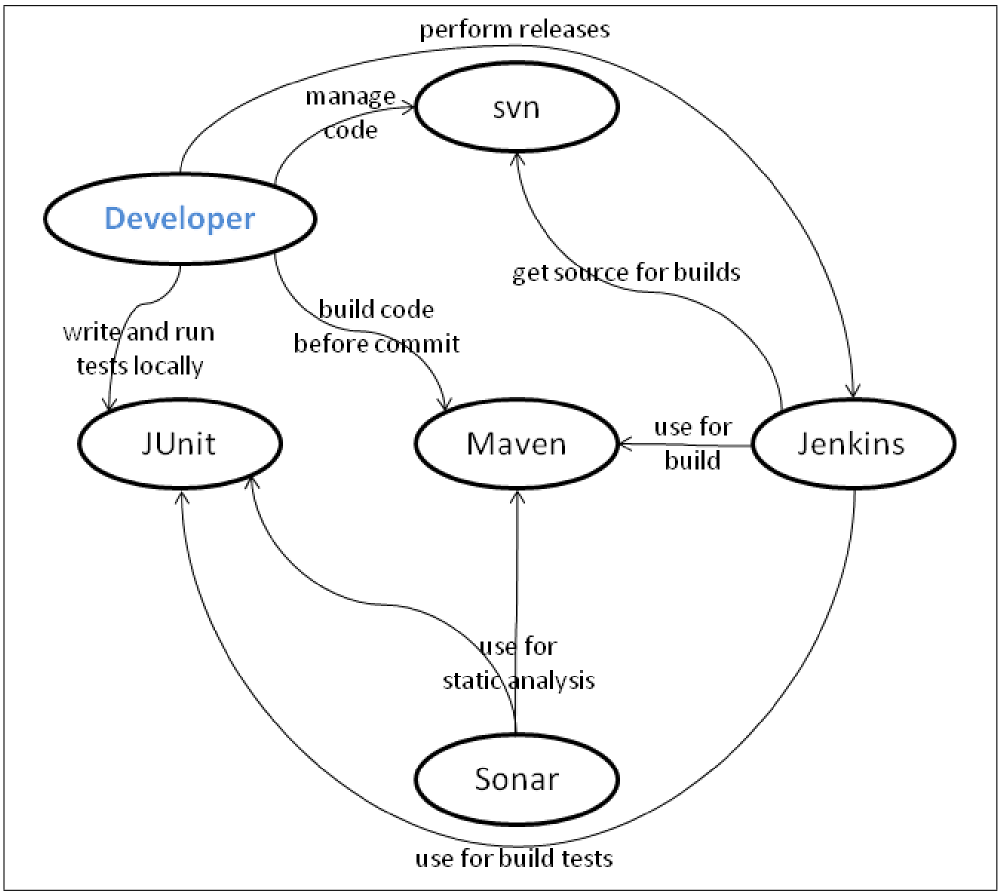


Figure 5.10: Developer's point of view

using Maven. Existing JUnit test cases are executed during that build. At this stage the developer knows that his changes work with the current JUnit tests and with the current Maven artifacts found in Artifactory. If no tests fail and the build is successful the developer stores the new source code into Subversion.

During each build Jenkins performs a Sonar analysis of the source code using the Maven code analysis plugins and the activated rules. If all important rules and metrics are met by the project it is ready to be released.

At the end of a development cycle the developer responsible for a project performs a release using Jenkins. Jenkins proposes the new version of the released artifact, tags [1] the released project and commits the new snapshot into Subversion.

5.6.2 Tester

As shown in Figure 5.11 a software tester manages the test cases in Subversion. He uses Jenkins to execute builds and tests. In Jenkins the test progress and the test results are shown. Looking into Sonar's analysis details, the tester can easily identify potential risks in a project and develop well-targeted test cases.

5.6.3 Software Architect

The software architect is responsible to keep an overview of the software architecture. The CI environment provides him with much important information as shown in Figure 5.12. Jenkins visualizes broken builds and test failures. The software architect is the person that designs the artifact landscape in Maven, meaning that he is the one that creates new projects and manages their dependencies.

The software architect must also act as a coach for the developers. Sonar helps to maintain and push code quality and the used coding standard. Jenkins should be configured by the software architect as well.

5.6.4 Project Manager

The project manager retrieves knowledge about the health and quality progress of a project using Sonar and Jenkins. Figure 5.13 shows the important aspects of the CI environment for the project manager. He can watch the graph displayed in the Sonar Time Machine (Figure 5.8). Based on that information he recognizes possible delays in the development schedule expressed by previously defined coverage and targets that are not yet met. Furthermore he can observe the number of failed tests in comparison

[1] In this context 'to tag' means to create a new folder named after the released version number in the Subversion tags folder and put the source code there.

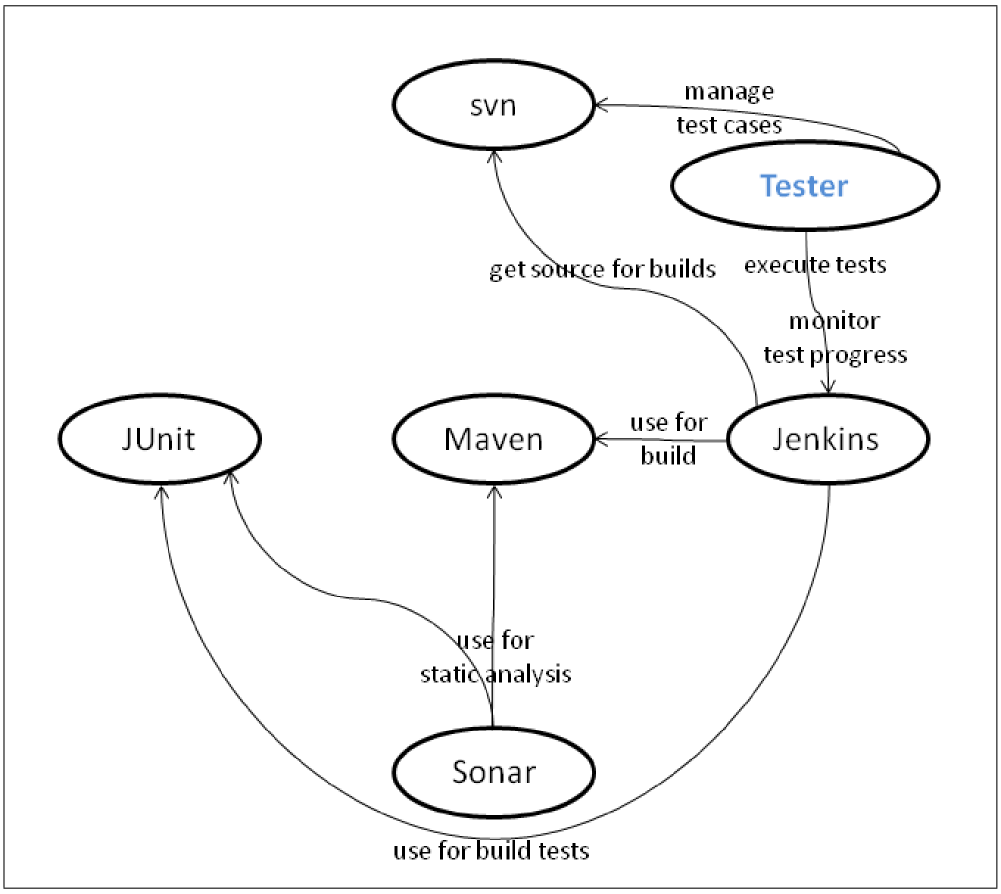


Figure 5.11: Tester's point of view

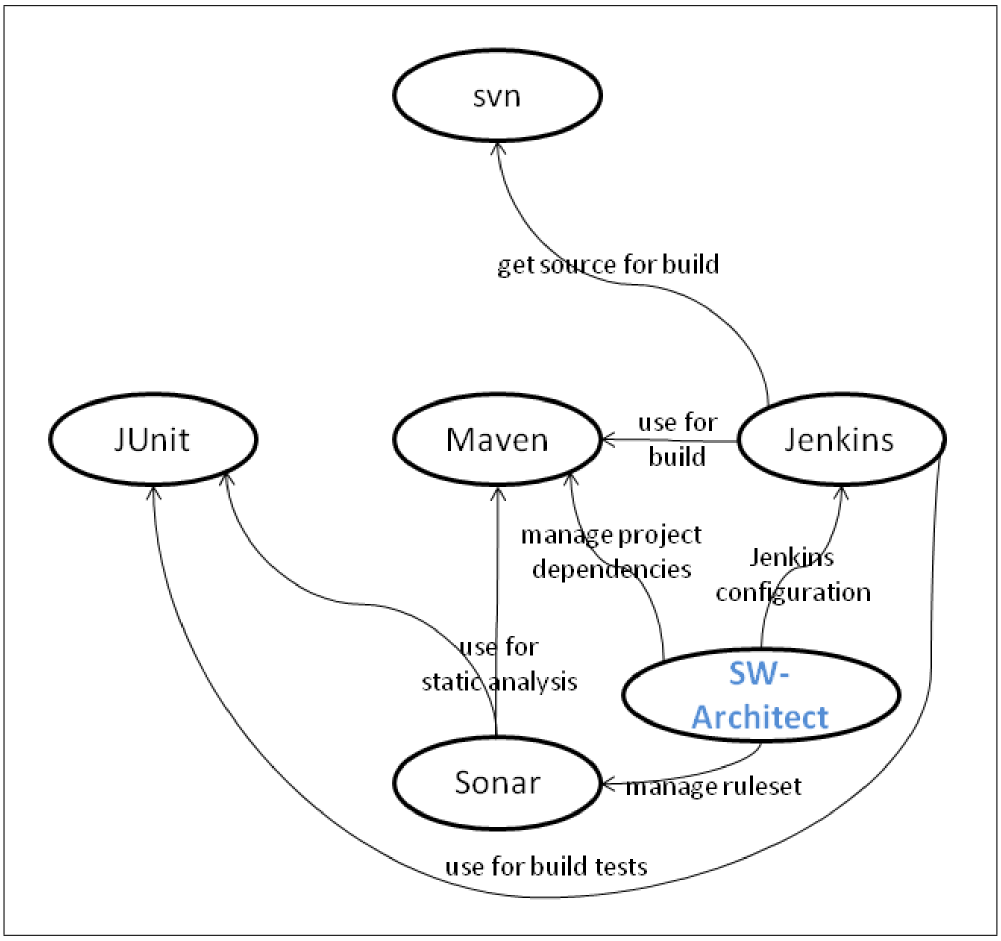


Figure 5.12: Software Architect's point of view

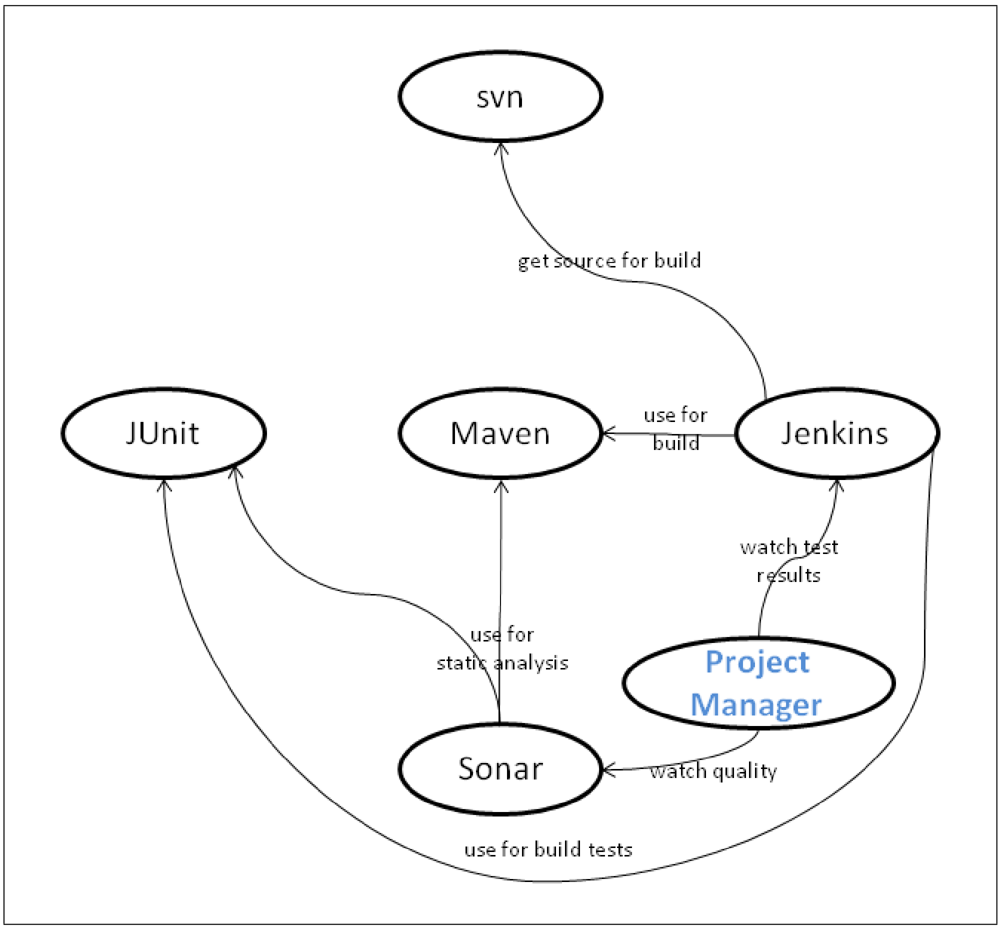


Figure 5.13: Project Manager's point of view

to the number of executed tests. All other information is too technical and therefore not usable by a manager for his work.

Chapter 6

Human Aspects of Software Testing

In all the previous sections we have talked about the technical aspects of software testing. One vital factor is missing though: the human being. This section sketches some of the human aspects connected to software testing and developing software testing processes.

6.1 The Nature of Testing

Testing is the process of executing a program with the intent of finding errors [32]. That is in short, a very precise definition of testing. There are many interpretations of testing though that are wrong and should be avoided in practice.

Testing can be understood by the developer to be meant to find the developer's personal errors and to emphasize that developers make mistakes. Many developers feel annoyed by the thought that tests are only made to find their errors. They are not. They are meant to support developers to deliver a product as well functioning as possible.

Testing does not exist to show that a program is working correctly, but it adds effort and value to the program by increasing its quality and reliability. If the tester is only driven by the idea to show that a program works correctly, he will only develop test cases that reflect the expected behavior and the correct usage. But good testing means much more, to be specific: the development of test cases that cover many situations of mis-usage of the program.

Showing that errors exist in a program is not a bad thing. Each test case that detects an error is a valuable test case, much more than a test case that works after the first try. Testing is a process that should accompany the development process. Failed tests must not be seen as a failure but as a support on the way to a stable software.

6.2 Breaking with Habits

Introducing an automated test framework that was not there before means breaking with tradition. At first developers will react very skeptical. They will feel controlled and agitated in their world. A special role is fulfilled by the person that introduces test automation. He is the one that has to establish a reliable test environment and to convince the developers that testing is a good thing that supports them in their daily work.

Furthermore the test team embodies the following important functions [20] and will:

- Compensate a lack of analytical skills and management faintness.
- Support junior programmers to learn the context faster.
- Provide a knowledge base stored in the test cases.
- Support the problem solving process.

In my personal opinion testing is:

- A very good way to take some pressure off the developers. They can concentrate more on the development work without having to fear that anything breaks if they can rely on good tests.
- The most important factor in ensuring excellent quality and getting results.
- A very good way to search for the requirements needed for a functional software.
- Not the answer to every problem. A good tester is creative and can communicate with the developers.

Part II

Application to a Project

Chapter 7

Project Prerequisites

The first chapter in the practical part will give an overview of the context where the Automated Test Environment is implemented. This comprises a company profile, test practices and project goals. After that a detailed description of the project phases and results is given.

7.1 Company Profile

RunningBall Global Sports Data is a company that was founded in the year 2006 in Graz. It is focused on real-time sport information over the internet and concentrates on the B2B-market. The idea is to sell the fastest possible information on running sport events. Among the customers, there are betting companies and sport analysts as well as sport clubs. RunningBall coverage is faster than TV coverage which usually is delayed by 5 to 12 seconds because of satellite-transmission [34].

The company's backbone is its global network of about 1100 scouts. They are match reporters responsible in attending sport venues all over the world and collect the information for RunningBall directly on-site. The information is entered into a smartphone and transferred via internet to the company's main server. This server then distributes the data to customers that use the Trader Client (RunningBall's graphical client program) or to the customer's servers for further automatized data processing as explained in Figure 7.1.

RunningBall is currently collecting and selling data of approximately 30.000 sport events per year. This makes the company one of the biggest suppliers in sport live data worldwide. Football was the first sport implemented, then followed basketball, tennis, snooker, darts and ice hockey. The company headquarters is located in Switzerland, and there are four other offices, among them, the office in Graz where the software development and maintenance is situated. In the software development department there are nine software developers, two testers, one project manager, one requirements

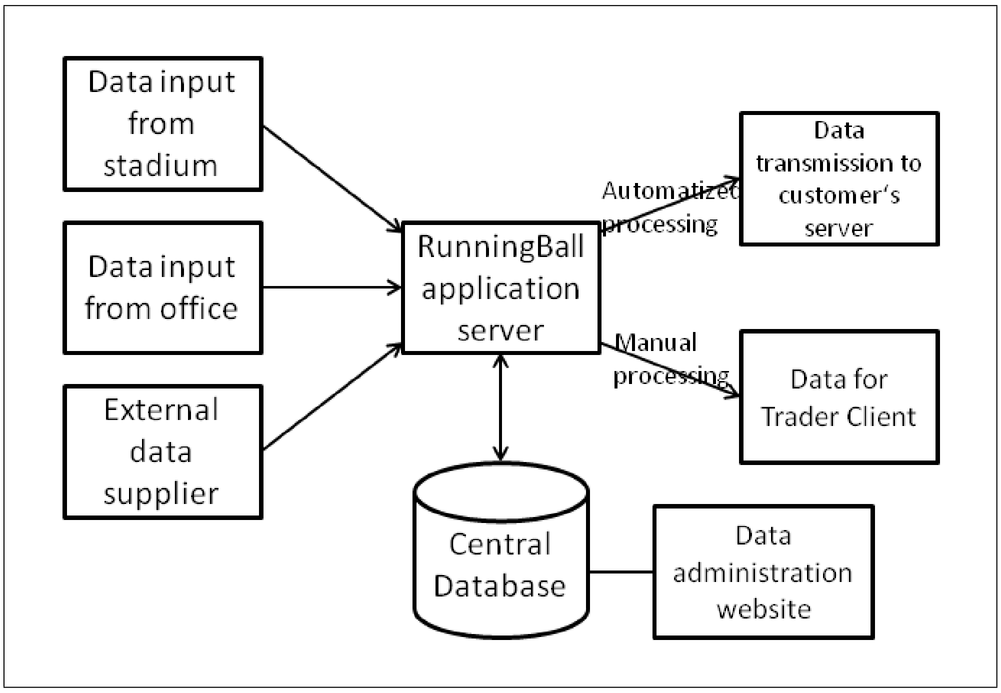


Figure 7.1: The RunningBall System

engineer, one head of development and one CTO (September 2012).

7.2 Important Terms

Ticker or *Sports Ticker*
A ticker or sports ticker denotes the digital equivalent to a sport game. It has a unique identifier and a sport type (one of the sports mentioned above). Besides that the country, the venue, the league and the competing teams or players are defined by a ticker.

Event
An event is a clearly recognizable happening during a ticker. For example if the ticker is of type football, a goal or a red card is an event in it.

Automatically generated event (AGE)
The RunningBall Application Server automatically generates events that emerge out of the context or combination of other manually entered events. The reason to generate some events automatically is to ease the usage of the input devices and to maximize the information content of RunningBall's data. Simple examples for automatically generated events of football are:

- When the home team has the ball and attacks the away team, the event

Attack is entered. If the ball approaches the away team's penalty box, the event *Danger* - which stands for a situation where a goal might happen within a short time - is entered. Out of the two events *Attack* and *Danger* the application server generates the event *Dangerous Attack* because it can be logically deducted from the previous two events.

- After a *Foul* by the away team committed on the home team the event *Free kick* is generated for the home team. That is because after a foul in soccer there is always a free kick for the other team.

Scout
A scout is a person that gets paid by RunningBall for collecting data which means watching a sport game and entering events into a special data input device.

Trader Client
The Trader Client is an application with a graphical user interface with which a customer can order and watch events for a ticker. Furthermore the Trader Client implements the possibility to display an overview of the upcoming sports tickers, historical information for teams and a billing view to display the current invoice amount. A customer can order sports tickers in the trader client that are then optionally transferred to him via the automatic data transmission service as well.

Master Data
At RunningBall the term Master Data denotes all data related to a ticker that can be considered static or rarely changing. This means country information, leagues, teams, stadiums and season information.

Ticker Statistics
Statistics are best described as the cumulated number of events of one type up to a specific moment. Statistics do not have to be entered by the scout because they emerge out of the entered events automatically. For example, if we have the events Goal Home, Goal Away and Goal Home for soccer, the corresponding statistics for the last goal event would be 2-1 which is the current score at the time of the last goal. There is a big set of further statistic-relevant events, such as corners, free kicks, penalties, red cards, yellow cards, etc. A statistic value pair is always connected to an event.

Value Event
An Event can have one or more value events providing it with more detailed information. The value event is a mechanism to enrich a normal event with information of any type.

7.3 Test Practices in the Company

In the company there are two testers responsible for a software system that is fast growing in complexity and size. The tested applications consist of different client applications, one server application and administrative websites. Regression tests are performed manually for each release. This means for a tester to look at the expected event processing order and enter the events into a client device manually in a way a scout would do it during his work. After entering events, their output data has to be verified in the Trader Client. Because of the increasing complexity of the software products regression tests of already existing functionality consume more time than the tests of the newly added features.

Test cases are written and tests are performed after implementation of the tested software. Developers have to wait for quite a long time for feedback from the test department about the test results. Most features are not written down in a formal language and only discussed in more detail if the implemented behavior does not match the expected behavior of the software.

7.4 Project Goals

During the initial meeting the expected features of the project called Automated Test Environment were defined and written down. Three priority categories have been selected to group the desired features.

7.4.1 Very Important Features

- Functional tests of the event processing logic (application server) performed as system test
- Component tests: Integration into the build system, nightly build (use the Automated Test Framework to be executed as unit test for the application server)
- Test framework
  - System test
  - Dynamic data basis: prepare the tested system, create a ticker in the system, wait until the new ticker is known by all components, log into the system, execute a simulated game.
  - Generate test cases, execute them and evaluate them for each test run. Monitor test progress.
- Introduction of test failure categories to be used in the evaluation

- Test case maintenance must be easy (XML format, maybe with an editor)
- Connection between test cases and use cases via the use case id.

7.4.2 Important Features

- Test all system interfaces that connect from and to the application server
- Test adapters should support different versions of the tested interfaces

7.4.3 Nice-to-have Features

- Perform a load test with an evaluation of the processing times
- Test the graphical user interface of Android and Swing client applications and not only their interfaces and protocols below

Chapter 8

Project Phases of the Automated Test Environment

The Automated Test Environment had to be stable and well functioning from the beginning on. Furthermore it should show its additional benefits quite fast. That is why the following phases have been chosen.

8.1 Standalone Application

At RunningBall all events for each ticker are stored in a database. This is the perfect starting point to generate test data quickly. A test data generator was implemented that could read ticker data from the database and store it into an XML file. The file format was defined by the Automated Test Environment. Later on such a file can be used repeatedly to test the Automated Test Environment itself by executing it with data parsed from the XML file. This contributes a lot to the development progress and stability of the Automated Test Environment itself.

During that phase the test department is taught how to use the Automated Test Environment. An executable version of the test data generator as well as the test executor is handed over to them. At first the tester enters data with an input client and saves the ticker id of that game. Using the test data generator an XML file can be generated for that ticker. This file is then used as a regression test each time a release has to be done for the application server. If new functionality has to be added, the XML file is modified by the tester and used again. At this time the test case generation and the test execution are automatized. The evaluation of the test result has to be done manually by observing the produced data in the Trader Client.

The advantage of shipping the first working version to the test department after a rather short development cycle is the collection of a lot of useful input to the development activities. Information about existing bugs in the Automated Test Environment and suggestions for usability improvement are reported. Shortly after the first release a second release is shipped that contains a test evaluation module. The results of the tests are written into a comma separated values (csv) file telling the tester if the test was successful (*OK*) or not (*ERROR*). There were some discussions of putting the test results into a specially designed database. The reason for the decision against a test result database has to do the big effort of implementing a database persistence layer and a database schema. Furthermore an existing database schema is much harder to change than the file format of a csv file and writing it into a csv file is much easier than writing into a database.

These are the steps a tester has to perform working with the Automated Test Environment at this time:

1. Create a ticker and enter events with a client a scout would use
2. Generate the test case xml file from the ticker's database data with the test case file generator
3. Start the Automated Test environment with the generated test case xml file

One disadvantage of shipping the Automated Test Environment at that early stage was the big number of test case XML files generated by the test department. Suddenly it was very difficult to keep track of the test case files.

8.1.1 Initial Architecture

The Automated Test Environment is a Maven Project that consists of three modules: Test Reference, Test Adapter and Test Framework. Each module is described in more details in this section.

- Test Reference
  - XML beans with the definition of the used XML format for test case files.
  - Data classes for event and ticker data
  - A class to de-serialize XML test case files
  - A class to load event data from the database and serialize it into XML test case files (Test Case Generator) (0)
  - Several utility classes used by the other modules
- Test Adapter

  - Output adapters: send data into the tested system
    - Old Scout Client (1)
    - New Scout Client based on Android with a different protocol (2)
    - Adapter to test the Feed Adapter which converts third party data into a format understood by the application server (3)
  - Input adapters: to receive data from the tested system
    - Trader Client Adapter (4)
    - Java Message Service (JMS) Adapter to test the integrated customer data supply (5)
  - XML beans for the adapter configuration files (server, port, username, password, adapter type)
  - Factory classes to load the input and output adapters

- Test Framework
  - Test system preparation
  - Test execution
  - Test evaluation
  - Test result generation

The numbers written next to the adapters can be found in Figure 8.1 to define the linkages into the tested system.

8.1.2 Test Case File Format

The representation of a sent and received event in a test case is implemented as an input-output-pair as described in Subsection 4.3.2. The simple test case to test for the automatic generation of the Dangerous Attack event can be seen in graphical form in Figure 8.2.

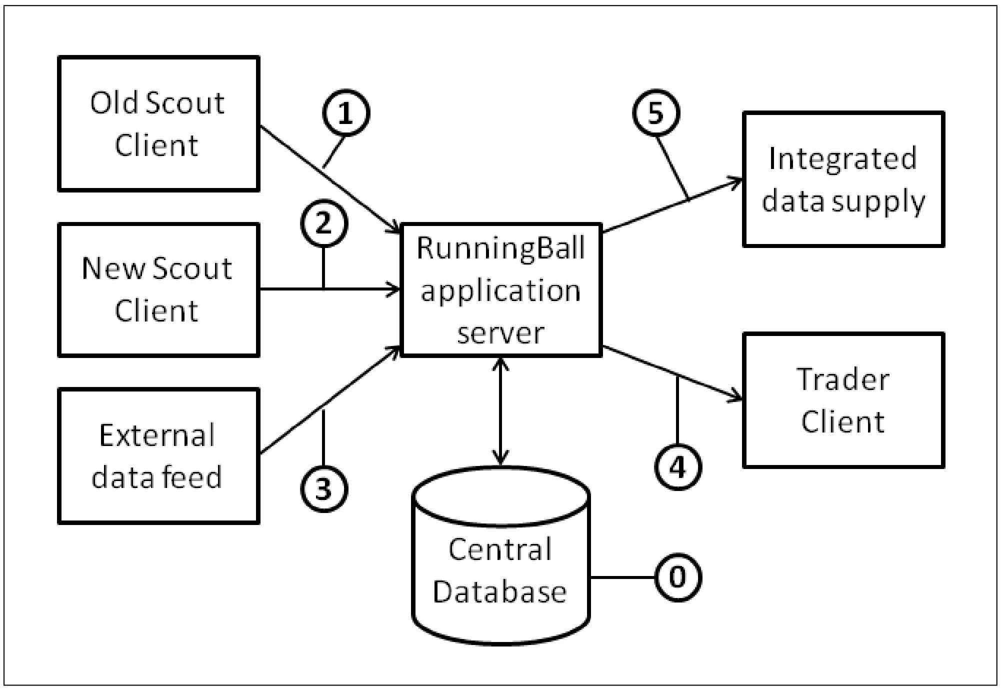


Figure 8.1: The Tested System with Adapter Linkages

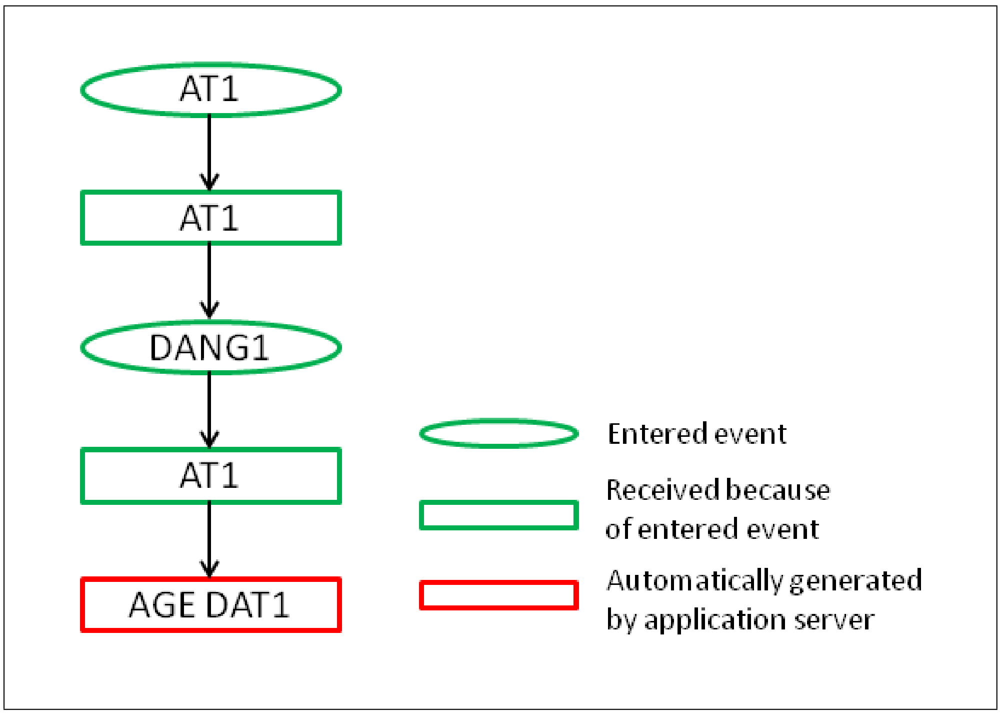

AT1
AT1
DANG1
AT1
AGE DAT1
Entered event
Received because of entered event
Automatically generated by application server

Figure 8.2: Test Case to test for Dangerous Attack

In an XML form that is understood by the Automated Test Environment this test case looks as seen in Listing 8.1.

```
<event_list>
  <event event_code_id="1024" event_code="AT1">
    <result type="default">
      <revent event_code_id="1024" event_code="AT1" />
    </result>
  </event>
  <event event_code_id="1052" event_code="DANGER1">
    <result type="default">
      <revent event_code_id="1052" event_code="DANGER1" />
      <revent event_code_id="1026" event_code="DAT1" generated="
          post" />
    </result>
  </event>
</event_list>
```

Listing 8.1: Test Case XML

The manual editing of XML files is error-prune and not very user-friendly. Besides that for a tester that edits such an XML test case it would be very annoying having to wait until the test run is finished to know that he has made a mistake in the XML file. That is why each XML test case is validated by the Automated Test Environment before starting the test run in three steps:

1. The Test for XML well-formedness ensures that the given test case is of correct and parse-able XML format.
2. The validation against an XSD [49] definition tests that it is of the correct XML schema.
3. For all important XML attributes there are predefined enumeration XML data types in the XSD schema.

8.1.3 Test System Preparation

There is quite a number of steps to be performed in the RunningBall database before a test can be started. Since the database of the tested system has to be restored with a live database backup quite regularly to keep up with the latest changes in the schema and the data it is necessary to perform some data preparation checks before each test run. Responsible for that is a utility class of the Automated Test Environment called TestSystemPrepareUtil. This class has to check and if necessary adapt the following data:

- The simulated consuming user groups
- The consuming user account
- The scout account with which data is sent into the system

- Country settings
- League settings: For example, in football some leagues are configured to have extra-time and a penalty shootout. They show a different event handling depending on the settings. Further league settings are the halftime duration, the starting score for darts, the number of periods for basketball or if a certain type of information is available for that league like player detail information.
- Team settings
- Stadium settings

8.1.4 Output Adapter Functionality

The main purpose of an output adapter is to hook into the tested system, use the commands provided by the Automated Test Environment, transform them into the protocol of the tested system interface and send data to it. Each output adapter has to implement an interface providing the following methods:

1. In the *Constructor* the output adapter is told the ticker id to which it should send data, the adapter configuration including server, port and user credentials, and the commands that should be sent.
2. The method *connect* connects the output adapter to the application server.
3. *login* performs a login to the tested server.
4. *openGame* with the correct ticker id tells the server that this adapter is going to send events to that ticker.
5. The method *sendEvent* can send all types of event data and its additional information. This includes for example for a football ticker the ball position coordinates for ball position events, the substituted players for a substitution event or in a darts game which segment was hit on the board and if it was a single, double or triple field.
6. *logout* is used to perform a logout.
7. *disconnect* interrupts the socket connection to the application server. This is used to simulate a disconnection that can last for different time spans to test the reaction of the application in case of a connection loss.
8. *shutdown* is used at the end of a test run and responsible to stop the output adapter in a clean way, namely ending all threads of the client and all started thread pools.

8.1.5 Input Adapter Functionality

In short words, an input adapter has to do the opposite of an output adapter. It has to connect to the tested system, receive data from it, transform it into the Automated Test Environment data format and pass this data on to the test evaluation. Besides event data an input adapter also has to handle ticker statistic data.

For the Trader Client testing statistics is a challenge. The Trader Client is implemented and optimized for fluent graphical visualization of the event and statistic data. It is not optimized to be tested easily. Whenever an event arrives, the event is passed to a GUI thread - this thread also calls the event reception method in the input adapter. The statistic of that event is passed to another GUI thread which also calls the statistic reception method in the input adapter. The Trader Client input adapter has to put together the statistic and event data because in the test data they are treated as connected too.

This is why a special logic has been created to be able to pass an event connected with its statistic from the Trader Client input adapter to the evaluation class. This procedure is only necessary for the Trader Client input adapter and not for the other adapters. When the event arrives before the statistic at the adapter the sequence shown in Figure 8.3 is used, otherwise the logic in Figure 8.4.

8.2 Test Data Rework

There are three important aspects that demand a rework of the test definition.

8.2.1 Separation of the Test Data from the Tested System

If the test data is generated using the logic of the tested system as has been done with the Automated Test System up to this point (see Section 8.1), all undetected errors of the system propagate into the test data. With that practice it can happen that some resulting test cases only serve to test the system if a system's error is still present. For that reason it is absolutely necessary not to use the tested system for the test case generation. In the first phase the only reason to use the tested system for test case generation was to obtain test data fast.

8.2.2 Organizational Change

It should be possible to set up an organizational and technological stripline between the development department and the test department. All test cases should be kept in a separate project that is completely the responsibility of

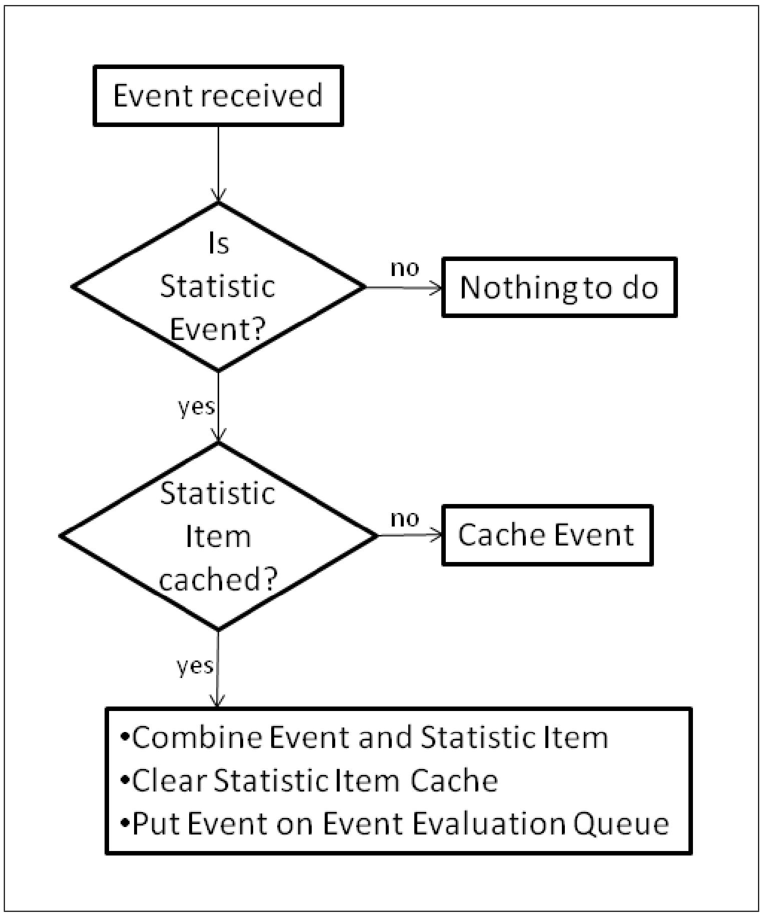

Event received
Is Statistic Event?
no
Nothing to do
yes
Statistic Item cached?
no
Cache Event
yes
•Combine Event and Statistic Item
•Clear Statistic Item Cache
•Put Event on Event Evaluation Queue

Figure 8.3: Event received before Statistic

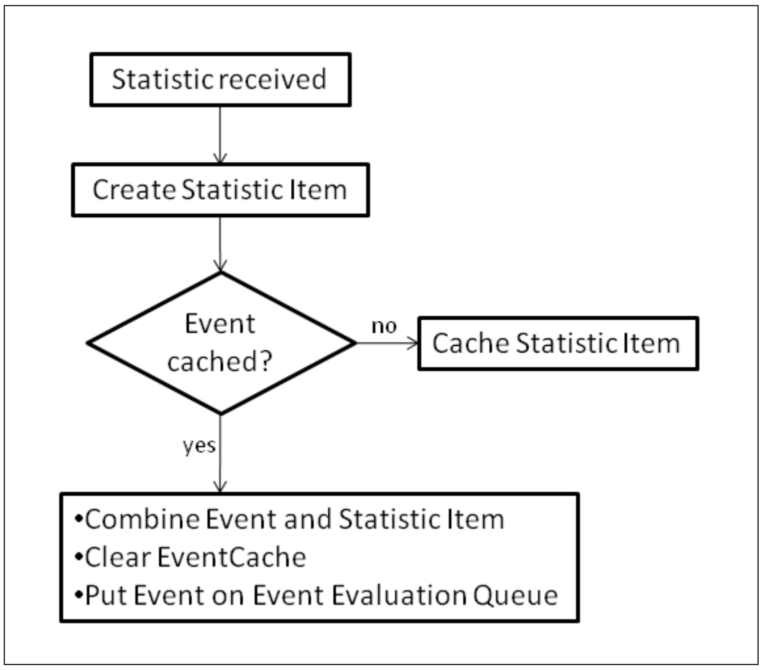


Figure 8.4: Statistic received before Event

the test department. There should be no direct link between the tested system and the test cases.

This task is solved by creating a new Maven project that contains only test case data XML files. This leads to the next advantage namely that the version of the test case data is decoupled from the version of the Automated Test Environment. The testers can decide by themselves when to perform a release of the test case project and increase the version number.

All the information that the Automated Test Environment needs, is where to find the test cases and which version to use. The test case artifact is loaded from a central repository during the execution of the Automated Test Environment using an URLClassLoader class. There is a Maven [43] plugin that can perform all the XML validations as described in Subsection 8.1.2 which ensures a comfortable process with test cases for the testers. Now we can see that a test case file is valid already after the build of the test case project without starting a test run.

Listing 8.2 shows a way how the Java classpath can be extended to a resource that is situated at a remote location using an URLClassLoader class.

```java
public static void addURLToPath(String url) throws
      MalformedURLException, NoSuchMethodException,
      IllegalAccessException, InvocationTargetException {
  URL u = new URL(url);
  URLClassLoader urlClassLoader = (URLClassLoader) ClassLoader.
      getSystemClassLoader();
  Class urlClass = URLClassLoader.class;
  Method method = urlClass.getDeclaredMethod("addURL", new Class[]
      { URL.class });
  method.setAccessible(true);
  method.invoke(urlClassLoader, new Object[] { u });
}
```

Listing 8.2: Classpath Extension with an URLCLassLoader

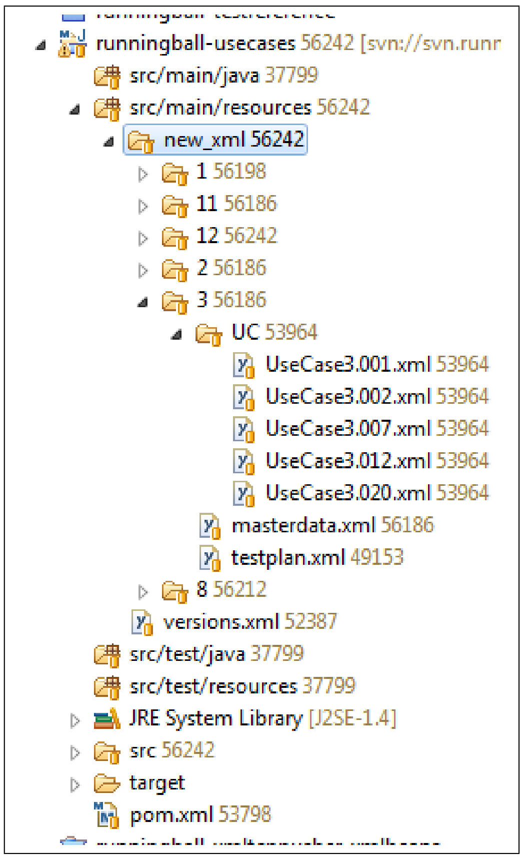
8.2.3 Introduction of a new Test Case File Structure

A further emerging organizational problem should be solved in this step. Since the testers are very busy they generate a huge number of test case files within a short time. The major problem is test case maintainability. A lot of test data is kept redundantly, and therefore changes are expensive and annoying. If short sequences of events that often occur in the same order can be isolated and put into a reusable so-called UseCase file this situation would improve dramatically. To do this, use cases and more importantly use case *ids* have to be created. There should be a new test case file structure:

- A file type to define all test preparation settings
- Another file type to contain event sequences for one use case
- A file type called test plan that combines test preparation and use case files.

The test plan file links to use cases in a defined order via their id. Since the event code ids differ for the home or the away team of a ticker it would be nice if one use case file could be used for events of both teams. It should be possible to pass the team number from the test plan file to the use case file and convert the correct event code ids depending on the team number.

The newly created Maven project for test data looks as shown in Figure 8.5. On the top part there are folders named after the id of the sport the test data is for. In them there are test preparation files, test plans and a folder containing all use case files named UC.



```
runningball-usecases 56242 [svn://svn.runr
    src/main/java 37799
    src/main/resources 56242
        new_xml 56242
            1 56198
            11 56186
            12 56242
            2 56186
            3 56186
                UC 53964
                    UseCase3.001.xml 53964
                    UseCase3.002.xml 53964
                    UseCase3.007.xml 53964
                    UseCase3.012.xml 53964
                    UseCase3.020.xml 53964
                masterdata.xml 56186
                testplan.xml 49153
            8 56212
        versions.xml 52387
    src/test/java 37799
    src/test/resources 37799
    JRE System Library [J2SE-1.4]
    src 56242
    target
    pom.xml 53798
```

Figure 8.5: Test Data Maven Project Structure

In Listing 8.3 an extract of the use case project's pom.xml is shown. The xml-maven-plugin is responsible to validate all found XML files against the provided XSD definition.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>xml-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <validationSets>
      <!-- check all files in folder xml for well formedness -->
      <validationSet>
        <dir>src/main/resources/new_xml</dir>
      </validationSet>
      <!-- validate all files in folder src/main/new_xml against
          src/main/xsd/test_data.xsd -->
      <validationSet>
        <dir>src/main/resources/new_xml</dir>
        <systemId>src/main/xsd/test_data.xsd</systemId>
      </validationSet>
    </validationSets>
  </configuration>
</plugin>
```

Listing 8.3: Use Case Project pom.xml Extract

Listing 8.4 shows a very short test plan. An important instruction is found in line four, where the required preparation data is defined. Afterwards the commands for connect, login and start match are entered. After only one dangerous attack for each team the ticker is stopped.

```
 1 <test_data sport_id="1"
 2   xmlns="http://rball.com/testreference/data/xmlbeans">
 3   <test_plan>
 4     <preparations path="new_xml/1/masterdata.xml" />
 5     <tickerstate_list>
 6       <tickerstate id="1" name="Not Started">
 7         <usecase path="new_xml/1/UC/UseCase0.020.xml" name="
               ConnectLogin"
 8           group="Scout" />
 9         <usecase path="new_xml/1/UC/UseCase1.002.xml" name="
               StartGameTeam2"
10           group="Scout" />
11       </tickerstate>
12       <tickerstate id="2" name="Rt First Half">
13         <usecase path="new_xml/1/UC/UseCase1.010.xml" name="
               DangerousAttack"
14           team="1" group="Scout" />
15         <usecase path="new_xml/1/UC/UseCase1.010.xml" name="
               DangerousAttack"
16           team="2" group="Scout" />
17         <usecase path="new_xml/1/UC/UseCase1.003.xml" name="
               StopFirstHalf"
18           group="Scout" />
19       </tickerstate>
20       <tickerstate id="4" name="Rt Pause">
21         <usecase path="new_xml/1/UC/UseCase1.004.xml" name="
               StartSecondTeam1"
22           group="Scout" />
23       </tickerstate>
24       <tickerstate id="8" name="Rt Second Half">
25         <usecase path="new_xml/1/UC/UseCase1.006.xml" name="
               StopSecondHalf"
26           group="Scout" />
27       </tickerstate>
28     </tickerstate_list>
29   </test_plan>
30 </test_data>
```

Listing 8.4: Simple Testplan Example

The test case testing for dangerous attack as described in Subsection 8.1.2 had to be changed only slightly to fulfill the new format as seen in Listing 8.5. This made the conversion and reorganization of the existing test data files not too time-consuming.

```
<test_data sport_id="1"
  xmlns="http://rball.com/testreference/data/xmlbeans">
  <usecase id="1.010">
    <event_list>
      <event sent_by_user="$P{User}" event_code_id="1024"
          event_code="AT1">
        <result type="default">
          <revent event_code_id="1024" event_code="AT1"
            affected_statistics="2048=0 1024=+1" />
        </result>
      </event>
      <event sent_by_user="$P{User}" event_code_id="1052"
          event_code="DANGER1">
        <result type="default">
          <revent event_code_id="1052" event_code="DANGER1" />
          <revent generated="post" event_code_id="1026" event_code=
              "DAT1"
            affected_statistics="1026=+1 2050=0"
                ticker_danger_state="true" />
        </result>
      </event>
    </event_list>
  </usecase>
</test_data>
```

Listing 8.5: Dangerous Attack in the new XML Format

It should be possible to put the same use case at any position in the test plan. That means that absolute statistic (see Section 7.2) values cannot be used. Relative statistical values are supplied instead embodying only the deviation of the statistic value compared to the preceding event.

Furthermore there are no event numbers used to make use cases independent of their test plan position. If we write a use case that should delete an event with a certain number, we define the information which event should be cleared as “delete the x^{th} last event that has the event code y”. Again by doing this we can place the same clear use case at various positions in the test plan.

The Automated Test Environment takes the challenge of transforming all relatively defined information to absolute values which can only be determined during a test run.

At this stage the tester has to perform these steps to operate the Automated Test Environment:

1. Write use case xml files and link them in a test plan file

2. Commit the written files into the svn repository and build the use case project

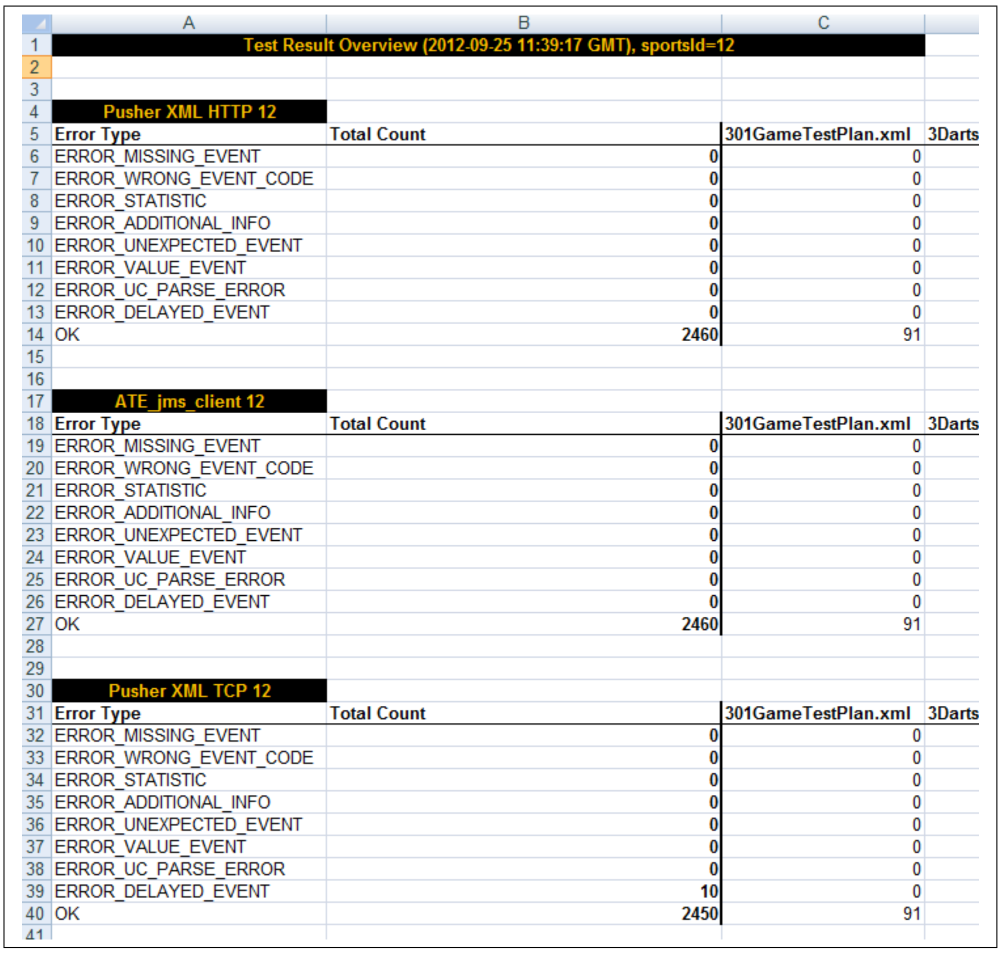
3. Configure the Automated Test Environment to use the written test plans and start it

8.2.4 Test Result Categories

It soon became obvious, that the differentiation of ERROR and OK is not sufficient for a detailed test result analysis. Besides that, in case of an error a more detailed test result makes it easier to track down the error. This is why the following eight erroneous and one correct test result categories are introduced:

- ERROR_MISSING_EVENT: An event that was expected has not been provided by the input adapter
- ERROR_UNEXPECTED_EVENT: An event has been received at the input adapter that was not expected
- ERROR_WRONG_EVENT_CODE: The event has the wrong event code
- ERROR_STATISTIC: The statistic connected to an event is wrong
- ERROR_ADDITIONAL_INFO: Additional event information such as the substituted player numbers or the exact game conditions are wrong; category to collect all errors that do not match into any other erroneous category
- ERROR_VALUE_EVENT: One or more value events of an event is wrong
- ERROR_UC_PARSE_ERROR: Logical parse error of a test case XML file that cannot be caught by the three step XML parse mechanism explained in Subsection 8.1.2; an example is a clear event applied to an event that has not been sent before
- ERROR_DELAYED_EVENT: An event has taken too long to be received
- OK: The event is correct

There is a test result summary Excel file displaying a summarized table of the test outcome and one sheet for each test plan. In case of errors there is a clickable file link to the detailed .csv evaluation file per input adapter and testplan. An example of such a test summary Excel file can be seen in Figure 8.6.



	A	B	C	
1	Test Result Overview (2012-09-25 11:39:17 GMT), sportsId=12			
2				
3				
4	Pusher XML HTTP 12			
5	Error Type	Total Count	301GameTestPlan.xml	3Darts
6	ERROR_MISSING_EVENT	0	0	
7	ERROR_WRONG_EVENT_CODE	0	0	
8	ERROR_STATISTIC	0	0	
9	ERROR_ADDITIONAL_INFO	0	0	
10	ERROR_UNEXPECTED_EVENT	0	0	
11	ERROR_VALUE_EVENT	0	0	
12	ERROR_UC_PARSE_ERROR	0	0	
13	ERROR_DELAYED_EVENT	0	0	
14	OK	2460	91	
15				
16				
17	ATE_jms_client 12			
18	Error Type	Total Count	301GameTestPlan.xml	3Darts
19	ERROR_MISSING_EVENT	0	0	
20	ERROR_WRONG_EVENT_CODE	0	0	
21	ERROR_STATISTIC	0	0	
22	ERROR_ADDITIONAL_INFO	0	0	
23	ERROR_UNEXPECTED_EVENT	0	0	
24	ERROR_VALUE_EVENT	0	0	
25	ERROR_UC_PARSE_ERROR	0	0	
26	ERROR_DELAYED_EVENT	0	0	
27	OK	2460	91	
28				
29				
30	Pusher XML TCP 12			
31	Error Type	Total Count	301GameTestPlan.xml	3Darts
32	ERROR_MISSING_EVENT	0	0	
33	ERROR_WRONG_EVENT_CODE	0	0	
34	ERROR_STATISTIC	0	0	
35	ERROR_ADDITIONAL_INFO	0	0	
36	ERROR_UNEXPECTED_EVENT	0	0	
37	ERROR_VALUE_EVENT	0	0	
38	ERROR_UC_PARSE_ERROR	0	0	
39	ERROR_DELAYED_EVENT	10	0	
40	OK	2450	91	
41				

Figure 8.6: Test Result Overview

8.3 Automated Test Environment used as JUnit Test

A new application field arises for the Automated Test Environment for three reasons:

- A project in the company that enables the user to create an in-memory H2 [17] database for a unit test run that is completely independent of a real database has just been finished. Using this project it is possible to access and manipulate a database with unit tests. This database only exists in the memory of the machine where the build is executed and embodies the entire database schema of the used live database.

- The RunningBall Application Server contains all its event processing logic in one class per sport called EventQueue.

- The test data is now a separate project at a central location that can be accessed easily.

The idea is to create a version of the Automated Test Environment that is executed as a unit test. Therefore a test runner, an output adapter that instantiates the tested EventQueue class and an input adapter that transforms the event data obtained from the EventQueue class into a correct format for test evaluation is needed. Many components such as the test system preparation and the evaluation module of the Automated Test Environment can be reused.

Normally JUnit recognizes test classes if they have the string *Test* at the end of their name. This is done automatically by the JUnit class runner. Besides that this class runner searches test classes for methods that are annotated with *@Test*. This mechanism is used to tell JUnit how to execute the Automated Test Environment and furthermore to interpret each sent event as a separate test method. This is an advantage for the test evaluation because we can see the result of each event represented as one JUnit test method.

As seen in Figure 8.7 we tell JUnit to execute the class *ATEEventQueueTest* with our own implementation of the JUnit class runner named *FactoryRunner*.

The class *FactoryRunner* overrides the method *computeTests()* which performs three important steps:

1. Firstly tell JUnit that the expected test methods are of type *FrameworkTestFactory*, where the name of the method is specified with the method *getName()*. This name is then listed in the JUnit test result. So we put information the about the tested event into that name.

2. Ensure that JUnit searches for all methods annotated with *@TestFactory* to be executed during a test run.

3. The last step is to tell JUnit that it finds all methods where the actual test assertions are executed with the annotation *@FactoryTest*. In this method the functionality of the test evaluation utility class of the normal Automated Test Environment is used to test the expected against the actual event and categorize the result.

When JUnit is executed, the entry point is the method *tests()* in the class *ATEEventQueueTest*. At first all test preparations are done. This means instantiating the memory database, creating the adapters, the tested application server event queues and creating all required master data including the tickers. After that the test is executed. The event processing method in the *EventQueue* class is called by the *AppServerUnitTestOutputAdapter*. The resulting events are then decoded by the *AppServerUnitTestInputAdapter*.

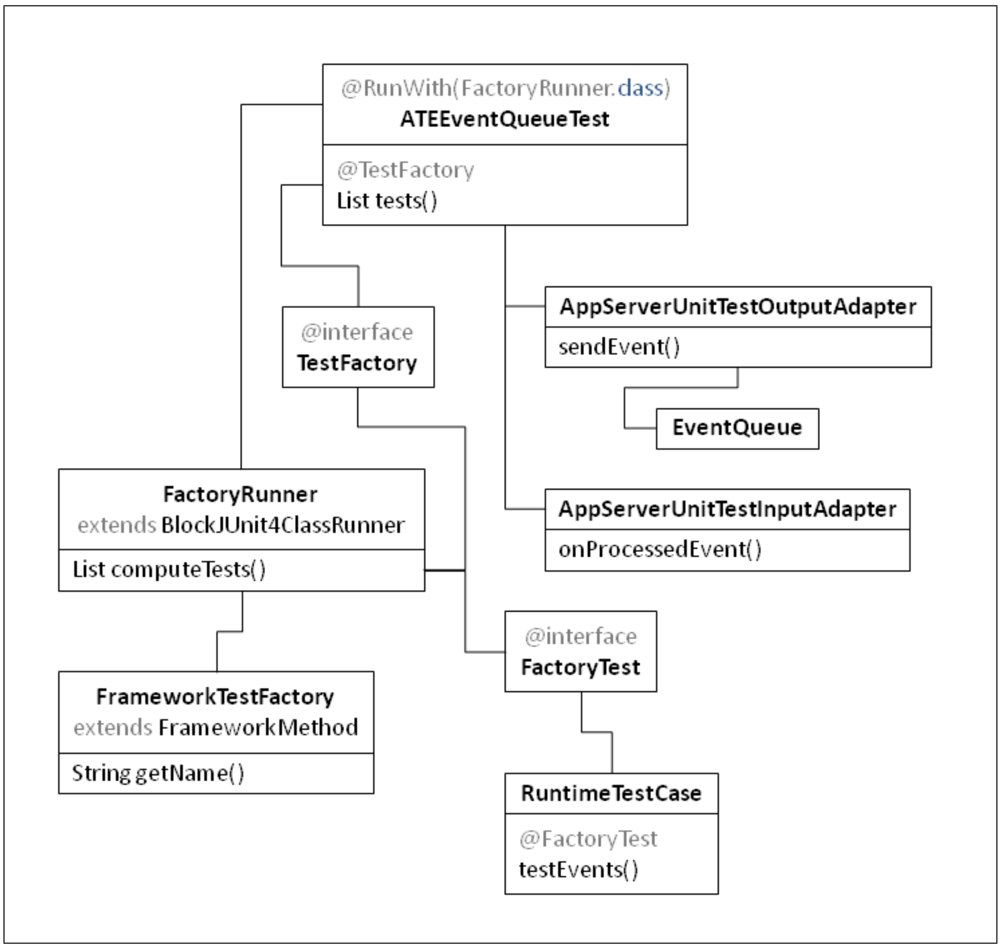


Figure 8.7: Automated Test Environment executed as JUnit Test

Besides that the input adapter creates a *RuntimeTestCase* object for each received event. The expected and actual events are put into the *RuntimeTestCase* object. All *RuntimeTestCase* objects are stored in a list. After all events have been sent through the event queue, the method *tests()* in *ATEEventQueueTest* returns the list of *RuntimeTestCase* objects to JUnit.

JUnit finally goes through all *RuntimeTestCase* objects, executes the *testEvents()* method and generates the JUnit test result. This test result is in XML format and can be displayed in the end by Jenkins (Section 5.3) or Sonar (Section 5.5). Eclipse has a nice JUnit test result view as well as seen in Figure 8.8. The test method name is composed of the test plan file name, the test plan line number, the use case id and a short description.

With Unit tests it is easily possible to measure code coverage and the progress of code coverage.

8.4 Nightly System Test

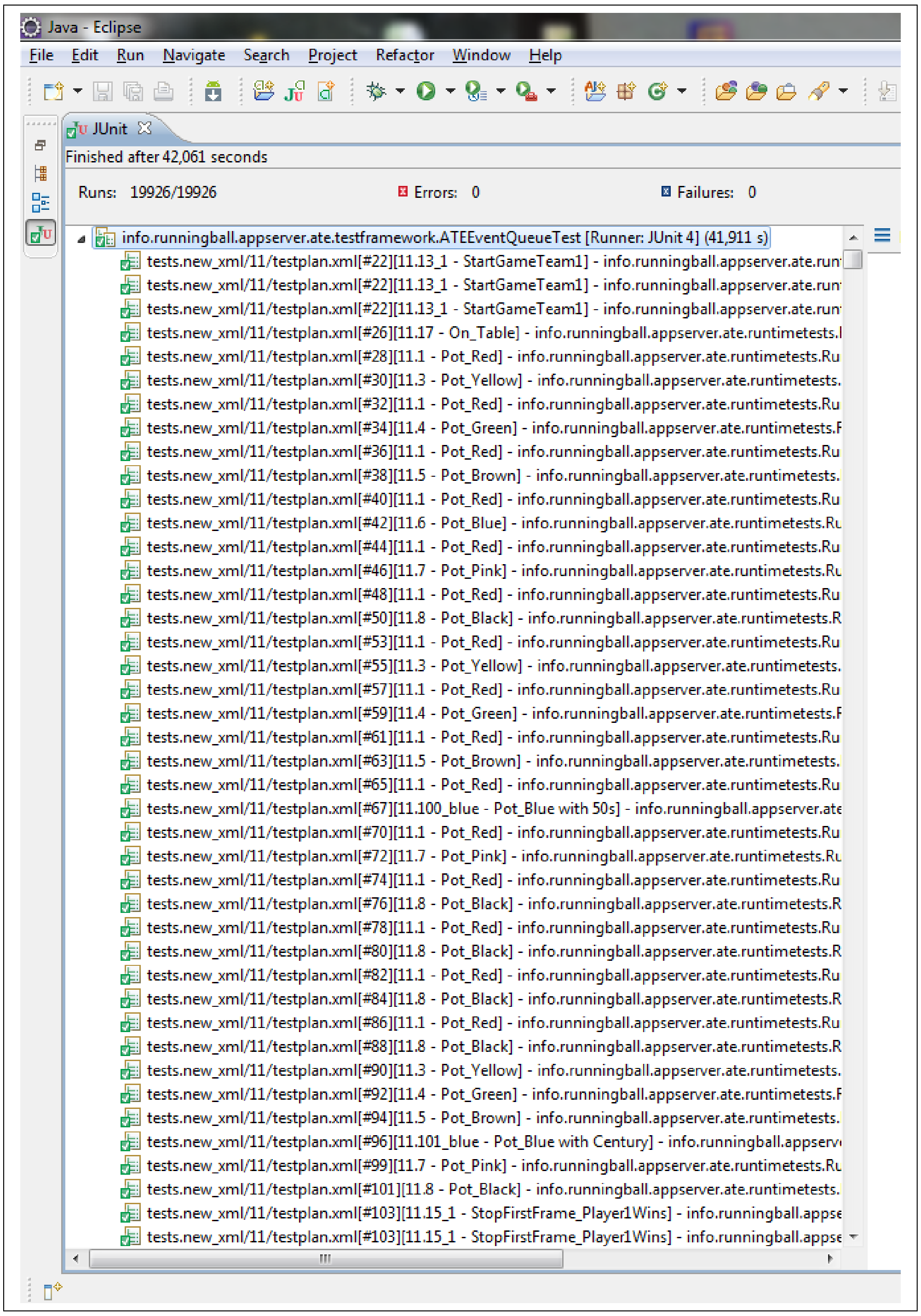
After the JUnit test the need for another type of test arises where the system of RunningBall applications is tested. Jenkins is used for the test execution because it is easy to configure and provides the perfect functionality for an automated periodic test runner. There is one Jenkins job for each sport. The job is started with the following arguments telling Jenkins to start the Automated Test Environment in the nightly system test mode for the correct sport:

```
-Patetest verify -DargLine="-Date_sportsid=1"
```

Additionally to the required build configuration settings the pom.xml of the Automated Test Environment contains a separate profile defining the goals for the nightly system test as seen in Listing 8.6. When Maven is started with the profile named *atetest* it is told to execute the integration-test target.

```
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.12</version>
  <groupId>org.apache.maven.plugins</groupId>
  <configuration>
    <sportsid>1</sportsid>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Listing 8.6: Maven Plugin Configuration for the Nightly Test



```
Finished after 42,061 seconds

Runs:  19926/19926          Errors:  0          Failures:  0

info.runningball.appserver.ate.testframework.ATEEventQueueTest [Runner: JUnit 4] (41,911 s)
    tests.new_xml/11/testplan.xml[#22][11.13_1 - StartGameTeam1] - info.runningball.appserver.ate.run
    tests.new_xml/11/testplan.xml[#22][11.13_1 - StartGameTeam1] - info.runningball.appserver.ate.run
    tests.new_xml/11/testplan.xml[#22][11.13_1 - StartGameTeam1] - info.runningball.appserver.ate.run
    tests.new_xml/11/testplan.xml[#26][11.17 - On_Table] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#28][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#30][11.3 - Pot_Yellow] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#32][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#34][11.4 - Pot_Green] - info.runningball.appserver.ate.runtimetests.F
    tests.new_xml/11/testplan.xml[#36][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#38][11.5 - Pot_Brown] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#40][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#42][11.6 - Pot_Blue] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#44][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#46][11.7 - Pot_Pink] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#48][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#50][11.8 - Pot_Black] - info.runningball.appserver.ate.runtimetests.R
    tests.new_xml/11/testplan.xml[#53][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#55][11.3 - Pot_Yellow] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#57][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#59][11.4 - Pot_Green] - info.runningball.appserver.ate.runtimetests.F
    tests.new_xml/11/testplan.xml[#61][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#63][11.5 - Pot_Brown] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#65][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#67][11.100_blue - Pot_Blue with 50s] - info.runningball.appserver.ate
    tests.new_xml/11/testplan.xml[#70][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#72][11.7 - Pot_Pink] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#74][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#76][11.8 - Pot_Black] - info.runningball.appserver.ate.runtimetests.R
    tests.new_xml/11/testplan.xml[#78][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#80][11.8 - Pot_Black] - info.runningball.appserver.ate.runtimetests.R
    tests.new_xml/11/testplan.xml[#82][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#84][11.8 - Pot_Black] - info.runningball.appserver.ate.runtimetests.R
    tests.new_xml/11/testplan.xml[#86][11.1 - Pot_Red] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#88][11.8 - Pot_Black] - info.runningball.appserver.ate.runtimetests.R
    tests.new_xml/11/testplan.xml[#90][11.3 - Pot_Yellow] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#92][11.4 - Pot_Green] - info.runningball.appserver.ate.runtimetests.F
    tests.new_xml/11/testplan.xml[#94][11.5 - Pot_Brown] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#96][11.101_blue - Pot_Blue with Century] - info.runningball.appserv
    tests.new_xml/11/testplan.xml[#99][11.7 - Pot_Pink] - info.runningball.appserver.ate.runtimetests.Ru
    tests.new_xml/11/testplan.xml[#101][11.8 - Pot_Black] - info.runningball.appserver.ate.runtimetests.
    tests.new_xml/11/testplan.xml[#103][11.15_1 - StopFirstFrame_Player1Wins] - info.runningball.appse
    tests.new_xml/11/testplan.xml[#103][11.15_1 - StopFirstFrame_Player1Wins] - info.runningball.appse
```

Figure 8.8: JUnit Test Result in Eclipse

Maven executed with the integration-test target looks in the test source folder of the Automated Test Environment for classes with a name starting with *IT*. This is why we implement a class named *ITATE*. This class is very short, containing only one method that is annotated with *@Test*. In this method the Automated Test Environment is started with the sport id provided by the Maven command line argument and told to write a JUnit-like test evaluation additionally to the normal evaluations.

For each test plan one JUnit XML evaluation file is written. The JUnit result is then displayed by Jenkins in the test progress chart (as seen in Figure 5.6). Furthermore test errors are displayed and when clicked on, the details of the errors are shown. If the JUnit test evaluation is not sufficient there is still the possibility to view the test result files generated by the Automated Test Environment as described in Subsection 8.2.4.

Chapter 9

Project Results

In this chapter an outline of the project's results is provided.

9.1 Quantifiable Results

9.1.1 Line Coverage Progress

The first measurable metric is the line coverage that results from the event queue JUnit tests as explained in Section 8.3. For a better visualization in Figure 9.1 only some sport types have been chosen. After a fast initial rise that is related to the fast implementation of new test cases the further gain in coverage is caused by the implementation of new test methods in the Automated Test Environment. The coverage of some sports has dropped recently because new features have been implemented in the tested product, the test cases are not written for all of them yet.

9.1.2 Number of tested Events

Another interesting number is the number of events that are sent through the system during a complete test run. This number gives a good estimate of what support test automation brings to the test department. If all this data would be entered by hand, each event represents at least one mouse click by the tester. Table 9.1 lists the number of events per sport type. Since the test data is the same for JUnit tests and the automated nightly system test the numbers apply for both types of the Automated Test Environment.

9.1.3 Nightly Test Progress

The test progress diagram of Jenkins (see Section 5.3) is a good way to illustrate the progress of the nightly automated tests. Figure 9.2 shows the test outcome of some recent test runs. The blue area represents the total number of tests executed, the small red areas represent failed tests.

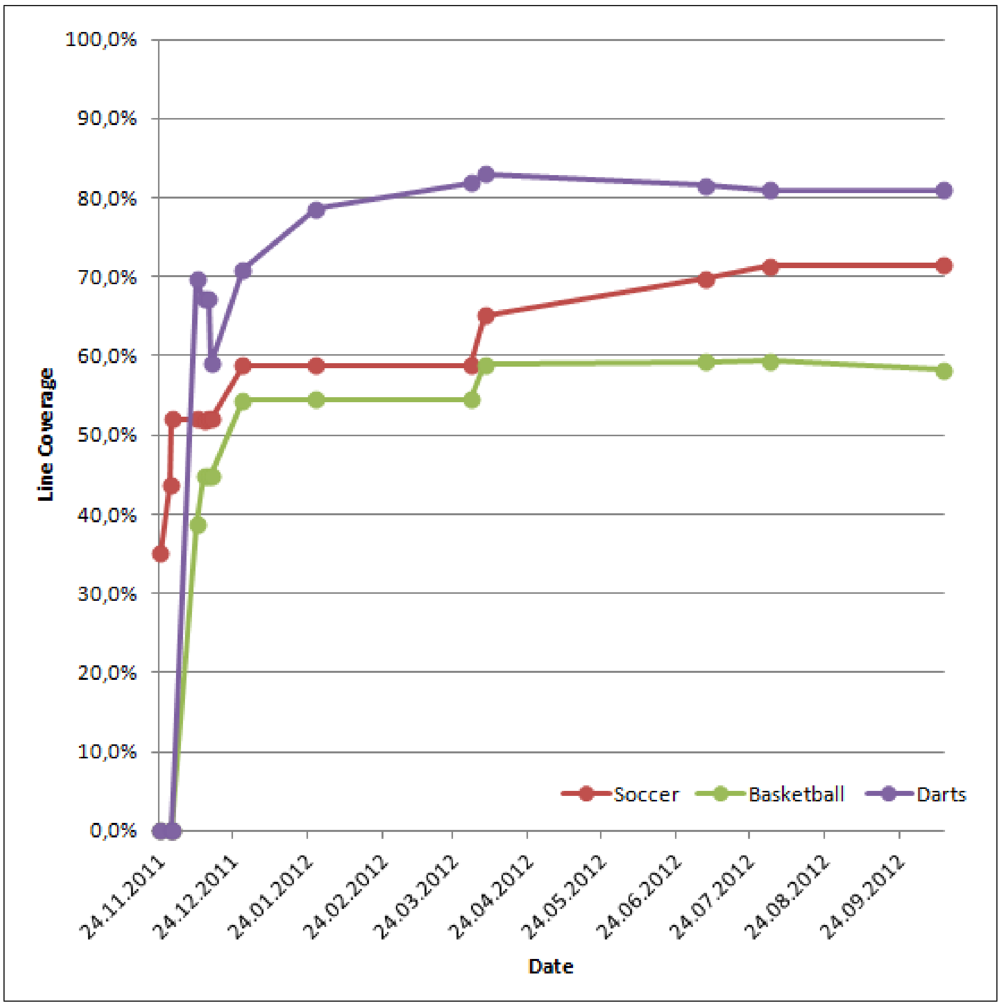


Figure 9.1: JUnit Line Coverage over Time

Table 9.1: Number of tested Events per Sport Type

Sport Type	**Number of Events**
Soccer	2921
Basketball	8879
Tennis	558
Ice hockey	4910
Snooker	239
Darts	2419
Total	**19926**

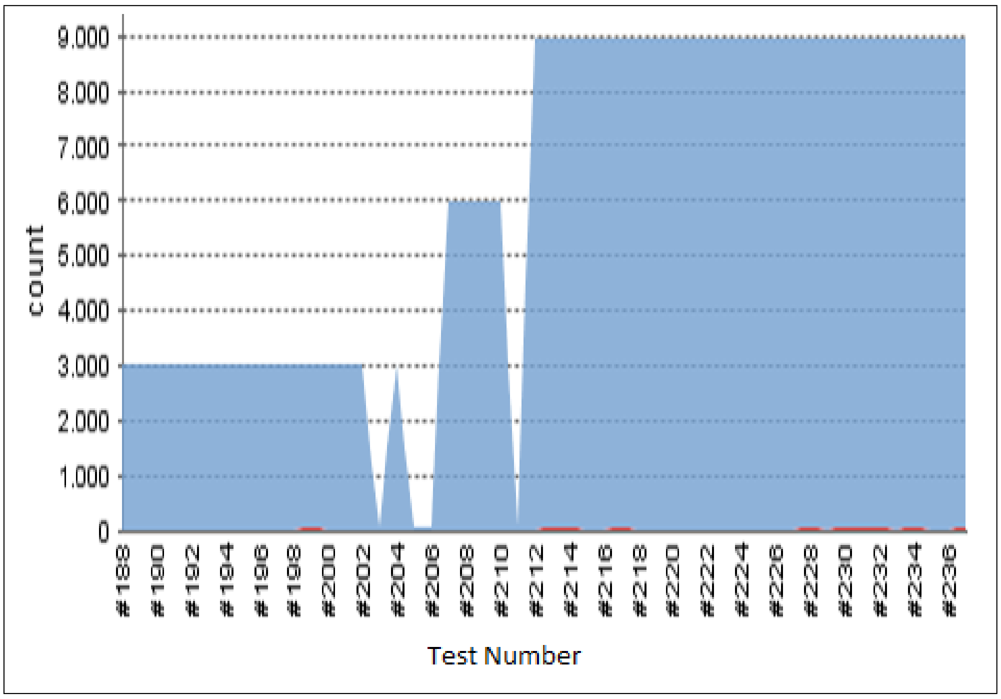


Figure 9.2: Nightly Test Progress

9.2 Changes in the Development Process

It took a while until the Automated Test Environment was powerful and stable enough to be used as an everyday development support tool. An underestimated factor was that people working in the development and test team had to be brought together to use the new testing tool. This meant breaking with their familiar working routines and introducing them to something new. It is in human nature to react skeptical when told about something so revolutionary. After breaking the ice many of the people involved understood the benefits of the new working techniques and began to like the Automated Test Environment.

Whenever an error is reported that relates to a feature that is testable with the Automated Test Environment the event logic that caused the error is put into a new use case and a test is executed. The error is fixed and then the test is performed again. In doing so a large collection of regression tests has been created.

Architectural changes in the application server project can now be done easier and with more confidence. The developer can rely on the detailed tests. The developer has more courage when a refactoring has to be done because of the good test coverage.

Last but not least the quality of the application server has improved. The large collection of use cases serves as a detailed knowledge base where all event sequence logic is stored in a readable manner. People developing the event sequence logic are now forced to put the logic into a defined language. Discrepancies and side effects of new event sequences can be seen at an earlier development stage now. The event sequence logic is growing fast with every new sport the company implements. It is not only up to humans to verify the correct event logic anymore.

9.3 Feedback from People Affected

I have asked some people affected by the Automated Test Environment project for their feedback and summarize it in the following subsections.

9.3.1 Software Architect

> “Using the Automated Test Environment it is now possible to write automatized tests in the back end. After the first tests were executed some errors were found that have existed in the software for a long time but have not been found with manual testing. With the test definition a repository was created which enables the developer to perform tests already during the development. In doing so the developer can avoid that his changes have any unforeseen side effects. The quality regarding requirements has improved a lot because the implemented requirements can be verified easily. The project state can be seen looking at the successful test cases compared to the unsuccessful.
>
> The communication between testers and developers has changed in a way that they do not talk about requirements but about test cases. Based on the requirements the testers create test cases which are now the foundation for the development. The collaboration has improved because the number of misunderstood requirements has decreased. If something is not clear the test case is reviewed by the testers and developers, and if necessary the test case is adapted.
>
> The Automated Test Environment executed as JUnit test gives the developer the possibility to execute tests fast during each development phase. Besides that the unit tests are executed with every nightly build. With that possible negative side effects are seen quite soon. Code coverage has become a measurable metric.
>
> With the nightly system test all test cases are executed as well. Defects are detected fast and one can react on them in an early development phase.
>
> The architecture of the Automated Test Environment can be described as extensible because many design patterns were im-

plemented. Important components have been abstracted with interfaces and therefore they can be exchanged easily. Using the adapter pattern in the Automated Test Environment it is ensured that it can be used for the testing of further products and components.

Because the test cases reflect a huge part of our system's logic it is obvious that this knowledge base will be used in further environments. My ideas for the future usage are the creation of test case documentations and an extension to be able to test user interfaces such as our Android client." [translated by the author]

9.3.2 Project Manager

"The progress of the project was satisfying. The biggest struggle was to prioritize the Automated Test Environment to a level high enough that suited its importance and sustainability. The expectations that were put into that project were fully met from the project management's perspective.

Tasks and work packages were designed in cooperation with the development and the test team. Feedback from the test department was considered and implemented according to their vision. Furthermore the after project phase was started to ensure the continuous integration of the development within our software systems.

Regression testing has been facilitated and become much faster. Bug fixes could be verified quicker and the developers had the benefit of an immediate feedback if their code was correct. Reproducible tests and automatized test evaluations led to a big leap forward of our software quality. In the future, continuous testing should contribute to a faster detection of errors in the development phase. With the automation of tests our test team gained more time for manual testing of other applications which of course has a positive effect on the quality of these applications.

For the project management it is much easier now to estimate the current development and test status. An estimate can be given if the project is within schedule or more resources need to be added to meet a deadline." [translated by the author]

9.3.3 Tester

"Before the introduction of the Automated Test Environment we had a lot more effort to secure that existing functionality still worked than effort that could be put into verifying new features. With the test automation the development iterations grew a lot in size and complexity because more tests could be done in a short time, and the developers obtained much faster feedback from the tests. The product management department had an unstructured knowledge base containing the event logic and normally detailed discussions were only made if there were problems in the logic. With the test automation though the knowledge base had to be created before the implementation.

My expectations for the Automated Test Environment were primarily the automation of regression tests. Furthermore I expected that less untested products would come from the development into the test department because of the early JUnit tests. The most important expectations were:

- Reproducibility of tests
- Error categories
- Fail Fast criteria
- Reliability and robustness of the test application itself
- Perform tests in an environment as real as possible

Deviations from the definition of features can now be detected earlier. The effort to define new features has increased on the short-term but pays off on the long-term. The biggest challenge was to verify new features in the Automated Test Environment itself because it has the requirement to be absolutely correct.

A difficulty was that the Automated Test Environment was underestimated as a source of error itself in the beginning. These errors could be fixed and the Automated Test Environment is reliable enough that now there are products that are tested exclusively using automated tests." [translated by the author]

9.3.4 Application Server Developer

"Before the usage of the Automated Test Environment there were only a few unit tests available in the application server project. There were no unit tests that covered functionality, that was used for more than one sport type. Developers performed some tests restricted to only one application manually and only on demand. Testers performed their tests manually as well. In the beginning I was not thrilled about the introduction of a test automation tool. That was because I assumed that it would only be a set of unit tests. At the moment I am even more pleased

> by the Automated Test Environment because it enables us to implement entire work flows.
>
> For sure the quality of the application server was increased because a huge number of tests can be performed with little effort and in short time. Besides that, tests have become reproducible.
>
> As mentioned before, the now implemented solution is quite different to the test automation that I expected. Regarding the tested product and the extreme detail of the tests my expectations have been exceeded." [translated by the author]

9.4 Relation to the Theoretical Part

At the time the theoretical part of this thesis was written, it was not clear which facts will really be needed in the practical part. Therefore this section gives an outline of the really implemented knowledge.

9.4.1 Software Test

The three basic steps of testing, test preparation, execution, and evaluation can be found in the corresponding modules of the Automated Test Environment. There are two scopes that were implemented: a unit test with the JUnit mode and a system test with the nightly system test mode.

The chosen test method is functional testing as described in Subsection 2.2.1 and to be said in more detail the test for correctness. The other described test methods do not suit the requirements of the Automated Test Environment as well as the test for correctness.

9.4.2 Used Metrics

There is a vast number of available metrics. In my opinion the Automated Test Environment should be as simple as possible and focused on its purpose. That means choosing and implementing only as many metrics as absolutely necessary. Used metrics are:

- The test progress curve provided by Jenkins as seen in Figure 9.2
- The complexity, test coverage and Sonar rules compliance seen in Figure 5.8. These metrics are used for both the Automated Test Environment and the application server project.

Metrics that can be deducted if they are needed without much effort out of the existing environment:

- Since all found bugs are reported in a bug-tracking system the Time Elapsed of found Errors can be determined.

- Many other measurements such as Lines of Code or Cyclomatic Complexity can be displayed using Sonar.

9.4.3 Test Automation

Nearly all of the information presented in Chapter 4 about test automation is implemented in the Automated Test Environment. A formal language for test data description was developed and is used. Jenkins is used to automatically execute both test types, the JUnit nightly build-attached test and the nightly system test. The practice of continuous build could be implemented as well.

The development strategy of the Automated Test Environment can be described as *Bottom-up*. In the first step only the three modules Test Reference, Test Adapter and Test Framework were defined. After that the connection points between them and the test data definition was made. Finally the details of each module were implemented.

The chosen test strategy is testing for functionality, and initially capture and replay was used to gather test data quickly. From my daily work I can tell that the test end criteria is either team consensus or a management directive to ship a product.

9.5 Future Issues of Scale

This section deals with the future challenges of the Automated Test Environment. Two perspectives are explained: What happens when the number of sports increases, and what happens if the number of developers and testers rises for example above ten.

9.5.1 More Sports means more Test Cases

More sports means no big issue for the JUnit test mode of the Automated Test Environment. At the moment all six sports that are in production are tested verifying more than 20.000 events with a test execution time of approximately five minutes. This test is performed at least once a day during the nightly build of the application server. If the number of sports rises the number of test cases and the test execution time will rise as well. Besides that it has to be ensured that the build process has enough memory available to test more events. I expect the test execution time to rise a little above straight proportional to the number of test cases because this was the case whenever a set of test cases was added for a new sport.

For the nightly system test more sports will lead to bottlenecks rather soon. The first problem is that the hardware where the tests are executed will soon be at its limits. Secondly it is not possible to run a test with multiple sports at once. So each sport has to be tested individually. This limitation mainly refers to some input adapters that can only handle one sport at a time. This issue can be categorized as a testability problem of the tested components. With the current configuration it will become difficult to schedule tests for more than ten sports each night.

9.5.2 More Developers and Testers

At the moment one to two developers are contributing to the application server project concurrently. It is easy to manage the division of work and to avoid a broken build because of test failures. But if the number of developers rose above five many problems will occur. The first will be the source code management system Subversion [44] because it does not allow to commit into a local repository. This situation would improve if a different source code management system such as Git [16] or Bazaar [5] is used. Then developers are able to work with a local repository and have more control over what is committed in the central remote repository.

The number of commits into the central repository will decrease for one developer but the effort to keep the tests of the build error-free will increase. The developers will spend more time merging code. Maybe a new development team role is needed namely a person that is responsible for the code integration and the tests.

More testers need more hardware resources. Maybe more than one test system is required. Regarding the test case management the testers face the same problems as the developers with the source code. Testers will have to switch to a more powerful source code management system as well.

Chapter 10

Summary and Next Steps

To sum up, it can be said that the project was a success. The high expectations put into it have mostly been met. The Automated Test Environment has become indispensable in everyday development work. New working techniques have emerged that changed existing inefficient processes. Furthermore the test automation led to a paradigm shift regarding testing. More effort is put into the requirements specification and the test case implementation before the start of development.

Whenever a new sport has to be implemented, at first the Automated Test Environment demands a detailed specification of the event sequences to be implemented. That means that the requirements definition has become more precise and substantial but also less contradictory than before. From the detailed specifications, use cases are deducted which then are implemented into test data suitable for the test automation program. The work and the thinking of all people involved in the development has become more structured.

The change is not completely finished though. At the moment it can be said that the development process is not quite test-driven but test-accompanied. This is not the ideal way as Kent Beck postulates in his book [7] but a lot better than it was before when nearly no structured testing was done at all.

The Automated Test Environment has some unpredicted positive side-effects as well. The introduction of the XML test case description schema has created a language to talk about event sequences. The written down and stored test cases embody a vast knowledge base for RunningBall. With the work on the test cases it became more and more clear that the great value of the company does not only refer to its speed of information transmission but also in the way the information is processed to provide more value.

The requirements asked from a developer to implement a project like the Automated Test Environment are various. Solid development skills are essential. Besides that a good know-how about software testing is beneficial.

The developer should be able to communicate his progress and changes, and make them transparent to his colleagues. For me one of the most important abilities that I gained during this project was the ability to say no. This means to stay focused and implement the most important features with the highest priority first. I accepted no creative wishes for additional features that would have distracted me from the main targets until the high-priority features were completed.

An aspect that was highly underestimated during this project was human behavior when confronted with changes. We had long discussions with the test team when we first told them that we wanted to switch from the simple recording and generation of test data using the tested application to real test data definition as described in Subsection 8.2.1. The testers did not want to change their working techniques and did not want to understand the disadvantages of generating test data using the tested application. Another underestimated fact was the demand for correctness of the Automated Test Environment application itself. That meant testing it extensively and writing a lot of unit tests.

For the near future many small add-ons for the Automated Test Environment are planned. In addition to the existing adapters new ones have to be implemented to test more applications in the RunningBall software system. Whenever a new sport is to be implemented new test cases are written. This often has the effect that new functionality has to be added to the Automated Test Environment as well. One bigger topic is usability which should be improved. The manual editing of XML files is not easy and error-prone. Another topic is to provide a test evaluation that can be viewed including more than one test run. Test progress can be seen easier with such an improvement. Finally there will be some issues of growth as mentioned in Section 9.5. There is still a lot of work to be done.

List of Figures

List of Tables

Bibliography

[1] Apache Maven Project. Maven Site Plugin. http://maven.apache.org/plugins/maven-site-plugin/, July 2012.

[2] V. R. Basili and B. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, Jan, 1984.

[3] K. Bassin, S. Biyani, and P. Santhanam. Metrics to evaluate vendor developed software based on test case execution results. *IBM Systems Journal*, 41:pp. 13–30, 2002.

[4] G. Bath and J. McKay. *Praxiswissen Softwaretest - Test Analyst und Technical Test Analyst*. dpunkt.verlag, Heidelberg, 2011.

[5] Bazaar. Bazaar. http://bazaar.canonical.com/en/, April 2012.

[6] K. Beck. *Extrem Programming explained - embrace change*. Addison-Wesley, Boston, 2000.

[7] K. Beck. *Test-Driven Development by Example*. Pearson Education, Boston, 2003.

[8] A. Beer. Component testing with junit. *Vorlesungsfolien aus Ausgewählte Kapitel der Softwaretechnologie 2 an der TU Graz*, 2010.

[9] A. Beer. Fundamental test process. *Vorlesungsfolien aus Ausgewählte Kapitel der Softwaretechnologie 2 an der TU Graz*, 2010.

[10] A. Beer. Testcase design methods. *Vorlesungsfolien aus Ausgewählte Kapitel der Softwaretechnologie 2 an der TU Graz*, 2010.

[11] F. Budszuhn. *Subversion 1.5 - Das Praxisbuch*. Galileo Computing, Bonn, 2009.

[12] A. Cockburn. *Writing effective use cases*. Addison-Wesley, Boston, 2001.

[13] L. Copeland. *A Pratitioner's Guide to Software Test Design*. Artech House Publishers, Boston London, 2004.

[14] Free Software Foundation. CVS - Open Source Version Control. http://www.nongnu.org/cvs/, April 2012.

[15] K. Frühauf, J. Ludewig, and H. Sandmayr. *Software-Prüfung - Eine Anleitung zum Test und zur Inspektion.* vdf Hochschulverlag AG an der ETH Zürich, Zürich, 2007.

[16] GitHub. Git. http://git-scm.com/, November 2012.

[17] H2. H2 Database Engine. http://www.h2database.com/html/main.html, June 2012.

[18] M. H. Halstead. *Elements of Software Science.* Elsevier Science Ltd, New York, 1977.

[19] S. Hawker and J. M. Hawkins. *The Oxford Popular Dictionary & Thesaurus.* Oxford University Press, Oxford, 1999.

[20] P. Henry. *The Testing Network, An Integral Approach to Test Activities in Large Software Projects.* Springer, Berlin Heidelberg, 2008.

[21] JFrog Ltd. About Artifactory. http://www.jfrog.com/products.php, April 2012.

[22] C. Jones. *Programming Productivity.* Mcgraw-Hill College, New York, 1986.

[23] JUnit.org. JUnit. http://www.junit.org/, June 2012.

[24] S. H. Kan. *Metrics and Models in Software Quality Engineering (Second Edition).* Addison-Wesley, Boston, 2002.

[25] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing: a context-driven Approach.* Wiley Computer Publishing, New York, 2002.

[26] R. Katz and T. Allen. Investigating the not invented here (nih) syndrome: a look at the performance, tenure and communication patterns of 50 r&d project groups. *R&D Management*, 12, 1982.

[27] B. Lo. Syntactical construct based apar projection. Technical report, IBM Santa Tereas Laboratory, California, 1992.

[28] M. Lorenz. *Object-Oriented Software Metrics: A Practical Guide.* PTR Prentice Hall, New Jersey, 1993.

[29] Martin Bisanz. JUnit & Friends. http://it-republik.de/jaxenter/artikel/2528/, June 2012.

[30] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, December 1976.

[31] M. Müller. Subversion 1.5. *Javamagazin*, 9:p. 11, 2010.

[32] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, 1979.

[33] M. Pezzè and M. Young. *Software Testing and Analysis - Process, Principles, and Techniques.* John Wiley & Sons, Hoboken, 2008.

[34] RunningBall. RunningBall Global Sports Data. http://www.rball.com/, September 2012.

[35] T. Schlitt. Continuous integration mit hudson. *Entwickler Magazin*, 1:pp. 39–46, 2011.

[36] R. Seidl, M. Baumgartner, and T. Bucsics. *Basiswissen Testautomatisierung.* dpunkt.verlag, Heidelberg, 2012.

[37] Sonar Source. Sonar. http://www.sonarsource.org/, January 2012.

[38] Sourceforge.net. Checkstyle. http://checkstyle.sourceforge.net/, July 2012.

[39] Sourceforge.net. Cobertura. http://cobertura.sourceforge.net/, July 2012.

[40] Sourceforge.net. PMD. http://pmd.sourceforge.net/pmd-5.0.0/, July 2012.

[41] P. Tahchiev, F. Leme, V. Massol, and G. Gregory. *JUnit in Action.* Manning Publications, Stamford, 2011.

[42] The Apache Software Foundation. Apache Ant. http://ant.apache.org/, April 2012.

[43] The Apache Software Foundation. Apache Maven. http://maven.apache.org/, April 2012.

[44] The Apache Software Foundation. Apache Subversion. http://subversion.apache.org/, April 2012.

[45] The Apache Software Foundation. The Apache Software Foundation. http://www.apache.org/, April 2012.

[46] The Eclipse Foundation. Eclipse. http://www.eclipse.org/, June 2012.

[47] Tigris.org. Tigis.org: Open Source Software Engineering. http://www.tigris.org/, April 2012.

[48] W3C. Extensible Markup Language (XML). http://www.w3.org/XML/, September 2012.

[49] W3C. W3C XML Schema. http://www.w3.org/XML/Schema/, October 2012.

[50] G. M. Weinberg. *An Introduction to General Systems Thinking (Silver Anniversary).* Dorset House Publishing Co Inc., U.S., 2001.

[51] C. Withrow. Error density and size in ada software. *IEEE Software*, 7, January 1990.