# Implementation of a System for Automatic Software Verification

Pär Eriksson, Jim Petersson

February 5, 2006
Master's Thesis in Computing Science, 2*20 credits
Supervisor at CS-UmU: Tomas Nilsson
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

## Abstract

One fundamental problem in software development is how to verify that the product meets its specifications. Fortunately there exist numerous methods for software verification. This process can often be made more effective by automating it.

This thesis includes a survey of common techniques in this area, ranging from formal methods to more practical approaches. It also presents an implementation of a system for automatic software verification. The system uses existing verification tools and manages the execution and result reporting from them. The system also includes a web interface from which the test results can be viewed.

***Keywords:*** Software verification, automatic testing, regression testing, software development, continuous integration

## Outline

**Chapter 1:** gives a brief description of the thesis as well as the goals and objectives.

**Chapter 2:** gives an overview of software testing. Terms as unit, integration, system and regression testing is adressed and explained.

**Chapter 3:** is a description of some of the tools used today at Scania.

**Chapter 4:** will focus on how our platform was implemented. It will give an overview of all incorporated modules and the implemented tests. It will also descibe the design of the database and the web site.

**Chapter 5:** gives a summary and a discussion about the limitations of the platform. It also presents possible future work on the platform.

## Acknowledgments

We would like to thank our supervisor at Scania, Kristian Krigsman and our supervisor at Umeå University, Thomas Nilsson for their contribution and help during this thesis. We would also like to thank the people at Scania that we interviewed in the beginning of our work. Also thanks to the other thesis workers and all employees at the engine software development department (NEE), for all interesting discussions around the coffee table. Last, special thanks to Cecilia and Johanna for their patience and support.

*Pär Eriksson and Jim Petersson*
Södertälje, December 2005

# Contents

# List of Figures

# Chapter 1

# INTRODUCTION

During software development, the need for testing is crucial. This is often the only way to determine if the system meets its specifications.

Tests can spot not only errors in the code, but also other mistakes made during the development process. If the design was wrong from the beginning, it is hard to compensate for that during the implementation phase. This leads us to the conclusion that testing is important during the whole development process. For a further discussion on this, see chapter 2.1

Testing is often carried out at several different levels, from the individual programmer checking that the variables are named correct, to large test phases involving the entire development team. Nevertheless, testing is important in all different levels of abstraction.

This report was written as a part of our master thesis at the institute of computing science at Umeå university. The work has been carried out at Scania CV AB (further denoted as just Scania) in Södertälje.

## 1.1 Background

Trucks and buses from Scania have over the years become more and more dependent on different embedded systems and software. One of the reasons for using software systems is because of the increasingly stricter emission legislations. Which in turn introduces the demand of on-board diagnostics (OBD), used for instance to detect whenever emissions from the truck are not in compliance with the legislations. As new engines are developed and the emission legislations become stricter, the software has increased greatly in complexity and size. This has made the development and maintenance of such systems to a rather difficult process.

At Scania testing the engine software is commenced at many different levels. Ranging from tests of a single function, to tests of the complete software, both on its own and also fully integrated with the truck. Some tests used during development are mandatory, for instance tests that do strict syntax checking on the code. Besides these mandatory tests, individual developers have their own special test methods that only they use.

## 1.2 Goals

Our task is to design and implement a general platform for automatic software testing. The platform should support an easy way to run tests automatically and report the results by e-mail and through a web page. The primary goal is to add support for existing tests used at Scania and secondary to implement new tests.

The platform should be extendable, meaning it should be easy to add new tests. Since Scania uses a version control system (VCS) it is important that the platform also supports this, however the platform should not be limited to just one specific VCS.

To be able to communicate properly via the web page a database has to be created. Through the web page developers should be able to find information about which source files they have errors in and how specific tests went.

A key concept of the design is that the platform should provide a dynamic environment which supports the possibility to add new functionality in the future.

The implementation is to be done in a PC/Windows environment.

## 1.3 Purposes

By having a platform that supports automatic testing and verification of the engine software, Scania strives to improve the development process. The expected results are a reduced implementation time as well as more controlled and centralized testing service. The time aspect is of great importance, since a system that is not functioning properly can slow down the development of the entire truck.

By having a centralized testing platform, the developers have easy control of which tests to run and the tests are conducted in a more organized way. By automating the process, Scania wishes to make sure that no errors are left undiscovered and that the whole system instead is checked and verified regularly.

## 1.4 Methods

Our work has consisted of several parts, and the methods used for solving the problem have been different for each.

In the beginning several interviews were made with developers at Scania. This was done to gain insight in the different techniques used for testing. The interviews also helped identifying problems and limitation with current tools.

A comprehensive literature study in the area of software testing and verification has also been made to obtain a theoretical background of the area.

Furthermore, implementation of the system has commenced in parallel with the other work described above. And, as questions and problems have arisen, literature or human resources have yet again been consulted.

The last part of this master thesis work has been more practical, as is has included integration of the system into Scania's development process.

# Chapter 2

# SOFTWARE VERIFICATION

## 2.1 Introduction

Software testing is a crucial part of most software development. But this often is not an easy, straightforward task. Up to 50% of the total cost of software development is associated with verification [10]. Verifying software often includes writing hundreds of lines of test case code, analyzing complex program executions and understanding the behavior of compounded systems.

We will now explain in more detail why, when and how software verification is done. Furthermore, we will give details of several different techniques for doing this, as well as their applications. This chapter also explains how to automate testing and the concept of continuous integration.

### 2.1.1 Why Test

When an engineer solves a problem, he needs to know if the solution really is the wanted one. This is a basic foundation that applies to most engineered work. In some cases, e.g. solving equation systems in mathematics, this verification process can be easily accomplished and does not need rigorous work. However, in software engineering testing is seldom that easy.

The main purpose to test software is to increase the quality. Often this refers to assuring that the system specification has been met, and that no other errors are present. However, errors can also be present in the system specification itself, which makes this definition a bit inaccurate.

Releasing software that contains errors can be combined with huge expenses. Not only the direct economical cost for fixing the bug, but also loss of confidence in the brand itself. This can lead to serious longterm problems.

A real life example is taken from the American telephone company AT&T [31]. A bug in their software system affected the long distance network, causing 50% of the calls to not be passed through. A whole day with failures passed before patches were installed to fix the bug. The direct cost of the lost traffic was between \$60 and \$75 million in addition to a loss of confidence.

### 2.1.2 What is Testable

What is testable is often a question of how much resources it is worth spending on testing it. Most systems and components can be tested, one way or another. The difference lies in how the tests have to be constructed. To illustrate this, imagine two functions; one that multiplies two values, and one that predicts the fuel consumption for a car. It is obvious that the latter is harder to test, because it depends on more things, e.g. type of engine, speed, road profile etc. Also, it is hard to know the expected result for such a function, given a specific input. It can, of course, be based on measured values, but it is no guarantee that they are correct.

Although the fuel consumption calculator requires much more testing, verification of the multiplication function should not be omitted either. Even if it seems to work like it should (for some values), what happens if two negative values are passed to it: if one value is very, very large or if one or both values are zero? This example points out one interesting and important fact in software testing; there is very much that can be tested. However, in real life, there is much e.g. the multiplication operator in programming languages, that already is thoroughly tested and checked.

To generalize testing there exist several different kinds of tests, which occasionally overlaps. These are adopted from the IEEE standard for software test documentation [6] and later used by [34]. Note that several of these will be further addressed later.

#### Unit

Component testing is about testing parts of the system to make sure that they follows the specification and operates as they are expected to. This can also be called component testing.

#### Integration

Testing where software or hardware components are combined, and the tests checks if they, when combined, operates correctly.

#### Recovery

Testing done to check if the restart and recovery mechanisms are working as designed.

#### Security

As security often is an important issue, these tests make sure that all security aspects have been revised.

#### Regression

Regression testing is about having tests for several parts of the system, and if changes are applied, run these tests again to check that the modifications did not introduce any new errors in previously correct code.

Many of these can be categorized as either *white box* or *black box* testing [34]. Black–box test treats the system as a “black–box”, so it doesn’t explicitly use knowledge of the internal structure. Black–box test design is usually described as focusing on testing functional requirements. White–box test focuses specifically on using internal knowledge, e.g. the actual source code of the software to guide the creation of tests.

### 2.1.3 When to Test

There exist many different approaches to when testing should be done. To generalise these, software development life cycles can be categorized into different strategies, or *methodologies*. We will now present some of the most common ones, along with an explanation of them. As we emphasize testing phases, details about the other phases are partial.

#### Code and fix

This is probably the most common methodology among beginners to software engineering [1]. Here, no specific design and specification process is present nor any test phase. Instead coding, compilation and testing are mixed unregulated, until a solution has been found. It can be thought of as a trial-and-error approach, and may work well for very small projects.

One major disadvantage of this method is that the testing phase is not clearly defined. Often testing occurs only if the program is not compilable.

#### Waterfall model

The *waterfall model* is a sequential model and is also one of the first formalized methodologies [1]. The first step is a requirement definition and analysis phase. This is followed by a general and detailed design phase which leads to the actual coding. When the implementation phase has completed, testing of units and integrated parts commences (see Fig. 2.1). The disadvantage with the waterfall model is that it is actually more suitable for developing hardware systems. This is because developing hardware systems is a more rigid process and the success can be more easily determined. To address this problem, modifications of the waterfall model have been made. One is the *V-model*, where every development step is associated with a test or verification phase at the same level of abstraction.

#### Spiral model

While the waterfall model was a sequential model, the *spiral model* is more of a cyclic model [11]. It consists of four different phases that iterates for some time (see Fig. 2.2):

- Feasability study: Determines objectives, alternatives and constraints.
- Evaluation and prototyping.
- Development of the product.
- Planning for the next round.

The major advantage with this approach is that evaluation and risk analysis is a recurring phase that is performed on all previous work. This makes it suitable for projects where user requirements are changing.

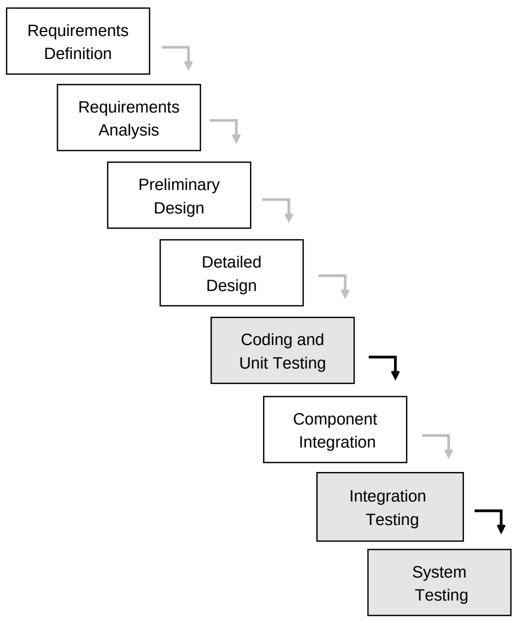


Figure 2.1: The different stages in the waterfall model, with the testing phases emphasised. Adopted from [1].

Testing the code is partly done for every new prototype developed, however a more rigorous test phase begins first when the full system has been completed. This includes unit-, integration- and system tests, which we will address later.

**Agile software development**

*Agile* development methods accept the fact that it is hard to produce a complete, well-defined specification from the beginning of a project. Instead they try to adopt the changes during the project in an effective way [1].

One popular agile methodology is *Extreme Programming*(XP). XP is based on principles of simplicity, communication and feedback. Testing is also a key concept in XP, and it is carried out somewhat different from the other methods discussed.

XP utilize Test-Driven Development (TDD), so testing is a continuous activity during the development process [35]. One concept is that tests always are formulated before the corresponding code is written. By this, the developers strive to verify the code at all time. Also, as soon as a bug is present, a test is written to ensure that the bug does not appear undiscovered again.

This will conclude our discussion of when testing are done during the development process. Note also that there are many other development methodologies, however with respect to the amount of testing involved, these are a representative subset.

### 2.1.4 When to Stop?

When to stop testing is a very hard question. It is impossible to continue testing until all the defects are discovered and removed, since there is no way to know when this is fulfilled. A common, but not very good, strategy today is to test until the project

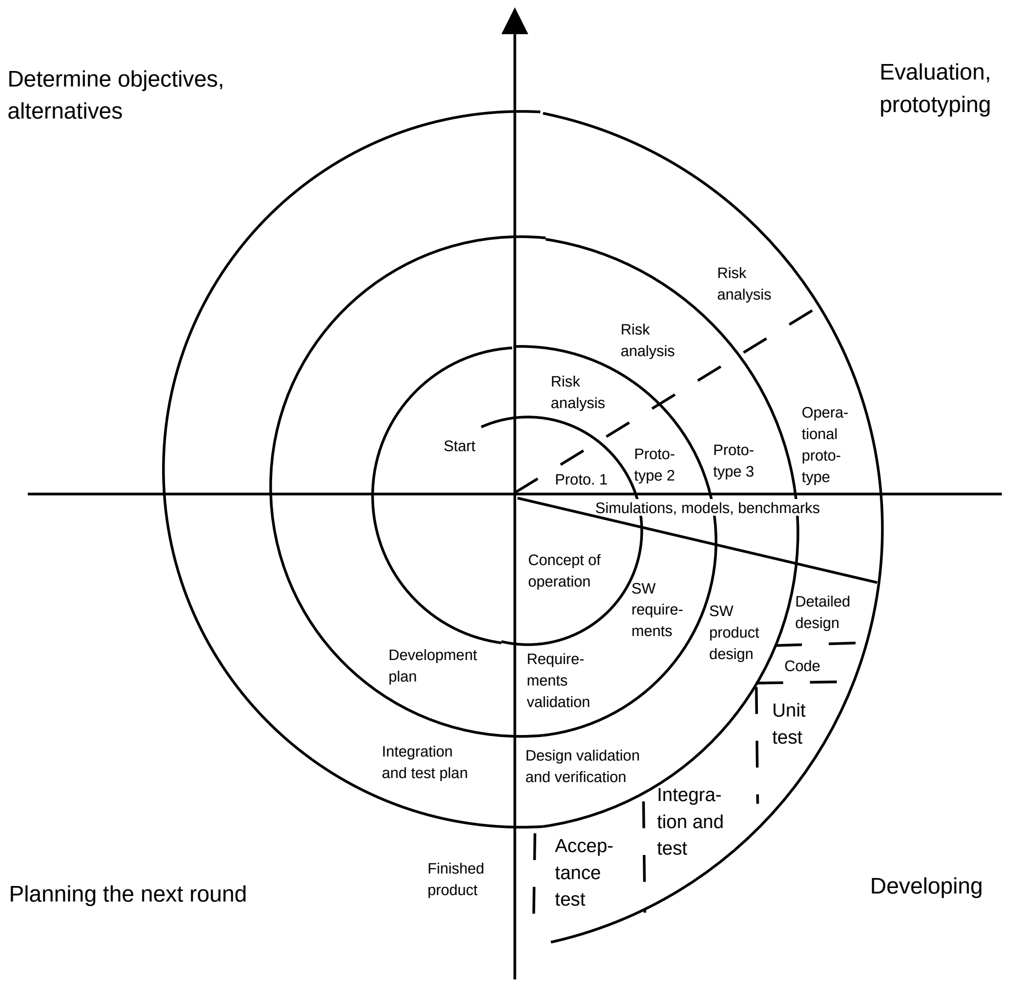


Figure 2.2: The spiral model, with the testing phases emphasised. Adopted from [11].

runs out of money or time. If the development process takes longer then was originally planned, the testing phase will suffer due to lack of time and/or money. Instead some other measure is needed. We will present three common strategies [2] for deciding when to stop testing:

### Defect density

This method uses the *defect density* as a measure of quality. The density is calculated as follows:

$$density = errors/codesize$$

where *errors* is the total number of errors discovered so far and *codesize* is thousands of lines of code (KLOC).

Suppose program A consists of 100 000 lines of code and that 700 errors were detected during the testing phase. Program A therefore has a defect density of: 700 / 100 = 7 errors per KLOC.

Assume further that the next program to be released, program B, consisted of 50 000 lines of code and 475 errors were discovered. The density of program B would then be 9.5 errors per KLOC.

Now suppose it is time to decide if a program, C is ready for release? It consists of 75 000 lines of code, and 390 errors has been detected. The density for program C would therefore be 5.2 errors per KLOC. With the prior results in mind, the expected density should be between 7 and 9.5. Since C's density is lower then this, it could be a hint that more errors remains to be found.

Of course it could simply be the case that program C was better coded and had fewer errors, but if density had been tracked for the last 20 projects and they had an average density of 8.2 with a small variance, more testing *is* probably needed.

### Defect pooling

This technique separates all discovered errors into two separate pools, A and B. Which errors goes into which pool is arbitrary, for instance all errors discovered by one developer could be put in pool A, and all errors discovered by another in pool B. Once the pool distinction is made, errors in the pools are tracked and compared. The number of unique errors reported at any given time is:

$$Errors_{Unique} = Errors_A + Errors_B - Errors_{AandB}$$

The number of total errors can be approximated by the formula:

$$Errors_{Total} = (Errors_A * Errors_B)/Errors_{AandB}$$

If the version 3.0 project has 400 errors in pool A, 350 errors in pool B, and 150 of the errors in both pools, the number of unique errors detected would be 400 + 350 - 150 = 600. The approximated total number of errors would be 400 * 350 / 150 = 933. This would mean that 333 errors are still undiscovered.

### Defect seeding

With defect seeding, one group intentionally inserts errors into the code, which are left for another group to discover. The ratio of the number of seeded errors detected, to the total number of errors seeded, provides a rough idea of the total number of unseeded ("real") errors that have been detected. The total number of errors can be approximated with the following formula:

$$RealErrors_{Total} = (SeededDefects_{Planted}/SeededDefects_{Found}) * RealErrors_{Found}$$

Suppose version 3.0 has 50 seeded errors. After some testing 31 seeded errors and 600 "real" errors has been found. The total number of errors could then be approximated as: 50 / 31 * 600 = 967. This means that there still is 367 approximated real errors left.

A common problem with this technique is forgetting to remove the seeded errors, or that new errors are introduced when they are removed.

We now also end the introduction to software testing, and move on to more specific verification techniques. First we will present *formal methods*, which is a theoretical approach to verification.

## 2.2 Formal Methods

Formal methods are a way of specifying and verifying software systems by applying techniques from mathematics and logic. The general outline of the method can be summarized in three main steps [14]:

1. *Formal Specification*: During the formal specification phase, the engineer completely defines the system using a modeling language. Modeling languages are fixed grammars which allow users to model the system with predefined types. This process of formal specification could be seen as the process of converting a word problem into algebraic notation.

2. *Verification*: After the first step is completed the engineer has ended up with a number of theorems describing the system. In the verification phase these theorems should be proven correct. Verification is a difficult and time consuming process, since even a simple system will produce several theorems that have to be proven. However, given the often high demand of time efficient software development, almost all practical use of formal methods take use of some automated theorem proving tool.

3. *Implementation*: Once all theorems have been proven correct, “all” that remains is to translate them into executable code.

The strength of formal methods is best illustrated when compared to “regular testing”. In 1972 Dijikstra claimed that “program testing can be used to show the presence of bugs, but never their absence” [19]. In other words, just because a test did not *find* any errors, it does not *prove* that the program is free from them. In contrast to this, formal methods can actually prove the specification to be error free. As systems become more complex, and safety becomes a more important issue, the formal approach to system verification offers an additional level of insurance.

Traditionally, formal methods have mostly been of academic interest and use, however lately it has been increasingly accepted in the industry [12]. It is now recognized that formal methods and testing are complementary techniques which can, and should be used in combination.

### 2.2.1 Modeling Languages

In order to describe a system in theorems, a *formal modeling language* is needed. There exist a number of such languages, where VDM and Z seem to be the most used today [32]. They are also among the few for which ISO standards exist. Both Z and VDM are based on set theory and first order predicate calculus, but they differ in syntax and structure. Another distinguishing feature is that Z has a way of decomposing specifications into small pieces called *schemas*.

A schema describing some part of the system, can among other things include information about the states it can occupy, the operations that are possible and the relationship between input and output. To give a clearer illustration of this, an example will be given in the following section.

### Z–notation

The following example is based on an example in [42] and illustrates how a small system can be represented in Z-notation. The system records people's names and their adresses.

It needs to keep track of peoples names and their corresponding adresses. Therefore the set of all names and the set of all adresses, [NAME, ADRESS] is introduced. The first step after this is to describe the systems state space, which is done by the following schema:

| *AdressBook* | |
|---|---|
| $known$: | NAME |
| $getAdress$: | NAME $\rightarrow$ ADRESS |
| $known =$ | dom $adress$ |

The schema is structured as follows: at the top is the name, in the middle declared variables and at the bottom relationships. This particular schema is named *AdressBook* and has two variables:

– *known* is the set of names with adresses recorded.

– *getAdress* is a function which, when applied to certain names, gives the adress associated with them.

The statement at the bottom of the schema describes a relationship which is true in every state of the system; it simply says that the set *known* is the same as the domain of the function *getAdress*. This means that *known* is a derived variable and it would be possible to specify the system without mentioning *known* at all. However, giving names to important concepts helps to make specifications easier to read.

One possible state of the system has three people in the set *known*, with their adresses recorded by the function *getAdress*:

– $known$ = {John; Mike; Susan}

– $getAdress$ = {John $\rightarrow$ Queen Street 23, Mike $\rightarrow$ Kings road 5, Susan $\rightarrow$ Green street 45}

Now that the state space is defined, it is possible to make operations on it. For instance the operation "add a new adress" could be specified in another schema:

| *AddAdress* | |
|---|---|
| $\Delta AdressBook$ | |
| $name?$: | NAME |
| $adress?$: | ADRESS |
| $name? \notin$ | $known$ |
| $getAdress' =$ | $getAdress \cup \{name? \rightarrow adress?\}$ |

The $\Delta$*AdressBook* declaration tells us that this operation will lead to a state change. The two variables with question marks after them (*name?* and *adress?*) are the input values. The part of the schema below the line first of all gives a pre–condition for this operation: the name to be added can not be one of those already known to the system (names must be unique). The second expression says that once the pre–condition is met, the *getAdress* function is extended to map the new name to the given adress.

After an adress has been succesfully added, it is natural to assume that the set of names known to the system will be extended with the new name, such that:

$$known' = known \cup name?$$

This can in fact be proven in a few steps, the proof will be left out here, but can be found in [42]. Stating assumptions like the one above and then proving them is a good way of making sure that the specification is accurate. Once a proof like this is done, its is verified that the system behaves correct. In contrast, if the system was only to be tested, a number of test cases would have to be written and even then, we would only be sure that the specification was met under those special test conditions.

Antoher operation on the system could be to find the adress of a person known to the system. This is also representented in a schema:

| *FindAdress* | |
|---|---|
| $\Xi$*AdressBook* | |
| *name?*: | NAME |
| *adress!*: | ADRESS |
| *name?* $\in$ | *known* |
| *adress!* $=$ | *getAdress(name?)* |

The declaration $\Xi$*AdressBook* indicates that this is an operation in which the state does not change. The use of an exclamation mark means that it is an output value: the *FindAdress* operation takes a name as input and returns the corresponding adress as output. The pre–condition for success of the operation is that *name?* is one of the names known to the system, if this condition is met, the output *adress!* is the value of the *getAdress* function with the argument *name?*.

From this small example, it is clear that formal methods require rigorous work. Even though it is possible to take use of automated theorem proving tools, it is still a lot manual labor to specify all theorems, functions and variables. Also note that the previous example is in no way complete, for instance it does not say anything about error handling. What should happen if the user tries to add a name already known to the system? By adding error handling to the system, a number of more theorems to prove would be yielded. Imagine the work needed if a large complex system should be proven correct by formal methods.

This illustrates one of the drawbacks with formal methods. Since it is a time consuming process, it may not be applicable in all projects. Therefore the most widespread technique for software verification is *testing*, which we will now move onto.

## 2.3 Testing at Different Levels

Testing is a process used to identify the correctness and quality of software. This is usually done by creating specific test cases and run them on parts of, or the whole program. A test can for instance check the return value of a specific function and compare it to the expected output. Usually the process of creating a specific test case is much faster then the creation of a formal proof, which can be extremely difficult or even impossible to create.

Software testing is usually categorized into three levels: unit, integration and system. Testing at different levels reveals different types of errors [30]. A description of each level now follows.

### 2.3.1 Unit Testing

A unit is a small piece of code, e.g. a function or a method. The idea of unit testing is, as the name implies, to do tests on these units and determine whether it behaves as you expect. The units should be isolated from the remainder of the code and to guarantee this, it can be a good idea to lift out the unit of its natural environment and execute it in a separate testing framework. By doing so, no dependencies to the other code can exist.

Unit testing is closely related to XP in the sense that one of XP's corner stones is that a test (unit test) should always be written before you write the actual code. By following this principle, the number of tests soon become very large, making it hard to organize and run them properly. To aid the developer, a number of unit testing frameworks have been developed. In 1994 Beck [8] introduced one such framework created for Smalltalk. The framework supported the user in creating unit tests, and managing them into *test suits*. Beck's program has later been converted to fit over 20 different languages and is now known collectively as xUnit.

By doing thorough unit tests, the confidence in the separate units' functionality increases. It is however not possible to predict if the interaction between the units meet the specification. This is the responsibility of integration testing to verify.

### 2.3.2 Integration Testing

The essential idea of integration testing is to take two or more separate units (which have been tested on their own) and combine them into one larger *component*. After this the combined units' interfaces are tested. A component can of course consist of more than two units, but it is a good idea to start with few units, run some tests and then add more units. If the units themselves have been tested before, any errors discovered at integration testing phase is most likely related to some error in their interfaces. Knowing this greatly reduces the number of possible error sources.

Eickelman and Richardson [20] describes six traditional strategies for integration testing: *critical module, sandwich, thread, bottom up, top down* and *big bang.*

– The critical module strategy uses a prioritization of the components to determine which should be integrated and tested first. As the name implies, the components to test first are those with critical functions, e.g. those related to safety.

– The top down approach requires the highest-level units to be integrated and tested first. This allows high level functionality and data flow to be tested early in the process. A drawback of this approach is that the high level components probably depend on some middle level components. One common solution to this is the use of *stubs*. A stub is an incomplete subprogram which are only present to allow the higher level functions to be tested.

- The bottom up approach instead requires the lowest level units to be integrated and tested first. Once they have been tested, higher level units are integrated. This makes the need for stubs minimal. But low level components are usually not intended to be run directly; instead they are most likely supposed to get called from a higher level. Because of this an external *driver*, which simulates low level calls directly is usually needed. Another drawback is that high level data flow is tested late.

- The sandwich approach combines top down and bottom up incrementally, meeting in the middle. Since the top down approach needs stubs and the bottom up approach needs drivers, the sandwich approach must have both stubs and drivers created, though it may reduce the total number needed.

- The thread test approach tests a single thread of control first and then it integrates additional threads for the entire system.

The sixth strategy is called Big bang and is simply a test of the full system. A test like this is however more often categorized as system level tests, which now follows.

### 2.3.3 System Testing

The goal of system testing is to test the whole system against its specifications [30]. It should verify that all components have been properly integrated and are functioning properly in its typical working environment. The distinction between integration testing and system testing is not always clear, since many of the tests conducted at system level could also be done at the integration level. Following are some typical system level tests:

- Recovery testing: The software is deliberately interrupted in a number of ways, for example taking the hard disc off line or even turning the power off. This is done to ensure that the appropriate techniques for restoring any lost data will function properly.

- Security testing: Unauthorised attempts to run the software, or parts of it, are attempted. It could also include attempts to gain access to data that should not be allowed.

- Stress testing: The limit of the system is tested by making abnormal demands on it. It could for instance include unusual large network traffic, extremely large input data to the system and so on.

A study [27] compared the time taken to prepare and execute tests at the unit, integration and system level. The results showed that unit testing was about two times as cost effective as integration testing, and more than three times as cost effective as system testing.

This is the traditional way of dividing testing. But when testing object oriented systems this may not be optimal.

### 2.3.4 Object-Oriented Systems

When developing object oriented (OO) systems, the standard way of testing may not be useful [26]. This is because the traditional, sequential way of executing a program is changed. Instead, in OO systems, objects can be combined in an arbitrary order, which introduce problems when testing with standard techniques.

In comparison to a classic procedural system, OO systems tend to have more of a state-based nature. Seipmann and Newton [40] claim that this can have a negative effect on testing since the objects are not only defined by their state values, but also by their associations to other objects. Also, the inheritance concept that is one of the fundamentals in OO programming, introduce problems. The question here is how testing of a subclass (derived class) affects the verification of the superclass (base class).

The traditional way of dividing testing into unit, integration and system is also a subject of investigation in OO systems. Jorgensen and Erickson [23] points out that there exist many different definitions of a unit, e.g. a single function, the smallest compilable segment of code or the amount of code that one person writes. Also, in OO systems, the definition of a unit may be a single class. Different researchers address this confusion differently, but we will present the definitions given by Johnson [26]:

- Unit Testing - Testing a method independently of other methods. However, this may require attributes or stubs for other methods to work.
- Class Testing - Testing a class and its methods when they interact. As with unit testing, this may need stubs to other class methods.
- Cluster Testing - In this category testing is conducted by grouping classes together and testing the interaction between them.
- System Testing - As before, system testing is about verifying the whole system in an environment that resembles the real configuration as accurately as possible.

Note that integration testing is replaced by class and cluster testing. This is because the object entity splits the integration testing phase into two parts. Some of the other problems that are associated with testing OO system may be addressed in the following way:

- Inheritance - To make use of the object hierarchy, test only those methods that are not already tested in the superclasses.
- Memory - Some languages, e.g. Java, have *garbage collection*. This should be verified by ensuring that the objects that not have any references to it are removed from the memory.
- Data flow - As objects can be constructed and destructed during runtime, this should be monitored to find abnormalities in the data flow.

### 2.3.5 Summary

Traditionally, testing can be divided into three categories: unit, integration and system testing. Unit testing is concerned with verifying small units of the code, e.g. separate functions. To be able to do this isolation, stubs are sometimes needed to replace other functions. Integration testing verifies the combination of several units, and system testing tests the whole system, including security and performance testing.

In object oriented systems integration testing often splits into class and cluster testing. This is because of the object distinction, and that both an object and its methods should be tested separately.

## 2.4 Static Testing

So far, we have been concerned with testing that includes code execution. By code execution we mean how the system operates. These kind of tests are also called *dynamic tests* [45]. Another kind of tests is more into verifying the code itself and how it is written. Consequently, these tests are called *static tests*.

A summary of some static testing areas now follows, based on [25].

#### Control flow analysis

This test is conducted to ensure that the code is executed in the right sequence and well structured. Another objective is to locate any unreachable code, and in that case, warn the developer. It is also concerned with checking if termination someplace needs to be specified. This can for instance be the case in loops.

Erroneous example:

```
for(int i=0;i<10;i++){
   if(i==9)
      return 10;
}
return 100; //Unreachable code
```

#### Data flow analysis

The objective here is to inspect how data flows to and from variables. This can for instance be to check that no variable are read before it has been assigned anything, or multiple writes without intervening reads. This process can be very complex, as global variables can be accessed from everywhere.

Erroneous example:

```
int myVar;

if(myVar == 0) //myVar not initialized
   [do this]
else
   [do that]
```

#### Information flow analysis

This test checks how the information flow in the code creates dependencies between input and output. For instance, it may spot loops where the input of one function depends on itself in some way, which often is unwanted.

#### Syntax checking

The objective is to find violations to the languages rules. It checks whether variables are used before they are declared or if variables of the wrong type are used.

Another test that falls into this category is naming convention checking. Its objective is to check that variable and function names follow a specific naming convention that has been specified in the project.

Erroneous example:

```
float a = 3.1;
int[5] myArr;

myArr[a]=2; //indexing with float
```

#### Range checking

The aim here is to check that the data remains within their specified ranges. It also verifies that the specified accuracy of the data is maintained. Some examples are: overflow and underflow analysis, rounding errors and array bounds.

Erroneous example:

```
int[10] myArr;
for(int i=0;i<20;i++)
   myArr[i]=1; //array overflow
```

#### Object code analysis

This test aims to confirm that the compiled data is a correct translation of the code. That is, checking that the compiler did not introduce any errors. This is often done by checking the mapping between the source code and the machine code.

## 2.5 Regression Testing

Anytime an error is discovered and the program is modified in order to correct the error, four separate outcomes are possible [47]. The modification can:

– Correct the error

– Fail to correct the error

– Correct the error, but also introduce a new error in previously correct code

– Fail to correct the error and also introduce a new error

The wanted outcome is of course the first one, but it would be desirable to be alerted if this is not the case. A *Regression test* does just that, it is a testing process that is used to determine if a modified program still meets its specifications. It is in other words, a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.

The idea of regression testing works well with the concept of XP, because by following XP´s principles the developer has already written test cases which most likely can be used in the regression *test suite*. A test suite is simply a collection of tests.

With the traditional classification of testing (unit, integration and system testing), it is worth mentioning that regression testing is not limited to just one of them, but can be and often are, applied at all of these levels.

### 2.5.1 Classification

As said before, regression testing is necessary whenever the software has been modified. There are two types of modifications: Adaptive or perfective modification is performed when the program specifications have changed and corrective modification is done to correct an error in the program when the specifications have not changed.

Using this distinction, regression testing can be classified as either *Progressive* or *Corrective* [30]. Progressive regression testing is performed when the specifications have changed and consequently, new test cases have to be created. Corrective regression testing is performed when the specifications are unmodified, therefore existing test cases can be used.

### 2.5.2 Test Case Selection

Regression testing can be a very time consuming process, especially when the test suite contains a large number of tests. This arise the question of which tests to run? It is desired to find the maximum number of errors using the minimum number of test cases, that is, make the regression test phase as time efficient as possible.

The simplest strategy is to run all tests at all times, this is called the *retest all strategy*. This strategy may not be time efficient since it often runs unnecessary tests. If the program has been affected by a minor modification, it is likely that most of the tests will give the same result as the last time.

The *selective strategy* instead uses a subset of the existing tests and in that way tries to cut down the time for test execution. The challenge with the selective strategy is how to choose which tests to run at a given time. Graves et. al [44] presents an outline of a typical regression test with a selective strategy:

1. Let $P$ be a program, $P'$ be a modified version of $P$ and let $T$ be a test suite for $P$
2. Select $T' \subseteq T$, to be the suite of tests to run on $P'$
3. Test $P'$ with $T'$, ensuring that $P'$ is correct with respect to $T'$
4. If neccessary, create $T''$, a new set of tests for $P'$
5. Test $P'$ with $T''$, ensuring that $P'$ is correct with respect to $T''$
6. Create $T''' = T' \cup T''$, the new test suite for $P'$

An important issue when using a selective strategy is that the selection phase (step 2) must not take to long. If the time selecting cases plus the time executing them is larger then, or equal to the time taken to execute all tests, nothing is gained. A number of selection techniques have been presented over the years, we will look further into one of those, namely Pythia.

### Pythia

Pythia is a *safe* test case selection technique that uses a *textual differencing* approach to find test cases. A safe selective strategy [37] selects every test from the existing suite that can expose errors in the program; it may however select unnecessary tests as well. The method was developed by Vokolos and Frankl [46]. Pythia was implemented using three common UNIX programs: `cc`, the C language compiler; `pretty`, a beautifier for C programs; and `diff`, a file comparison program. As a consequence of the tools involved, Pythia can only be used on C programs.

Pythia maintains a history of which block of code that was executed for each test case in $T$. The first step in the selection is to compare $P$ with $P'$ to identify modified code. This is done with `diff`. Since `diff` analyzes each line lexically, it could report irrelevant changes in the code, for instance formatting differences. To avoid this, Pythia transforms the code into a standard form. This is done using `pretty`. Following is an example of some code before and after using a tool similar to `pretty`:

**Code before:**

```
int foo(int k) {
  if(k==11) printf("hello\n");
   else printf("good bye\n");
}
```

**Code after:**

```
int foo(int k)
{
   if( k == 11 )
     printf("hello\n");
   else
     printf("good bye\n");
}
```

After the differences between $P$ and $P'$ has been found, Pythia creates $T'$ from those test cases in $T$ that executed the modified blocks of code. The execution tracing is done by using `cc`.

This is just the basic overview of Pythia, for a more detailed description refer to [46].

### 2.5.3 Regression Testing in Practice

Onoma et. al has done a study where they looked at how regression testing is done in the industry [7]. They found out that even though development itself differs from one company to another, the process of regression testing is rather similar and can be summarized in the following steps:

1. Modification request: The software is changed due to a bug or because the specifications has changed.
2. Test case selection: Select test cases that are appropriate to use for the modified code.

3. Test Execution: Execute the selected tests. Since the number of tests often is large, this step is usually automated.

4. Failure identification by examining the test results: After the tests have been executed, their results must be examined to see if the modified software behaves correctly. This is often done by comparing test output with expected output.

5. Fault identification: If a test has indicated an error, the source code associated with it has to be examined.

6. Fault mitigation: Once the error is located it should be corrected.

They also found out that some companies did not bother with a test case selection; instead they simply used the retest-all strategy. Even if a selective strategy is not used, it is important to do a *test case revalidation* from time to time. The revalidation is done to identify test cases that are no longer valid for the modified software, e.g. a test that checks the output from a function, which in its modified version has changed its output. When an invalid test case is found, it should either be discarded or rewritten to fit the modified code.

### 2.5.4 Summary

Regression testing is conducted to ensure that a modified program still meets its specifications. Corrective regression testing is performed when the specifications are unmodified and progressive regression testing is used when the specifications have changed. An important issue is the problem of test case selection. The most simple approach is to run all tests, this is called the retest all strategy, however it is often unnecessary to run all test cases if only minor parts of the code have changed. A selective strategy instead tries to identify only those test cases that are testing modified code.

## 2.6 Automatic Testing

By following the principles mentioned so far, it will not be long in the development process before a large number of tests are created. Since executing and examining all tests can take an extensive amount of time, (especially if they are to be executed frequently), it would be desirable if this process could be automated. As a matter of fact, for large projects, automated testing is often crucial if the test suite should be executed at a regular basis. Nevertheless, having an automated testing process does not make manual testing obsolete, nor is it applicable for all sorts of tests.

### 2.6.1 Automatic Generation of Test Cases

Even if the execution of the tests are automated, it is still a tedious work to manually create all new tests. Following are two approaches to automate this:

#### Equivalent output

Korel and Al-Yami proposes a way to automate the generation of test cases [29]. Their approach is intended for regression testing at the unit level and their main goal is to only create tests that reveal errors, i.e. no unnecessary tests. In short, this is done by comparing common functionality in the original program, $M_{old}$ with the modified version, $M_{new}$.

If $x$ is output from $M_{old}$ and $y$ is output from $M_{new}$, then an error *may* have been discovered if $x \neq y$. In order to be sure that a "real" error has been discovered, it is important that $x$ and $y$ *actually* should be identical. If this is the case the output values are said to be *equivalent.* Equivalent output values represent common functionality of $M_{old}$ and $M_{new}$. In order to use Korels and Al-Yamis approach, all such pairs of equivalent parameters in $M_{old}$ and $M_{new}$ has to be identified . When this is done, the actual generation of test cases can commence.

The process can be summarized in four steps:

1. Generate input values: $z$
2. call $M_{old}$(in: $z$, out: $x$ )
3. call $M_{new}$(in: $z$, out: $y$)
4. if $x \neq y$, then report an error

As can be seen the general procedure is not very difficult, yet one problem remains: How to generate the input values ($z$) for which $M_{old}$ and $M_{new}$ produce different results? Korels and Al-Yamis show that this problem may be reduced to the problem of finding input values that executes a *selected program statement.* In our case that is step 4 (if $x \neq y$, then report an error). In other words, the wanted input values are those that lead to an execution of step 4. If such values are found, then an error has been found and a test case can be created. This test case can then be added to the existing test suite and used in future testing phases, to make sure that this error does not occur again.

There exist tools that tries to generate input values, for which selected program statements are executed. One such program is TESTGEN [28], which is a test data generation for Pascal programs. Simplified, TESTGEN takes an implementation of the four steps above, and then it tries to find input values that lead to execution of the last step.

### Branch coverage

Another approach to automatically generate test cases is to create tests so that certain parts of the code get executed. Tests that use this criterion are called *coverage tests.* [33]. The simplest coverage criterion used in practice is the *branch criterion.* This requires that every conditional branch of the program gets executed. For example, obtaining full branch coverage of the following code:

```
if(a >= b)
   {do this}
else
   {do that}
```

would require program input where $a$ is greater than or equal to $b$ and another where $a$ is less than $b$. This would assure that both "do this" and "do that" get executed.

One program that uses this approach is JTest [3], which is a tool for static analysis and test case generation for java code. JTest tries to create test cases that execute every possible branch of each method it tests. For example, if the method contains a conditional statement (such as an if block), JTest will generate test cases that test the true and false outcomes of the if statement.

Using the branch coverage approach, test cases that do not discover any errors will undoubtedly be created, while Korel and Al-Yamis approach creates only tests that will uncover an error. However, this does not mean that the branch coverage is an obsolete approach to test case generation. There are occasions when Korel and Al-Yamis approach is not applicable. Their approach demands for instance that the modified method still has the same output (equivalent) as the old one, which is naturally not always the case. Also their approach requires some manual work in finding these equivalent parameters.

One important aspect of the branch coverage method is that even though every conditional statement has been *executed* without crashing, it does not mean that the statements are error free. To be assured of this, the actual output from the tested statements has to be examined.

### 2.6.2 Manual Work Still Needed

It is often the manual tasks that reveal most new errors, not the automated ones. An automated test simply revalidates the code it is testing. It is not the *repetition* of an automatic test case that expose most errors, but the *development* of the test case itself. The repetition of automated tests simply makes sure that the error does not occur again. As much as up to 80% of the errors found during automated testing are found during the development of test cases [38].

### 2.6.3 Common Mistakes

Berner et al. has followed five companies where the process of automating testing has not been entirely successful [38]. By examining the companies' testing process they discovered four common mistakes with regard to test automation:

#### Misplaced or forgotten

Tests that are hard to do manually are most likely even harder to do automatically. The reason why a test is hard to carry out could be because it is situated at the wrong testing level. For instance, unit tests are usually concerned with program logic. Doing a unit test at the system level would be complicated and unproductive. It was also the case in some companies that they only did tests on the unit and system level, i.e. the integration test phase was left out. This will most likely lead to difficulties when doing tests at the system level.

#### Wrong expectations

In many companies the expectations of test automation was unrealistic. They expected to save a lot of money and time with the smallest effort. As mentioned before, manual testing is still needed and time has to be spent on keeping the test suite up to date. If the test suite is not maintained regularly, it will soon contain many inadequate test cases.

#### Missing diversification

When a testing process is to be automated, the natural way to start is to automate whatever the testers have been doing manually so far. As a result many companies start by automating some GUI based system tests. This is often not a good idea, since the development of test cases interacting with a graphical user interface (GUI) can be very time consuming; GUI's tend to change frequently, which forces test cases to be rewritten. Also, verification of the system may be hard since the systems state is not always visible.

#### Restricted to execution

Often only test execution is considered for automation, but sometimes it can be more effective to automate other processes. This can for instance be installation and configuration procedures. It can also be a good idea to use an automatic tool for test case generation and/or test case analysis. Though, as we mentioned earlier, up to 80% of the errors are found during the development of test cases. With this in mind it is important that the tool used for automatic test case development generates high-quality test cases.

### 2.6.4 Summary

Since the number of tests often become quite large it is important to automate this process to some extent. The easiest part to automate is often the execution of the tests, but other parts can be automated as well. For instance there exist different strategies to automate the generation of test cases. One such strategy tries to create test cases that execute every conditional part of the code, this is called branch coverage. When automating the test process it is important to realize that manual work will still be needed; test cases has to be revalidated from time to time, test output may need human inspection etc.

## 2.7 Continuous Integration

As long as small software development projects are discussed, which perhaps only include one developer, checking that the whole project is functioning is relatively easy. This is because the developer often can have every part and aspect of the program in his head. Usually the whole program is also built and run several times per day [24].

However, as the project gets larger, every developer is working on his part of the system and it might be hard for the individual to get an overview of the whole project. When this is the case, the need for *continuous integration* increases. By this we mean the need for continuous verification that the whole project is compilable and functions as specified.

The benefits of such an approach are that bugs that are introduced may quickly be revealed and fixed. Otherwise, some bugs might be “hidden” in the program, because the developer only tested that his part of the code functioned, not the whole system. These bugs might then remain hidden until someone tests the whole system, which in worst case scenarios could take weeks or perhaps months. Finding bugs after such long time can then be a very tedious and time consuming job. With continuous integration the majority of such bugs are revealed the same day as they were introduced.

A prerequisite of this approach is to have some sort of single source point. This is because it is time consuming to gather all the code from every developer as soon as an integration test is to be performed. A version control system (VCS) solves this by having all the source code stored and as any changes are made, the developer uploads it to the VCS. In this way, any developer as well as those responsible for the continuous integration always can get the latest version of the code without having to collect parts from different developers.

To improve the concept of continuous integration ever further, Fowler [24] recommends that all parts of it operates automatically. This is indeed possible if the source code is stored centrally (using a VCS), the build process can be made using scripts and automatic tests are developed. With this, it is easy to determine the “health” of the project by simply checking the status of the continuos integration. This leads to the concept of “daily builds”.

### 2.7.1 Daily Builds

Daily builds is a continuous integration technique that originated in the PC industry to get control over the development process [17]. As the name implies, the technique involves regularly building of the code. This process begins from scratch and takes the newest source files from the VCS and tries to build (compile and link) it. For the build to be successful a number of criterions have to be fulfilled:

– All source files that are included have to be present.

– There cannot be any linking errors to other packages.

– The code must be consistent and there cannot be any compilation errors due to e.g. syntax- or semantic errors.

– Some systems also demands other things for the build to be successful, e.g. specialized hardware.

To be able to make daily builds an effective method, the build process has to be relatively fast. This is of course a measure that varies from one project to another, but the key concept is that it is an important factor that must be taken into consideration.

Automatic testing can be seen as the second part of a daily build. The build process can spot many errors, especially if a strongly typed language is used [24]. Nevertheless, to be able to ensure a certain level of quality, testing has to be done. These are generally done automatically as previously described. Often these tests are referred to as “smoke tests” [36], which originates from the time where electrical engineers switched on a device to see if smoke came out from it.

Whenever an error has been spotted during the daily build, the appropriate developer has to be notified of this. By using the principles described earlier, this is automatically done by checking what file the error was in and using the VCS to find out the last person that made changes to that file. Then, for instance an automatic generated e-mail can be sent to that developer.

Another question that arises is how often the daily build should be run? Although the perhaps most obvious answer is “daily”, Fowler [24] recommends it to be run as often as possible. The more often a daily build is run, the less work and effort is needed to fix any problems that come from it. Of course, one limitation is the time needed to build and test the code.

### 2.7.2 Case Study: Microsoft's Daily Build Strategy

Microsoft, as one of the world's largest software developers, has through the years constructed many popular products. Some of the most famous and commonly used are the Windows family. Cusumano et al. studied the developer teams during the engineering of Windows 95 [16] and we will now present how Microsoft applied the daily build concept during this project.

Microsoft's daily build process consisted of several steps. First, the developers checked out a working copy of the code from the VCS. This enabled them to make changes and implement new features in the product. As soon as a new function was added, each developer made a private build that contained this new feature and tested it locally. Whenever this was considered complete, the code was checked back in to the VCS. This process also included an automated regression test to ensure that their changes did not cause errors anywhere else in the code.

Every day, a designated developer created a complete build of the product from the code on the VCS; a daily build. To be able to build the product source code completely, the other developers were forced to check in their piece of code before a specified time every day. Another rule that was applied was that if a developer had caused the build to "break", the errors had to be fixed immediately. To make sure that all versions of the program were verified during this process, one build for each platform and market was done. In this way, Microsoft could develop a large scale product using several small development teams and still having control of the development process.

However, Windows 95 still contained several bugs when released. This points out that it is very hard to verify large software products although a defined testing strategy exists.

### 2.7.3 Current Tools

There exits several tools for conducting daily builds in an automated way. Many of these support not only building the code but also e.g. running unit tests, getting/putting files on a VCS and sending notifications. We will now present some of these tools.

#### Automated Build Studio

Automated Build Studio is a commercial program for automating tasks [15]. One example of this is daily builds. The program supports different VCS such as CVS, Subversion and PVCS as well as different scripting languages. To support automatization, it lets the user write macros that do certain things at a specific time, e.g. build the code every night. This is possible through a support for the most common compilers and build tools.

Automated Build Studio contains a GUI that lets the user create and modify macros as well as tasks. It also contains support for reporting test results to developers as e-mail, ICQ or newsgroup messages.

#### Luntbuild

Luntbuild is a build automation and management tool [4] which in contrast to Automated Build Studio is open source. It has support for multiple VCS and build tools. However, it does not support testing the software, only building it.

Luntbuild is configured through a web interface, which also supports the possibility to see results from the builds and search for previous build runs.

## 2.8 Human Factors

Testing can be a process that is greatly taken into account during the development of software, by e.g. having designated testers and providing tools and systems to support this. However, there are still a lot that can go wrong due to the fact that people often have different opinions and thoughts about testing.

Because conflicts between developers often have the impact of reducing the product quality and work performance, it is of great importance to have a way of managing testing as a part of the development process. Cohen et al. [13] conducted a study among software development teams and categorized problems into three *layers of conflict*: process, people and organization conflicts. We will now investigate these further.

### 2.8.1 Process Conflicts

A frequent problem is that software testing gets too little time in comparison to other development activities. Testing is often pushed into the future and the planned time for this is significantly reduced. The result is often conflicts between programmers and testers, where testers often have to wait until the implementation has finished. Cohen et al. mentions one tester that could not commence his task until the night before the software was meant to be ready.

As time is a requirement to be able to conduct testing in an effective way, it is crucial that testers get designated time resources. Managers also have to avoid "transferring" time from the testing phase to the programming phase. Instead, a common "time pool" can be used and taken from, when someone needs more time.

Due to the differences in responsibility, testers and programmers also often have different focus on their job. Programmers and developers often think more on the technical issues in the design, while testers and managers focuses more on user requirements and how to solve these. This is a natural behavior and does not need to be a problem. However, sometimes it leads to solutions that not everybody agrees are the best.

To avoid different goals among the team members, it has to be clear to everyone where the focus is. From the beginning, group and individual goals need to be clearly defined. This is not only to avoid that conflicts arises, but also to ensure high quality and time efficiency.

### 2.8.2 People Conflicts

Testers and programmers not only perceive the development process differently, they also often have their own mental process and personality attribute. According to Cohen et al, many testers were described as "compulsive" and "very detailed", while the programmers more often were "creative", "temperamental" and "individualistic". This can also become a source of conflicts, especially if the team members do not know each other well enough and the only communication is problem solving related.

To handle this, many organizations encourage social contact between programmers and testers. This can for instance be team building activities and conflict handling sessions. As with most problem solving, a good relationship between the participants increases the chance of success.

It was also noticed that programmers tend to view their code as an extension of themselves, thus taking it personally when someone finds any error in it. This can in some cases start conflicts. For instance, when a test shows a fault in the code, the programmer may accuse the test to be wrong instead of the code. These kinds of conflicts, which are based on different perspectives, can be reduced by job rotation. This means that team members tries each others tasks, thus gaining insight in what the other does.

### 2.8.3 Organization Conflicts

The organization itself can lead to conflicts if the testers and the other developers are not integrated in a suitable way. Programmers can see themselves as being on the top of the chain, thereby having power over testers. During their study, Cohen et al. found that many testers felt they had to struggle to maintain their place relative to the other developers. The managers plays a big role here, they have to support testers in ways that makes them important in the eyes of the developers.

Another solution is to separate the teams physically and thus improving the team spirit. However, the downside of this is reduced possibility for communication between the teams. In most cases, it is easier to ask someone at the other side of the room, than sending an e-mail to someone at the other side of the building.

## 2.9 Summary

A number of aspects in software verification has been discussed, ranging from why testing is important to specific techniques to automatically create tests.

Software verification can be done in two major ways, either through formal methods or through testing. Formal methods specify and verify software functionality by applying techniques from mathematics and logic. This makes it a very theoretical approach.

Testing on the other hand, is a more practical approach since it is more concerned with the actual implementation of a program rather than just underlying theorems. Testing can be categorized in a number of ways; e.g. by level of abstraction (unit, integration and system), static or dynamic, white or black box, automatic or manual. Though, many of the categories blend into each other, for instance it is common to have automatic tests at the unit level.

To make testing feasible in large projects, automating part of or the whole test process and continuous integration has been shown to be efficient. Testing can otherwise be a very time consuming process, as the number of possible errors can be immense.

An aspect that is commonly overlooked is that testing must be incorporated in the development process, so that the developers understand the importance and purpose of it. Otherwise problems and conflicts can arise due to the fact that people often have different opinions and thoughts about testing.

## 2.10 Discussion

Clearly not all methods and techniques are suitable for all kinds of development. They all have certain areas in which they work well and some in which they are not so effective. Following is an approach to which methods we think would be suitable to use at Scania.

When looking at the techniques presented in section 2.1.3 for when testing should be done, it is clear that some of them would not be suitable. For instance, the “code and fix” strategy is simply too primitive to use at Scania, because many developers are involved and the system developed is large and complex.

The waterfall model is also not fitting, because with all testing left to the end of development, critical errors could survive for a long time in the system before discovered. Another risk of saving all testing to the end is that the time originally intended for it may be decreased since the development takes longer then planned.

In the spiral model testing is to some extend incorporated in the development process, where some testing is done on every new prototype developed. But since full unit, integration and system test are left until the full system is completed, problems similar to the waterfall model could occur. We think that especially unit testing should be included at an earlier time in order to assure that the individual units are working satisfactory.

The method we propose as the best fitted is a variant of “agile software development”. With this method, testing is included in the development process all the time (using TDD as proposed in XP the tests are actually written before the code itself). As the testing is conducted at an early time, critical errors can hopefully be discovered early. Even though a special testing phase probably is needed when the system is completed, the major part of the errors has most likely already been found.

We also think that it would be effective to introduce the strategy of “Continuous integration”. This would help to inform the developers of the status of the system at all times. New errors in the system could then be discovered not long after they have been introduced. This makes the debugging of the code much easier; if the code was known to be error free yesterday, the error has to be located in some part that was changed since then.

Another important aspect is regression testing. This is a technique we think could make the development process much more effective, particularly if used at a regular basis as in continuous integration. By doing so, new incorrect code could easily be discovered. We do not see any need to use a regression test case selection technique. This is because the number of test cases at this point would not be that many, meaning that nothing would be gained by a selection of them. This could however be an issue in the future if the number of test cases grows large.

In order to take full use of the continuous integration strategy it clearly needs to be automated. If it is to be done manually, the risk is that it will be forgotten or put aside due to lack of time. The question of how often the testing should be done is dependant on the development process. For instance, if there are only minor code changes once a week, it would probably be unnecessary to run the tests every day.

This concludes the background of software testing and the various techniques and methods that exists. We will now move onto more specific applications used today at Scania.

# Chapter 3

# CURRENT TOOLS AT SCANIA

As we have mentioned throughout this report, testing is necessary to ensure a certain level of quality and performance. This applies to most engineered work and undoubtedly also when developing engine software for trucks. At Scania there exists several techniques for doing this, and we will now look into some of them.

We will first describe two techniques for doing static testing of the code: QA C and Lint. These have different responsibilities are used concurrently during the development and refinement of the engine software.

We will then move onto discussing methods for dynamic testing of the code, using commercial programs and languages. As dynamic testing usually requires more user attention and are harder to do than e.g. syntax checking, these techniques are not as ordinary as the first ones.

Many techniques that is used to verify the engine software have also been left out, as they are not applicable to our work. This includes for instance testing an actual engine (with the engine software) in a controlled lab environment or running the whole truck to see if it behaves as expected.

## 3.1 Static

### 3.1.1 QA C

QA C [43] is a static analyzer for C code, and is designed to help the user write better and safer code. It can report over 1100 potential problems, problems that may not be reported by the compiler. It tries to find code segments that are over complex, non-portable, hard to maintain or which simply diverge from local coding guidelines. It will also issue warnings for code that is not compliant with the ISO C standard.

The reported warnings can be displayed in a number of ways; it can be viewed through a "message browser" which categorizes and groups warning occurrences across all source files. It can also be viewed in the form of annotated source code, either in plain text or in html format with links to additional information and advice. It is also possible to get only the warnings in single files.

QA C can be extended with additional modules, for instance modules that contain industry, company or standards-specific checks. One such module is the *MISRA* compliance module. MISRA, The Motor Industry Software Reliability Association [5], is a collaboration between vehicle manufacturers, component suppliers and engineers which seeks to promote best practice in developing safety related systems in road vehicles and other embedded systems. The MISRA guidelines are now also being adopted in aerospace, telecoms and medical companies as a basis for ensuring the integrity of code. Up to this day MISRA contains only guidelines for C code, but work with MISRA C++ has recently started.

The department of Engine Control System at Scania is currently using QA C with the MISRA module.

### 3.1.2 Lint

Lint is a tool that checks the code and finds e.g. syntax errors, inconsistencies and redundant code [41]. It is capable of reporting over 800 different errors, and many of these are the same as QA C. Another similarity is that Lint follows the ANSI C standard, but also has support for K&R C and ANSI/ISO C++. K&R C is an informal standard that preceded ANSI C and was stated by the authors of the famous “The C Programming Language”, Kernighan and Ritchie.

As previously said, much of the errors generated by Lint are also discovered by QA C, however there exists some features that are worth mentioning:

– *Value tracking.* This technique gains information about variables across statements. This can for instance be checking that an array is not indexed out-of-bounds, when the array itself is declared somewhere else in the code.

– *Strong type checking.* In comparison to e.g. Pascal or Java, C and C++ itself has a weak type check. One example is that all integers can be used as Booleans and any Boolean is of type int. Lint addresses this confusion by clearly distinguish between where to use Booleans and where not.

## 3.2 Dynamic

### 3.2.1 SHTL

SHTL (Scania Heavy Truck Library) is a model library for simulating the complete truck [9]. It is written in the Modelica language and is developed in the Dymola environment.

Modelica is a language for physical modeling. One great advantage to other modeling languages is that it is multi-domain capable. This means that is can be used in several different engineering domains, in contrast to languages that are focused on one special market.

Modelica itself is only a model specification language. To use it, SHTL uses the Dymola application. Dymola transforms the equation representation of the Modelica models into executable code. Dymola also includes a GUI that enables the user to connect models and systems to each other, and configure parameters.

SHTL is a library that consists of models for different parts of the truck. This can for instance be the engine, wheels or the cooling system. It originates from STARS [39], which is a whole vehicle model described in the Modelica language. While STARS is primary used for estimating fuel consumption of a vehicle under certain circumstances, SHTL aims to be more general and flexible. This makes it easier to use SHTL whenever a developer wishes to simulate some parts of or the whole truck concerning some functionality.

SHTL is currently not used by the department of Engine Control System, but there exists a desire to test the engine control system in a simulated truck, during the development. This is mainly because it can be hard to verify certain functionality with other available tests.

### 3.2.2 Frec

Some of Scania's field test trucks are equipped with a flight recorder (FREC), a device similar to the black box in airplanes. It can record various kinds of data from a wide range of sensors. FREC logs the data on flash cards which are regularly taken care of to store the measured data. Each flash card contains text-files which are transformed to .mat-files in order to handle them in MATLAB.

Typically FREC is used when new functionality has been added to the control system. The developers download the new software to the truck and after the truck has been on the road for a day, week or sometimes even months, the recorded data is collected and can be examined. A drawback of FREC is that it can only record a limited number of in-signals at the same time. The developer can however decide *which* signals to record.

### 3.2.3 Simulink

Simulink is a program that runs as a companion to MATLAB. Simulink and MATLAB form a package that serves as a tool for modeling dynamic systems. It provides a graphical user interface (GUI) that is used in building block diagrams, performing simulations, as well as analyzing results. In Simulink, models are hierarchical so you can view a system at a high level and then click to go down through the design levels.

Simulink can also be used to automatically generate C-code from modeled block diagrams. Furthermore, external code can be inserted into Simulink models using so called S-function blocks. External code to be included can be written in MATLAB, C, Fortran or Ada. A special tool is included to aid the user in this process.

Simulink is widely used at Scania, it is used for making models and simulations. It is for instance used together with FREC-data, in a way so that the Simulink model uses FREC data as in-signals.

### 3.2.4 Implementation issues

One part of this master thesis was to investigate how these dynamic testing and simulation techniques could be used to test the engine control system and further how this could be done automatically. Three proposals using the described tools came up, and each will now be presented:

#### Using SHTL

As Dymola has support for importing external code, the engine control system could be connected to a complete truck model. This would enable full scale testing of the complete truck. Furthermore, this option would support interactive control of the truck during the simulation as well as possibility to change conditions, e.g. the road profile.

One possible automatic test using this proposal could be to simulate the truck with the current engine control system driving a predefined road. The test should then observe the output signals from the engine control system to check for abnormalities and errors.

A problem with this approach is that the engine model used in SHTL is too simplified to be used together with the real engine control system. The model does not take nor generate the amount of signals that the engine control system requires to be tested completely. However, there exists one engine model that fulfils this. To be able to use this proposal in practise, the current engine model thus has to be replaced by the more advanced one.

Another problem is that the engine control system itself is not executable on a PC although it is written in a common programming language. This is because it depends on hardware specific components on the physical engine control unit.

Thus, to be able to test the control system together with SHTL, some hardware specific code has to be replaced by e.g. stubs before the simulation.

### Using Simulink

One big advantage with Simulink is its possibility to transform code to s-functions. This is currently used by some developers to test certain parts of the engine control system. It is relatively easy in Simulink to configure input signals and interpret output from the system.

This approach proposes that a major part of the engine control system is transformed into an s-function in Simulink. It is then connected to the advanced engine model described above, which is a simulink model. Testing different functionality can then be done using this set up.

While it is relatively easy to control certain parameters and signals in Simulink, it can become a problem to make this kind of testing automatic. This is mainly because some manual work is needed to transform and connect the two systems. Certain test conditions and expected results must also be defined and controlled to do this.

### Using Simulink and FREC

The third proposal is similar to the second, but instead of using the described engine model, FREC data is used. This makes the testing being based on authentic values instead of simulated ones. The test settings are almost the same as the second proposal; the engine control system is transformed into an s-function and testing occurs in the Simulink environment.

As mentioned, FREC data is not complete as it can not record all signals. This can also become a problem if this proposal is to be used. Another problem is that only non-feedback functionality in the engine control system can be tested. These are functions that do not require any feedback from the engine. This is because the FREC data itself is static and cannot change whatever the engine control system tells the engine to do.

# Chapter 4

# IMPLEMENTATION

## 4.1 Analysis

The idea of this project evolved from some developers at the Department of Engine Control System (NEE) at Scania. NEE develop code for the engine control unit, a unit that controls several engine functions such as fuel injection amount and fan speed. This control system also exists in several different versions, depending on e.g. the engine type. Our work was, as told in the introduction, to design and implement a platform that could test and verify this control system.

From the start, there existed many thoughts about what our system should be capable of doing. This included different kinds of tests, where many were somehow connected to those described in the previous chapter. As mentioned, the idea was also that these tests should be run automatically. As our work progressed, we discovered that many ideas were too hard or time consuming to implement. This was partly because many testing methods were not adapted for testing the whole code, but mainly because it would be very hard to do it automatically.

Apart from this, we needed to design a platform that was not limited to a specific type of test, or even a specific type of tested software. As we did not know exactly which tests or future versions of the engine software that the platform should support, we had to design it as generally as possible. The support for future engine software versions and tests was also a great requirement from Scania.

Therefore, we designed it to be modular, extendable and scalable. The modularity concept is important because it makes the platform clean and makes it easy to change different parts. The platform must also be extendable, as we do not know what new versions of software and tests that it should support. Another important concept is that it must scale. Although it now seems like we only are going to run two or three versions of the engine software, with perhaps four to five different tests on each, the platform should not be limited to this.

The initial phase of the work lead to the following requirement specification, that points out in more detail what the platform should support.

### 4.1.1 Requirement Specification

Following are the summarized features that the platform should support.

– *Incorporate two VCS systems, Subversion and PVCS*

- *Possibility to add new VCS systems:* Since the VCS system used may well change in the future, the platform should have the possibility to add support for others.

- *Incorporate four tests, Q AC, Lint, SS and CUnit:* QA C, Lint and SS are test programs already used at Scania, CUnit is not currently used but we intend to use it for unit testing

- *Possibility to add new tests:* The platform should not be limited to those tests above, but should have the possibility to excute other tests as well.

- *Execute several tests on each project:* There should not be a limit on how many tests that can be executed on a single project.

- *Check out and build several projects:* It should be possible to check out several projects from different VCS systems. By project we mean one version of a software, for instance a specific version of the engine control software.

- *Report test results through email:* When the test result have been interpreted, an e-mail should be sent to the appropriate persons (e.g. the persons responsible for the incorrect source files)

- *Add test results to a database:* Besides sending e-mail, the platform should also report test results to a database. This information is later used by the web site.

- *Never crash because a test crashes:* We can never be sure that a newly created test will function properly, it may for instance run into an infinite loop or even crash. Due to this it is very important that the platform has superior error handling.

- *Configurable through an xml file:* The configuration of which projects to check out, which tests to excute and so on should all be done through a single xml file. Nothing of this should be hard coded.

- *High quality object oriented code:* Since the platform will probably be quite complex and because other developers may extend it in the future, it is important that the code is well designed and easy to understand.

## 4.2 Early Platform Design

Once we had analyzed the problem, taken into account the requirements and the functionality it should support, we decided to do a basic design of the platform. For this we used a top down approach and focused only on high level functionality of the platform, not considering any implementation issues or problems situated at lower levels. During this work we identified five distinct phases that occur in one "run" (one execution of the platform and its incorporated tests). These are illustrated in Fig. 4.1.

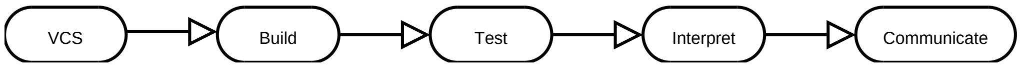


Figure 4.1: Early design and program flow of the platform

1. *VCS:* In this phase all projects that should be tested are checked out from the VCS. As noted earlier the projects can be located at different locations and with different VCS systems. It is also important that *all* files are checked out, not just the modified ones.

2. *Build:* The next step is to build the actual source code. The build should be done from scratch to assure that it is actually compiling all files checked out from the VCS and not just some of them. All warnings / errors that the compiler reports should be saved in a text file, which can be interpreted later.

3. *Test:* Here all tests for all projects get executed. The nature of the tests could differ significantly from each other, some tests may be static other dynamic etc. But they all have to produce some kind of output files, which similar to the build output files can be interpreted later on. At this phase it is important to keep track of which tests to run on which projects. One scenario could for instance be that test A, B and C should be executed on Project 1 and test A, C, D and E should be executed on Project 2.

4. *Interpret:* After all tests have been executed, their output files should be interpreted to find out if the tests failed or not. If a test is to be fully automatic this is a very important step, because without a proper interpreter it is hard to know if a test was successful or not. In some cases the creation of a test result interpreter is quite easy, like for tests that produce a parsable text file with warnings and errors. But in some cases this can be very hard, for instance a test that only produces a graphical plot of some function. When this phase is finished all results that originated from the tests should have been handled.

5. *Communicate:* The last step is to gather all interpreted results and insert it to the database and also send e-mail to responsible persons. The data to be inserted into the database should contain information about how the tests went, which errors were discovered and so on.

The execution of the platform itself will be started automatically, using an external scheduler program. This makes it easy to customize when and how often it should run.

## 4.3 Final Platform Design

When the early design of the platform was done, we started to look deeper into the functionality of each of the five phases identified. This work brought up problems not thought of before and resulted in some changes of the design and addition of new functionality. Our work continued with the identification of specific classes, objects and methods. State flow diagrams were made and different scenarios were created and tested to assure that the platform met its requirements. The resulting final design differs some from our early thoughts but the program flow is almost still the same.

Following are a more thorough explanation of the final platform. Since a run of the platform is very sequential, we have chosen to give the explanation in the same sequential way, describing modules one at a time in the same order as they gets executed.

Before we start describing the separate modules we will give a description of some common design patterns that are used by many of the modules. After all involved modules have been described, a summary and an overview of the complete platform will be given.

### 4.3.1 Design Patterns Used

To make the platform meet the requirements, we used a number of *design patterns* during the implementation. For those of you not familiar with these, a short description now follows of those used.

Design patterns describe simple and elegant solutions to recurring design problems in object oriented systems [18]. They make it easier to reuse successful designs and architectures. A design pattern has in general four elements:

1. The *pattern name* that describes the essential idea and makes it easy to communicate the solution between developers.

2. The *problem* describes when to apply the pattern. It could for instance be certain preconditions or contexts.

3. The *solution* explains the elements that are included in the design pattern. The solution is not a concrete implementation but more a template of how to solve the problem.

4. The *consequences* are the expected outcome when using the design pattern. It also describes trade-offs that are necessary when using the pattern.

#### The Facade pattern

The intent of the Facade pattern is to provide a unified interface to a set of interfaces in a subsystem [18]. In other words, it reduces the complexity and coupling between two subsystems by providing a simple interface that they can operate on. In this way, a subsystem that wants to use another subsystem needs only to know its facade. It does not have to bother about underlying details.

Our platform uses a modified version of the Facade pattern in a way we have chosen to call *plugins*. All things that can change, e.g. VCS and tests, are implemented in the platform as plugins. In this way it becomes easy to anyone that wants to add a new VCS or test as they only needs to know the facade for that part of the platform.

#### The Factory pattern

One recurring problem is having several concrete classes that represent modifications of the same thing. It is then often unwanted to choose and create these concrete classes directly from the class that is going to use it, because of relation and collaboration issues. To address this, the Factory pattern gives a way to encapsulate the instantiations of concrete classes [22]. This is done by inserting that actual choice and instantiation in a separate class that is called a factory. The class that is going to use the chosen class is thus not aware of which concrete class that is being used.

The platform uses the Factory pattern when choosing and instantiating concrete tests and VCS. It also uses a factory when parsing the configuration file. This also makes the process of e.g. adding a new test easier, as no code in the actual platform have to be changed. The only thing that has to be done is to add a new instantiation in the corresponding factory.

### The Visitor pattern

The intent of the Visitor pattern is to provide an effective way of performing operations on the elements on an object structure [18]. New operations can be added to the visitor instead of changing the element on which it operates. The Visitor pattern is useful when an object structure contains different classes, and the actual operations depend on their concrete classes.

We use the Visitor pattern when traversing a tree that represents which tests that are going to be executed. As we further will explain in the next subchapter, that tree contains different types of nodes. Furthermore, we use different types of visitors as well, depending on the current task.

## 4.3.2 ConfigReader

The first step in the execution of the platform is to configure it. The platform needs to know not only what projects and tests to execute, but also several parameters and what VCS systems to use. All this is done with a configuration file written in XML. The ConfigReader module is responsible for parsing the configuration file and making the information available for the rest of the platform.

To store the configuration settings internally we use a tree structure. This was done because of the nature of the configuration. As well as nodes in a tree can have several children, any project can have several tests (that should be executed on it). Furthermore, several VCS systems can exist, which may contain several projects. The tree is hence built up from the configuration file and consists of the following nodes: VCS, Project and Test nodes, where VCS and Project nodes are allowed to have children. We also included the possibility to add Config nodes, which could contain some settings that should apply to several children. However, Config nodes are not currently used and will not be further discussed, as we added them for possible future needs.

Similarly, the configuration file is built up with the corresponding XML tags. To illustrate this, we will now give an example configuration file:

```
<CONFIG>
  <TESTSTRUCTURE>
     <VCS Value="PVCS" RepURL="pvcs://computer1/">
        <PROJECT Value="myProject1"  ProjURL="pathToProject1">
           <TEST Name="myTest1" TimeOut="5000"></TEST>
           <TEST Name="myTest2" TimeOut="3000"></TEST>
        </PROJECT>
     </VCS>
     <VCS Value="Subversion" RepURL="svn://computer2/">
        <PROJECT Value="myProject2"  ProjURL="pathToProject2">
           <TEST Name="myTest1" TimeOut="5000"></TEST>
           <TEST Name="myTest2" TimeOut="3000"></TEST>
        </PROJECT>
        <PROJECT Value="myProject3"  ProjURL="pathToProject3">
           <TEST Name="myTest3" TimeOut="5000"></TEST>
        </PROJECT>
     </VCS>
  </TESTSTRUCTURE>
```

```
   <GENERAL>
      <BASEPATH>C:\TEMP\ATF</BASEPATH>
   </GENERAL>
</CONFIG>
```

There are several things to note here. First, there are three different sections in the file:

– CONFIG, contains all configuration settings.

– TESTSTRUCTURE, contains the configuration of which VCS, projects and tests to run.

– GENERAL, contains general configuration settings. In this example, BASEPATH is an absolute path to the folder where all output (checked out files, test result files and log files) from the platform should be.

Second, there are several parameters to some of the tags. For instance, the TEST tag takes the TimeOut parameter that states, in seconds, how long time the test has to complete. Also, the PROJECT tag takes the ProjURL parameter that shows the path on the VCS where that projects files are.

Note that this example does not show nor explain all required parameters. For a complete guide to how the configuration file works, refer to appendix C.1.

The configuration file is parsed using Java's XML parser and DOM (Document Object Model). As mentioned, the ConfigReader module builds a tree that represents this file. The example file showed above would generate the tree showed in Fig. 4.2.

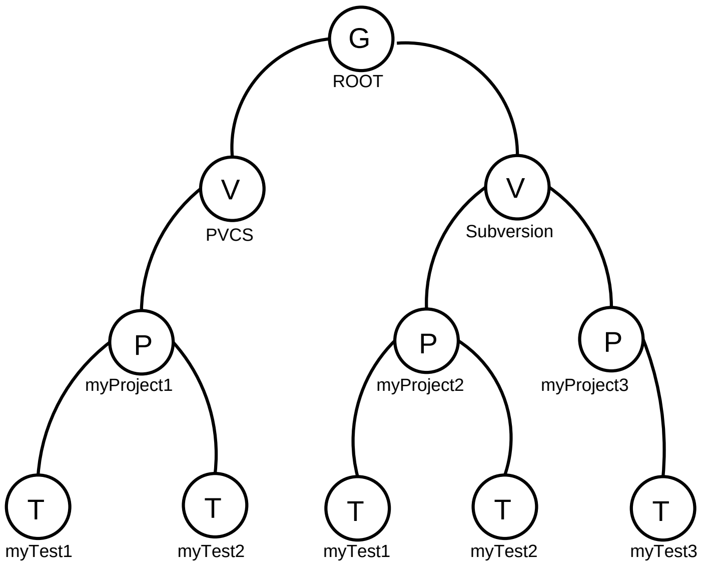


Figure 4.2: The generated tree from the example configuration file

The different node types are defined in an inheritance hierarchy. A test node (denoted by T) inherits from simple node (not shown). Furthermore, a group node (G) also inherits from simple node. Both VCS nodes (V) and project nodes (P) inherit from group node and can thus have children. See appendix B for further details.

With the tree built up, everything is set up for the VCS module to begin checking out source files. We will now move onto describing that module, where we also will explain how we use the configuration tree.

### 4.3.3 VCS

The VCS-module handles all communication with the involved VCS. It has functions for checking out different projects, obtain the person who last checked in a specific file etc. It is used by other modules as soon as VCS related information is needed. For a complete overview of all classes see appendix B.

#### Diverse VCS systems

Since different VCS have different ways of checking out projects etc, there is no way to write a general class for this. Instead specific code has to be written for each VCS. To organize this, all code that handles the communication with a specific VCS system are contained in separate plugins.

Because of the requirements on the VCS module, the underlying plugins must implement a specific interface. The interface has three defined methods:

- Check out a project
- Get the responsible developer for a file
- Get the date for when a file was last checked in

It is the plugin's responsibility to implement these methods correctly.

#### Traversing the tree

As mentioned earlier, information regarding which projects to check out is contained in the configuration file, which has been transformed into a corresponding tree structure by the ConfigReader module. In order for the VCS module to find out which projects to check out it needs to traverse this tree structure and collect the information needed.

The traversing is done in a depth-first order using a "visitor" to perform the required tasks at each node, as described in section 4.3.1. A description of the VCS-visitor's behavior at the relevant nodes will now be given:

- *VCS-node:* Retrieve information about which VCS is used (Suberversion, PVCS etc.), the URL to it and which user name and password that is required for access. After it has got this information it can create the correct VCS plugin using the factory pattern. When this is done it moves on to the VCS node's children.
- *Project-node:* Get the projects relative URL and have the created VCS plugin check out this project. After the project is checked out there is no need to traverse further down this branch in the tree (only test-specific nodes below), so the visitor simply returns. If the plugin for some reason should fail to checkout a project (e.g. due to network problems), the corresponding project node will be marked as failed, so that no tests will be conducted on that project.

When all projects have been checked out the VCS module's job is completed for now. All source code should now be available locally and ready to execute tests on.

### 4.3.4 TestExecutor

This module is as the name implies responsible for executing all tests. Besides this it should also take care of every test's output files.

#### Tests

The tests to execute can differ very much from another; one test could be a simple syntax checking test while another could be of a more dynamic and complex nature. This resembles the case of the VCS module where different VCS had to be handled. Our solution to this is therefore the same as in the VCS module: every specific test execution is contained in a separate plugin. Most commonly the plugins execute an external test, meaning that the test itself is not located in the plugin.

In the VCS module the plugins were forced to implement a certain interface, in this case they must inherit from an abstract test plugin base class. This abstract class has implemented methods for returning which project the test should run on, where the source code is located etc. In addition to these implemented methods there are two left for the creator of the plugin to implement. The first one is the actual execution of the test and the second one is for returning the output files from the test that is needed later when interpreting the result.

#### Build

In the early design (see 4.2) we had a separate phase, or module, for building the code which was conducted before the tests were executed. This is something that changed in the final design. When we started to think more about the building process it came to our conclusions that it can be seen as just another test, ”the test of building the code”. Therefore instead of having an own module for this, it is now included as a separate test plugin. The building process redirects its errors and warnings to a text file, which is later interpreted.

#### Error handling

There is no way to guarantee that the execution of a test will behave correctly, it may for instance run into an infinite loop or it may simply crash. Since we don't want the platform itself to crash or get halted by a non terminating test, we need some way to handle this. To cope with this we start every test in a new thread, separate from the one the platform is running in. Furthermore, every test has a maximum execution time. If this time is exceeded and the test has still not terminated the test is marked as a failure and is forced to terminate.

#### Traversing the tree

In order to find out which tests to run, the tree structure must be consulted. A test executor visitor is used for the traversing and performing various tasks at the nodes.

- *VCS-node:* Record which VCS system that was used, then move on to its children.
- *Project-node:* Record which project the node corresponds to, then traverse deeper in the tree.

– *Test-node:* This is obviously the most important node for this visitor. Once it has reached a node of this type it has knowledge of what project the test belongs to and where the projects source code is located. Using a factory it can now create the plugin associated with this test.

  Before the plugin is started the visitor needs to tell it what project it belongs to and where the source code (that should be tested) is located. After the plugin has received this information, it is started in a separate thread.

After the visitor has traversed the complete tree, all tests should have been executed. So far the only thing we know is if the test terminated correctly or not. We still have no information of the actual results from the tests, (did they report any errors in the code?). What we do have are the tests output files, but it is not part of this modules responsibility to look into these, instead this is where the Interpreter module takes over.

### 4.3.5 TestInterpreter

With the tests executed and done, it is time for the platform to interpret the information that they resulted in. This is the job of the TestInterpreter. Structurally, it is very similar to the TestExecutor module, as we soon shall see.

#### Test output

The vast majority of tests are external, meaning that they are separate programs that get executed by the test plugins. But in the same way as tests can differ very much from each other, their output can also vary a lot. One test can for instance just generate a file with numbers while another generates plots or charts. This fact made us use the plugin pattern again, in the same way as with the execution of tests.

As mentioned, whenever a developer wants to add a new test, a new test execution plugin must be written and deployed. In addition to this, a test interpreter plugin must also be constructed. Often this interpreter is a text parser that parses error messages from files that have been created by a specific test. There is however a possibility to use the same test interpreter for several tests. This can for instance be an XML parser that recognizes some agreed format.

Similar to the TestExecutor module, every new interpreter must implement an abstract class. Just one method has to be implemented; the one that starts the actual interpretation. This method also takes the file paths to all the output files that have been created by the test.

To report that an error or warning has been found, a method in the abstract class is called. There are four arguments (all optional) that are passed with this method call:

– Error message: Any message that should be reported for this error. In most cases, this is parsed from the output file.

– Error degree: A degree that represents the seriousness of this error. There are a number of predefined degrees that may be used, to ensure the possibility for comparison between tests.

– Source file name: Often the output files also contain the source file that generated that error. For instance, most syntax checking tests report in which file they found an error. This filename can then be parsed by the interpreter and passed as an argument.

– Output file name: This is the output file that has been interpreted to generate this error. It is reported to make it possible to check the original output file later.

As we shall see, the third argument is very important to make us structure the reported errors and make developers responsible for the errors generated.

### Result structure

Internally, we use one class that represents a source file and another that represents a notification (could be an error, a warning or anything else that comes from the interpreter). When an error is reported, a notification object is created. If the source file name argument is set, a corresponding source file object is created. But this only occurs if an object for this source file name has not already been created.

Also, source file objects have references to every notification object that originated from that source file. In this way, it is easy to check what errors a particular source file has.

However, not all notification objects have a corresponding source file. This is because some errors and warnings are too general to apply to just one source file, or it might be hard to know in what source file the error originated from. Imagine for instance a test that runs a particular piece of software and a *segmentation fault* occurs. In most cases, it is impossible to know where the fault is. So the interpreter can only report that a segmentation fault occurred, not in what source file the error is.

### Error handling

Similar to the TestExecutor module, every test interpreter runs in a separate thread. This is again to ensure that the platform does not crash or halts, if a "third party" interpreter fails. Every plugin also have a timeout time that must not be exceeded, which is the same as for the test execution.

If any fault is discovered or if the interpreting time exceeds the limit, the test is marked as a failure (as we cannot guarantee that the output is complete) and the platform moves on.

### Traversing the tree

To find out which tests to interpret, the tree structure must again be consulted. A test output visitor is used to traverse the tree and perform specific tasks at the nodes.

– *VCS-node:* Record which VCS system that was used, then move on to its children.

– *Project-node:* Record which project the node corresponds to, then traverse deeper in the tree.

– *Test-node:* Again, this is the most important node for this visitor. Once it has reached a node of this type it knows that the corresponding test has been performed by the TestExecutor module, assumed that the node has been marked a success. It also knows where that test's output files are on the file system. Using a factory it can now create the correct interpreter plugin. The plugin is then started using the method call described above, passing an array with all output file paths.

  The visitor then waits either for the interpreter to finish, or until the maximum execution time has been exceeded. If it succeeds, the error reporting method in the abstract class has most probably been called several times during the interpretation. All this information has also been stored in a linked structure as described above. This structure is then passed on to the InformationHandler module, which we will now discuss.

### 4.3.6 InformationHandler

The InformationHandler module is a module that complements and organize the notifications from the TestInterpreter. As mentioned, the result from the latter module is a linked structure.

#### The work of the InformationHandler

One requirement was that we wanted to find the developer that made the errors that have been reported. We also want to know the latest time when a particular source file was sent to the VCS. Therefore, this module iterates the linked list of source files and asks the VCS module to get the last developer that made changes to this source file and when that occurred. The timestamp is then stored in the source file object. But, as a particular developer can “have” many source files, it would be redundant to also store the developer in the source file objects. Thus, we use a third linked list: a developer list. As a developer name is returned from the VCS module, a search is commenced in the developer list. If the developer is found, a reference to the particular source file object is added to that developer object, otherwise a new developer object is created and the reference is added.

The outcome when the entire source file list has been iterated is a structure that contains developer, source file and notification objects. It is now easy to get what source files a particular developer recently made changes to and further the notifications that came from those files. Yet again, those notification objects that could not be connected to a particular source file cannot either be connected to a particular developer.

Figure 4.3 shows an example of this structure. A developer can have several source files and a source file can have several notification objects. Note also that all notification objects are not connected to a source file.

With the structure complemented, the work of the InformationHandler is done. The structure is now passed on to the Notifier module and later the Database module.

### 4.3.7 Notifier

The Notifier is a module that notifies developers that they have errors in their files. This is done by sending an email with a summary of the files where errors have been present.

The work of the Notifier becomes relatively simple as a linked list with all developers already exists, as previously talked about. The module iterates this list, and checks what source files a particular developer object references. It then composes a mail message that is sends to that developer using a SMTP connection.

A requirement that appeared during the development was that some developers should never receive email from the platform due to different reasons. This can be configured though an ignore list in the configuration file discussed in the ConfigReader section (see 4.3.2). The Notifier simply checks, for every developer, if that developer exists on the ignore list and in that case, does not send anything.

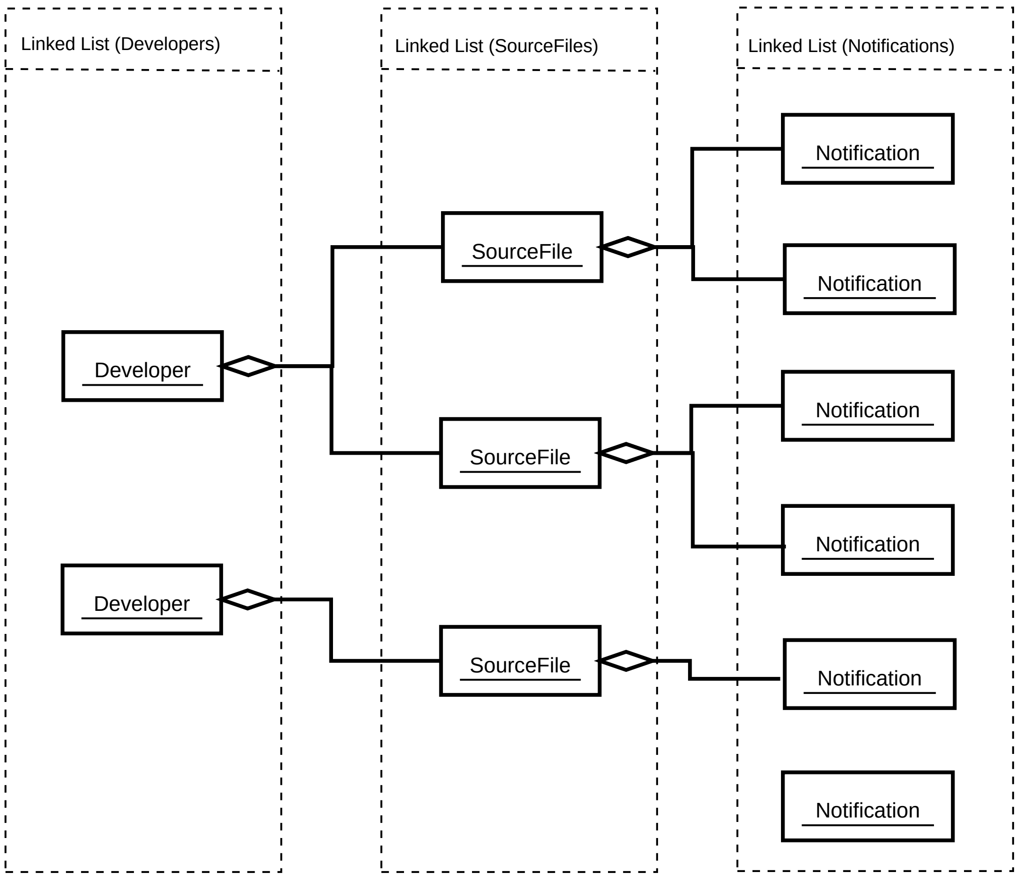


Figure 4.3: The internal structure of the interpreted results at the InformationHandler.

### 4.3.8 Database

This is the last module that gets executed in the platform. When reaching this phase all projects have been checked out, every test has been executed, results are interpreted and responsible developers have been found. What still remain though is to insert the result into a database. Without this step, all interpreted data would be lost by the end of a run.

#### Inserting data

The data that is to be inserted is found in the three linked lists described earlier. Those lists are available for the database module use. For a detailed database schema description see 4.5. The insertion of the data can be described as:

```
1 For all developers
   1.1 Insert the developer into the database
   1.2 For all source files this developer has references to:
      1.2.1 Insert the source file into the database
```

```
        1.2.2 For all notifications this source file has references to:
          1.2.2.1 Insert the notification into the database and mark notification as
                  inserted
2 For all notifications
    2.1 If not already inserted, insert it
```

But one thing is still missing; there is only information about tests that actually reported any errors. Imagine a test, $A$, that has been executed on two different projects. The test executed properly but did not report any errors in either of the two projects. Hence, no information about the test can be found in the lists. This may not seem like a big problem, as missing information about a test would then simply mean that it did not report any errors.

Imagine another test $B$ that also was executed on the same projects. But this test ran into problems during execution and crashed before it was completed. As with test $A$, this would lead to no reported errors for test $B$. However the reason for not reporting any errors are completely different in test $A$ and $B$, but this would not be distinguishable when looking at the information in the database. Clearly, some information concerning the execution of the tests themselves needs to be inserted.

While the lists does not contain any information about this, the tree does. As mentioned before, the test nodes get marked as failed if the test fails. This is also true for the project nodes (if a project was not successfully checked out). Therefore we can traverse the tree again and check the status of the nodes. Like in the other modules a visitor is used for traversing the tree.

**Traversing the tree**

The following tasks are conducted by the database visitor:

– *Project-node:* Retrieve status about the node (failure or success) and insert this into the database.

– *Test-node:* Retrieve status about the node (failure or success) and insert this into the database.

When all data has been inserted to the database, the module has completed its work. This also marks the end of a run in the platform.

Besides the seven modules described so far, the platform consists of two "help modules" for completing its task. Those modules are not executed in the same sequential way as the other seven, but are instead used by them when needed. A description of them follows in the next chapter.

### 4.3.9 Additional Modules

**FileSystemHandler**

Many of the previously described modules need file handling of some sort. For instance the VCS module must know where to check out the projects, the test executor module needs to copy test output files and the test interpreter module must be able to retrieve those files. All functions related to file handling has been gathered in this single module. It is made up by one single static class, so that all other classes can use it directly. It has functions for creating folders, copying files, searching for files and many more.

#### Log

This module is responsible for writing platform related information to a log file. It could for instance be at what time a certain test started and finished or which projects that were checked out. If a module runs into some problem (e.g. a failed test), it can report this to the log module which keeps track of all reported warnings. When a platform run has finished, this module creates a summary of the complete run with the reported warnings.

### 4.3.10 Summary

In fig. 4.4 the platform is shown with the earlier described modules integrated. The program flow of a complete run can be summarized in the following seven steps:

1. The Manager asks the ConfigReader to parse the xml file and build the corresponding tree structure.
2. The Manager asks the VCS to check out all projects.
3. The TestExecutor module is told by the Manager to start executing all tests.
4. OutputHandler is started by the Manager. It tells the TestInterpreter module to start interpreting the test results.
5. The OutputHandler passes the data returned from the TestInterpreter to the InformationHandler, which completes this list with the corresponding developers.
6. The list is passed on to the Notifier which sends an email to the responsible developers.
7. Finally, the list is passed to the Database module, which inserts the data into the database.

Note that in step 2,3,4 and 7, the modules need to traverse the tree to get the necessary information, e.g. location of source code and which tests to run.

## 4.4 Implemented plugins

For the platform to actually do anything it needs to have implemented plugins. First of all it needs a VCS plugin to be able to check out the source code and second it needs at least one test plugin (executor and interpreter) to do tests on the code.

Two VCS plugins (Subversion and PVCS) have been implemented, i.e. the platform can communicate with two different VCS systems.

A total number of five test plugins were also implemented. The first two has been mentioned earlier, namely QAC and Lint (see section 3.1). The third is the actual building of the code and the fourth is special test developed at Scania, used to check for inappropriate cycles in the code.

The last implemented test plugin is an approach to introduce unit testing. For this, CUnit was used (CUnit is part of the xUnit family as described in section 2.3.1). The reason for choosing CUnit is because it has a nice way of organizing individual tests into suits, and it can present the summarized test results as an xml file. By having the results in an xml-file it is easy to create a reliable interpreter.

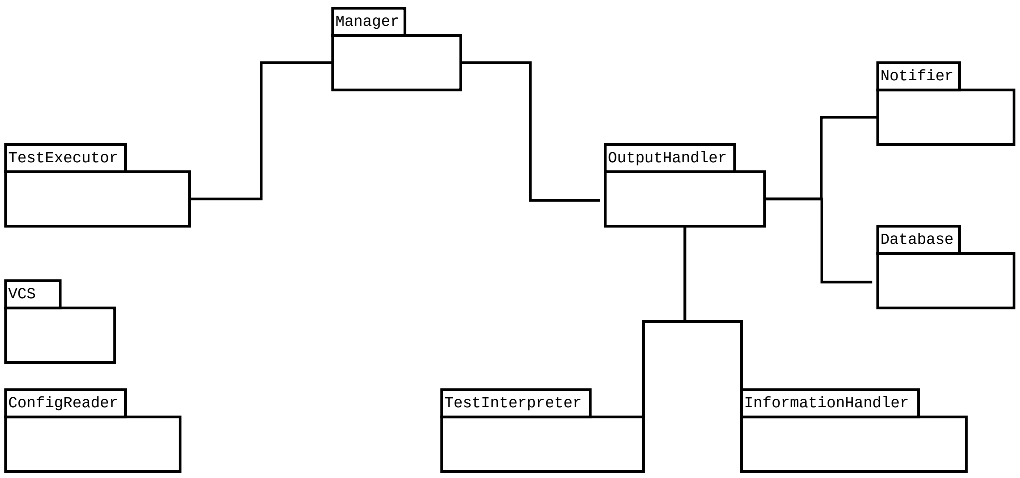


Figure 4.4: The platforms's modules integrated.

The test-executor plugin has to compile the test file together with the corresponding source files (where the functions to be tested are located). If the compilation succeeds the plugin executes the generated CUnit program, which in turn produces the resulting xml file.

## 4.5 Database

To present the results the platform reports its interpreted data to a database which is used by the web site. The database consists of seven different tables which we have designed so that they will fulfill the requirements of Boyce-Codd Normal Form (BCNF). For a database to accomplish this, it has to meet up with a number of conditions and tests. We will not give a further explanation of them because it is beyond the scope of this thesis. For a detailed description of BCNF see [21].

### 4.5.1 Incorporated Tables

A description of each table will now be given. Unique identifiers (primary keys) are underlined.

#### Run

This table keeps a record of data related to a specific run (one execution of the platform). It has fields for the time and date of the run and the path to the log and xml files.

| Run | | | | | |
|---|---|---|---|---|---|
| run_id | run_date | run_time | xml_file | log | log_summary |

#### Developer

This represents a single developer with its unique id, first name and last name.

| Developer | | |
|---|---|---|
| sss_id | first_name | last_name |

### Sourcefile

A table containing all source files in which there have been errors at some run. The file name together with the project is the primary key. The last column specifies if e-mail should be sent when errors in this file are found.

| Sourcefile | | |
|---|---|---|
| file_name | project | notify |

### Project_history

Contains information about which projects that were tested at a specific run. It also includes the success of failure or the project (failure of a project means that it was not checked out correctly). The *run_id* column is a foreign key to the *run* table.

| Project_history | | | |
|---|---|---|---|
| ph_id | project | run_id | p_succeeded |

### Sourcefile_history

This table keeps track of which source files that had errors in a specific run and which developer that was responsible for the file at that time. The column p*h_id*, which is a foreign key to the *project_history* table, together with *file_name* makes it possible to find a specific file at a certain run.

| Sourcefile_history | | | | | |
|---|---|---|---|---|---|
| sfh_id | ph_id | sss_id | file_name | sfh_date | sfh_time |

### Test_history

Has information about the success or failure of a certain test for a specific project.

| Test_history | | | |
|---|---|---|---|
| th_id | test | ph_id | t_succeeded |

### Notification

This table has information about the actual warnings or errors messages that a test produces. The column *sfh_id* is a foreign key to *sourcefile_history* table, making it possible to see which file the message belongs to, if any. *Th_id* is a foreign key to the *test_history* table, which is used to identify which test produced this message.

| Notification | | | | | |
|---|---|---|---|---|---|
| not_id | message | sfh_id | output_file | error_degree | th_id |

## 4.6 Web Site

### 4.6.1 Requirement Specification

Following are the summarized features that the web site should include.

– *Present an overview of the latest test results*

– *Maintain history of earlier runs:* It should be possible to look at test results from not just the most recent run, but also earlier runs.

– *Present which files a specific developer has errors in:* The web site should provide a way for a specific developer to see just his errors.

– *Show all errors for a specific project:* A page presenting all errors related to a specific project.

– *Show all errors for a specific test in a specific project:* A page presenting all errors related to a single test in a specific project.

– *Provide a link to all output files for a specific test:* The user should have access to the output files a test produces (i.e. the test results).

– *Choose files for which a developer does not wish to receive email notifications:* The user should have a choice of which files he wants to receive e-mail notifications for.

– *Provide an administrator page where information on the latest run can be found:* Besides the pages giving information about the tests, there should be a page presenting platform related information. This could be the platform's current configuration (xml file), warnings about tests that could not run properly or possible problems with e-mail sending.

– *Search for a specific file:* The user should have the option to search for a specific file and receive information about it.

### 4.6.2 Design

The web site was created with one focus primary in mind; to make developers aware of problems and encourage them to fix this. This idea is mainly incorporated by making the web page personal, that is, every developer logs in to the site and is presented the results related to that particular person.

A presententation of some aspects with the design of the web page hereby follows.

#### Structure

As mentioned, the first page of the site is a login page. Here, the user enters the user id, an unique six character employee code that every developer has. This leads to the main structure which consists of four choices. These links are situated in a separate frame that is always visible and available. The four choices are:

1. *My notifications* This is the default page that appears when the user logs in. Here, a list of all source files that had notifications in them and was changed by the user are presented. A notification is defined as all kinds of errors, warnings or

information, i.e. everything that is reported by an interpreter. For each file, we also present a history. This is a description of how the number of notifications has changed during the previous test runs. Every item in the history is also clickable; showing all individual notifications for that test run together with the current run, for easy comparison.

The mentioned list is called the active list, but the user can also choose to show a list called the passive list. This list shows the files that the user once had notifications in, but currently someone else has checked in, making the file appear in the other developer's active list. In this way, a developer is not free of responsibility as soon as someone else makes changes to the files. As it is hard to track exactly what errors a certain developer has introduced and when these are fixed, this is a way of utilising common responsibility. For an example screenshot of the web page, see figure 4.5.

2. *All notifications* This page is more of an overview of the project. First, a listing of the total number of notifications in all projects and tests is shown. The user can then choose to see the files that had notifications by clicking on one test. This will show a list in the same way as the active list. However here all files are shown, not only the ones the user had made changes to. Those notifications that are not connected with a source file (as talked about in section 4.3.6) are also presented here.

3. *Configure* This section can be used whenever a developer wants to configure something in the platform. The developer can here choose to not receive email when notifications appear in a certain source file.

   Although this is currently the only configurable thing, this may be a section that expands in the future. For instance, it would be useful to be able to configure the actual tests that are performed.

4. *Help* This option shows a FAQ.

#### Requirements revisited

Almost all functions specified in the requirement specification have been implemented. One thing that has not is the possibility to search files. This is a function that can be hard and time consuming to implement. Besides, most internet browsers have built in support for searching web pages, and we found that to be sufficient.

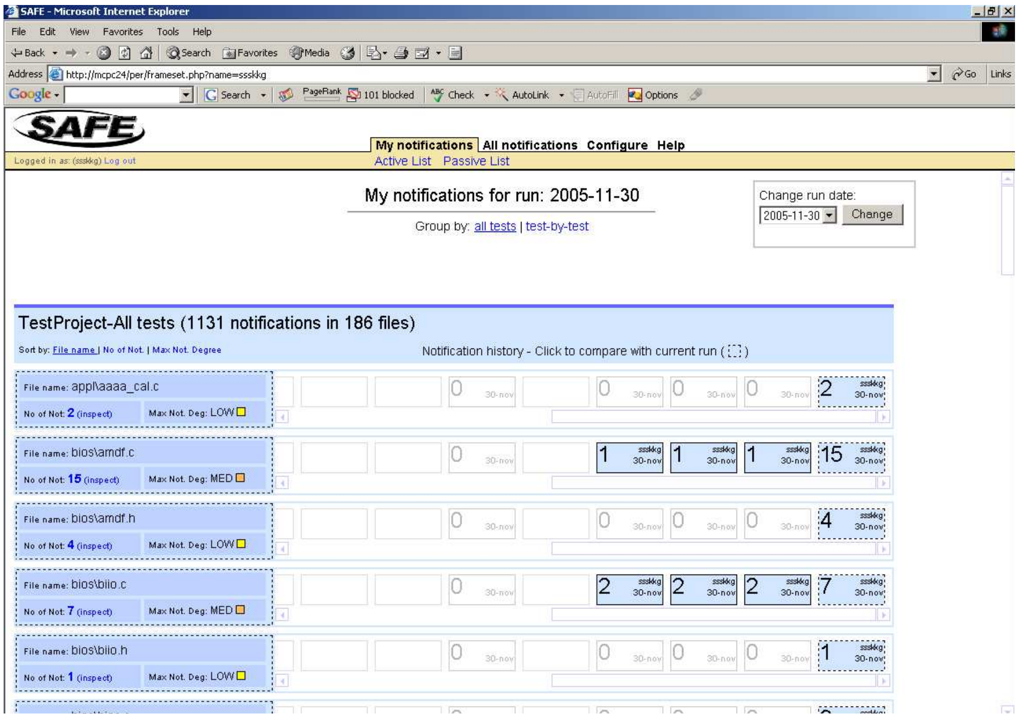
## 4.7 Tools Used

### 4.7.1 Languages

As the work consisted of several systems; a platform that excuted the tests and a connection through a database to a web site, several languages had to be used during the implementation phase. The following were used:

#### Platform: Java

As the problem was analysed, it was realised that with the demands on modularity and extendibility an object oriented language was preferable. Among these, Java was the


SAFE - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://mcpc24/per/frameset.php?name=ssskkg
Go Links
Google Search PageRank 101 blocked Check AutoLink AutoFill Options
SAFE
My notifications All notifications Configure Help
Active List Passive List
Logged in as: (ssskkg) Log out
Change run date:
2005-11-30
Change

My notifications for run: 2005-11-30

Group by: all tests | test-by-test

TestProject-All tests (1131 notifications in 186 files)

Sort by: File name | No of Not. | Max Not. Degree

Notification history - Click to compare with current run ( ⬚ )

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| File name: appl\aaaa_cal.c<br>No of Not: 2 (inspect) Max Not. Deg: LOW 🟨 | | | | 0 30-nov | | 0 30-nov | 0 30-nov | 0 30-nov | 2 ssskkg 30-nov |
| File name: bios\amdf.c<br>No of Not: 15 (inspect) Max Not. Deg: MED 🟧 | | | | 0 30-nov | | 1 ssskkg 30-nov | 1 ssskkg 30-nov | 1 ssskkg 30-nov | 15 ssskkg 30-nov |
| File name: bios\amdf.h<br>No of Not: 4 (inspect) Max Not. Deg: LOW 🟨 | | | | 0 30-nov | | 0 30-nov | 0 30-nov | 0 30-nov | 4 ssskkg 30-nov |
| File name: bios\biio.c<br>No of Not: 7 (inspect) Max Not. Deg: MED 🟧 | | | | 0 30-nov | | 2 ssskkg 30-nov | 2 ssskkg 30-nov | 2 ssskkg 30-nov | 7 ssskkg 30-nov |
| File name: bios\biio.h<br>No of Not: 1 (inspect) Max Not. Deg: LOW 🟨 | | | | 0 30-nov | | 0 30-nov | 0 30-nov | 0 30-nov | 1 ssskkg 30-nov |

Figure 4.5: Screenshot showing the active list.

one that suited our requirements best. Java has many classes for file handling, error handling, file parsing and database connection. All these were functions known to be of value.

One factor that also has to be taking into consideration is performance. C++ generally is faster then Java e.g. due to Java's virtual machine. However, this became a less important problem since the system can be run during the night.

Java also has great support for XML (eXtensible Markup Language), which was the format intended to use for configuration.

### Configuration: XML

As mentioned, the configuration of which tests to execute on which project is done in a XML file. XML was chosen because it is extensible and standardised. Furthermore, the XML standard is strongly typed, which decreases the risk of errors in the configuration file.

### Server side scripting: PHP

The web site had to get most of its information from the database, and therefore be very dynamic. PHP (PHP: Hypertext Preprocessor) was chosen for the server side scripting and database access. This was mainly because PHP is free and open source, but also because it is easy to write and understand.

#### Web page: HTML with CSS

The actual web page is built with HTML (HyperText Markup Language) and CSS (Cascading Style Sheets). CSS is used for most styling issues as it is more flexible then only using HTML.

### 4.7.2 Applications

During the work different applications were used. A short description of the most important now follows.

#### Database: MySQL

To pass information from the platform to the web page (and back) a MySQL database was used. MySQL is free and suited the requirements well. To insert and get information, both the platform and the web page naturally use SQL (Structured Query Language) queries on the database.

#### Web server: Apache

The web server application is Apache that has been configured to support MySQL and PHP. Apache is also freeware.

#### Development: Eclipse

Eclipse is a free development environment that was used for all coding. An advantage of Eclipse is its great support for Java development.

#### Development: Subversion

Not only was support for Subversion implemented in the platform, it was also used for keeping track of the source code during the development.

# Chapter 5

# DISCUSSION

## 5.1 Conclusions

The result of this master thesis is an investigation of techniques for verifying software, both at Scania and in the software development community. Furthermore, a system capable of conducting automatic testing on code and report the results to the developers, have been implemented and integrated into the development process. The system itself has been named SAFE (Scania automated code-verification system), which points out its main purpose.

The success of SAFE depends on how well it manages to support a reduction of the errors in the code. As the system is just completed and has not been thoroughly incorporated yet, this is a hard question to answer right now. By looking at the requirements, we can however evaluate the system and predict how well it is going to achieve.

The demands on an extendable system have been met, as we used plugins for most parts that may be changed in the future. We have also implemented an easy way to communicate the results to the developers, via the web site. The platform can also be automatically executed, as no observation or managing is needed.

However, SAFE is no better than the tests that it executes. This brings us also to the question of reliability of the reported errors. As the results are presented on the web site file by file, it can be easy to think that this is the absolute truth; that files not present does not contain any errors and files that are, only contains the number of errors indicated. The reliability of this depends merely on the accuracy of the tests. As we previously have mentioned, it is in most cases impossible to know when all errors have been found and this is something that the users of SAFE (and most other testing tools) have to bare in mind.

Another matter that should be mentioned is the approach to use external tests. It makes the integration of new tests easy, but the communication between the actual test program and SAFE is not optimal, as it goes via a text file. An important thing to note here is that this not a limitation in SAFE, but more a necessary approach to avoid changing the test programs.

During the development of the system there have also been some noticeable issues, which we now will talk about.

The actual engine control system is highly secret. Therefore we could not at any point during the development of SAFE run it on the complete source code. We did however have access to a static, lightweight version that did not contain the most secret modules. This fact should, on the other hand, not be a large issue since the design should hold for basically any code.

The time available for completing the master thesis work has been sufficient. We have tried to maintain a time plan throughout the work and it have showed to be very useful.

Hopefully SAFE will contribute to an improvement of the enginge software in Scania trucks, see fig. 5.1

Figure 5.1: A 470hp R-series truck from Scania and the authors

## 5.2 Limitations

The platform and the web site have a number of known limitations. Some of them are due to lack of time and others because no solution was found for them. Following are the limitations that have been found.

- The project name must be unique. This is because we use the project name to identify separate projects. If the same name for two different projects is used, an exception will be thrown and the platform will exit.

– The project names, test names, VCS URL and project URL may not contain spaces. Instead underscore is recommended to use. However, if the repository URL contains white spaces, %20 must be used.

– Only complete folders and subfolders can be checked out. There is no way to specify just a limited number of files. It is nonetheless possible to do *tests* on a subset of all files. This is accomplished by having this control in the test executor plugin.

– There is now way to distinguish between errors in two different runs. For instance, if a file has five errors in two subsequent runs, it is not possible to conclude if it is the same five errors or if some old have been fixed and new ones have been found.

– Only a single source file can be associated with a notification object. Often this is not a problem since a notification usually originates from just one file, but it is possible to think of cases where multiple files are affected. No support for this exists though.

– Only information related to erroneous files are inserted to the database. This means that if a file was successfully tested in a certain run, no information (regarding that run) can be found in the database about that file.

## 5.3 Future Work

The work with the platform and the webpage has been conducted with extendibility in mind. In this way, it should be easy to hook on new modules or adjust the current code. During the work there have been many ideas of how to further develop and extend the platform. A presentation of the most promising now follows:

– *Extension of the CUnit test.* The current CUnit plugin has some limitations; it is for instance tedious to add a new unit test and it is not adapted for working with the whole engine control system. Thus, this plugin should be further developed as it can be a great way of testing. One extension could be to make it possible to upload unit tests via the web site. This could be done by choosing which source file that contains the function to be tested and insert the actual test code in a text area. This would greatly increase the simplicity with the CUnit plugin.

– *Adding new tests.* As discussed in section 3.2, there exist several dynamic test techniques that could be realized as automatic test plugins in the platform. Although there are many problems and questions on how this can be done, it would be desirable to at least have one of these techniques. This is because many of the current tests are either static/syntax checking or at the unit level. No test at the integration or system level currently exists.

  There may also exist other testing techniques at Scania or elsewhere that could be implemented as plugins. The possibility to add these could also be further investigated.

– *GUI to the platform.* The current platform has no GUI. The only interface to it is via the configuration file and some constants. The output from it, except the test result, is only the log files that are written during one run. One future project could be to create a GUI to the platform, where an administrator could easily change settings and check the status of runs. Such a GUI could also support an easy way to add or delete tests.

– *Extend the web site.* The web site could be extended to support more options and possibilities. This could for instance be functions for searching and sorting error messages and source files. It could also be better ways to compare different runs, projects and files or have certain files under observation.

  To make the web site as useful as possible, a comprehensive study of what functionality to support should be made. The result of this could also lead to changes in the platform, or changes in the database design.

# References

[1] Software life cycle. http://searchcio.techtarget.com/searchCIO/downloads/LifeCycleModelling.pdf , accessed October 14, 2005.

[2] Gauging software readiness with defect tracking. http://www.stevemcconnell.com/ieeesoftware/bp09.htm, accessed October 24, 2005, 1997.

[3] Jtest. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, accessed October 21, 2005, 2005.

[4] Luntbuild. http://www.pmease.com/luntbuild/, accessed October 20, 2005, 2005.

[5] Misra c. http://www.misra-c2.com/, accessed November 9, 2005, 2005.

[6] IEEE Std. 829-1998. Ieee standard for software test documentation. Technical report, The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017-2394, USA, 1998.

[7] M. H. Poonawala A. K. Onoma, W. Tsai and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41:81–86, 1998.

[8] Kent Beck. Simple smalltalk testing: With patterns. http://www.xprogramming.com/testfram.htm, accessed October 11, 2005, 1994.

[9] P. Bengtsson. Structuring of models intended for complete vehicle simulation. Technical report, Dept. of Information Sc., University of Uppsala, Uppsala, Sweden, 2004.

[10] P. E. Black. Automatic software test generation from formal specifications. http://www.itl.nist.gov/div897/docs/automatic_software_test_generation.html, accessed October 12, 2005, 2003.

[11] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, pages 61–72, 1988.

[12] J. Bowen and V. Stavridou. Safe–critical systems, formal methods and standards. *Software Engineering Journal*, 1993.

[13] M. J. Garfield C. F. Cohen, S. J. Birkin and H. W. Webb. Managing conflict in software testing. *Communications of the ACM*, pages 76–81, 2004.

[14] M. Collins. Formal methods. Technical report, Carnegie Mellon University 18-849b Dependable Embedded Systems, 1998. http://www.ece.cmu.edu/~koopman/des_s99/formal_methods/, accessed October 13, 2005.

[15] Automated QA Corp. Automated build studio. http://www.automatedqa.com/products/abs/index.asp, accessed October 20, 2005, 2005.

[16] M. A. Cusumano and R. W. Selby. How microsoft builds software. *Communications of the ACM*, 40:53–61, 1997.

[17] L-G. Andersson E-A. Karlsson and P. Leion. Daily build and feature development in large distributed projects. *Proceedings of the 22nd international conference on Software engineering*, 2000.

[18] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns.* Addison-Wesley Professional Computing. Addison Wesley, 1995.

[19] O. J. Dagl E. W. Dijkstra and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[20] N. S Eickelman and D. J. Richardson. What makes one software architecture more testable than another. *Joint proceedings of the second international software architecture workshop (ISAW–2) and international workshop on multiple perspectives in software development (Viewpoints 96) on SIGSOFT 96 workshops*, 1996.

[21] R. Elmasri and S. B. Navathe. *Fundamentals of database systems, [ch 10].* Addison Wesley, 2004.

[22] K. Sierra Eric Freeman, Elisabeth Freeman and B. Bates. *Head First Design Patterns.* O'Reilly Media, 2004.

[23] C. Erickson and P. Jorgensen. Object-oriented integration testing. *Communications of the ACM*, pages 30–38, 1994.

[24] M. Fowler. Continuous integration.
http://martinfowler.com/articles/continuousIntegration.html accessed 21 October, 2005.

[25] A. German. Software static code analysis lessons learnt. Technical report, Intellectual Property Department, QinetiQ ltd., Technology Park, Farnborough, Hampshire GU14 0LX, 2003.

[26] M. S. Johnson. A survey of testing techniques for object-oriented systems. *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, 1996.

[27] C. Jones. Software Engineering Series. Mcgraw-Hill, 1991.

[28] B. Korel. Testgen – a structural test data generation system. *Proceedings of the 6th International Conference on Software Testing*, 1995.

[29] B. Korel and A. M. Al-Yami. Automated regression test generation. *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 143–152, 1998.

[30] Y. Li and N. J. Wahl. An overview of regression testing. *ACM Sigsoft. Software Engineering Notes*, 24:69–73, 1999.

[31] E. Macdonald. Bugs and breaches. *International Journal of Law and Information Technology*, 13:118–138, 2005.

[32] T. McGibbon. An analysis of two formal methods: Vdm and z. Technical report, Rome Laboratory RL/C3C, 1997.

[33] G. McGraw and C. Michael. Automatic generation of test cases for software testing. http://www.cigital.com/papers/download/tcg-cogsci.pdf, accessed October 20, 2005.

[34] S. Norden. Developing a test method for use in a small software organization - a case study. Technical report, Institute of technology, Lund, 2002.

[35] M. Olan. Unit testing: test early, test often. Technical report, Computer Science and Information Systems, Richard Stockton College, Pomona, NJ 08240, 1998.

[36] B. Appleton R. Cabrera and S. P. Berczuk. Software reconstruction: Patterns for reproducing software builds. *Proceedings of the 1999 Pattern Languages of Programming Conference*, 1999.

[37] G. Rothermel and M. J Harrold. Analysis regression test selection techniques. *IEEE Transactions on Software Engineering*, 22 aug 1996:526–551, 1996.

[38] R. Weber S. Berner and R. K Keller. Observations and lessons learned from automated testing. *Proceedings of the 27th international conference on Software engineering*, 2005.

[39] T. Sandberg. Heavy truck modeling for fuel consumption simulations and measurements. Technical report, Division of Vehicular Systems, Department of Electrical Engineering,Linköping University,, 2001.

[40] E. Seipmann and A. Newton. Tobac: A test case browser for testing object-oriented software. *ACM Software Engineering Notes*, pages 154–168, 1994.

[41] Gimpel Software. Pc-lint and flexelint. http://www.gimpel.com/html/products.htm, accessed November 9, 2005, 2005.

[42] J. M. Spivey. *The Z Notation: A Reference Manual [ch 1.2]*. Prentice Hall International (UK) Ltd, 1992.

[43] PhaedruS SystemS. Qa c. http://www.phaedsys.org/prqa.html, accessed November 9, 2005, 2005.

[44] J. Kim A. Porter T. L. Graves, M. J. Harrold and G. Rothermel. An empirical study of regression test selection techniques. Technical report, National Institute of Statistical Sciences, 19 T. W. Alexander Drive, PO Box 14006, Research Triangle Park, NC 27709-4006, 1997.

[45] E. Tran. Verification/validation/certification. Technical report, Carnegie Mellon University 18-849b Dependable Embedded Systems, 1999. http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/index.html, accessed October 13, 2005.

[46] F. I. Vokolos and P. G. Frankl. Pythia: A regression test selection tool based on textual differencing, 1997.

[47] J. A. Whittaker. Vad är test av mjukvara? och varför är det så svårt? *OnTime*, March 2003:6–11, 2000.

# Appendix A

# Abbreviations

**BCNF** Boyce-Codd Normal Form

**CSS** Cascading Style Sheets

**DOM** Document Object Model

**FREC** Flight Recorder

**GUI** Graphical User Interface

**HTML** HyperText Markup Language

**IEEE** Institute of Electrical and Electronics Engineers

**KLOC** Thousands of line of code

**MISRA** The Motor Industry Software Reliability Association

**PHP** PHP: Hypertext Preprocessor

**SHTL** Scania Heavy Truck Library

**SQL** Structured Query Language

**TDD** Test-Driven Development

**VCS** Version Control System

**XML** eXtensible Markup Language

**XP** Extreme Programming

# Appendix B

# UML Diagrams

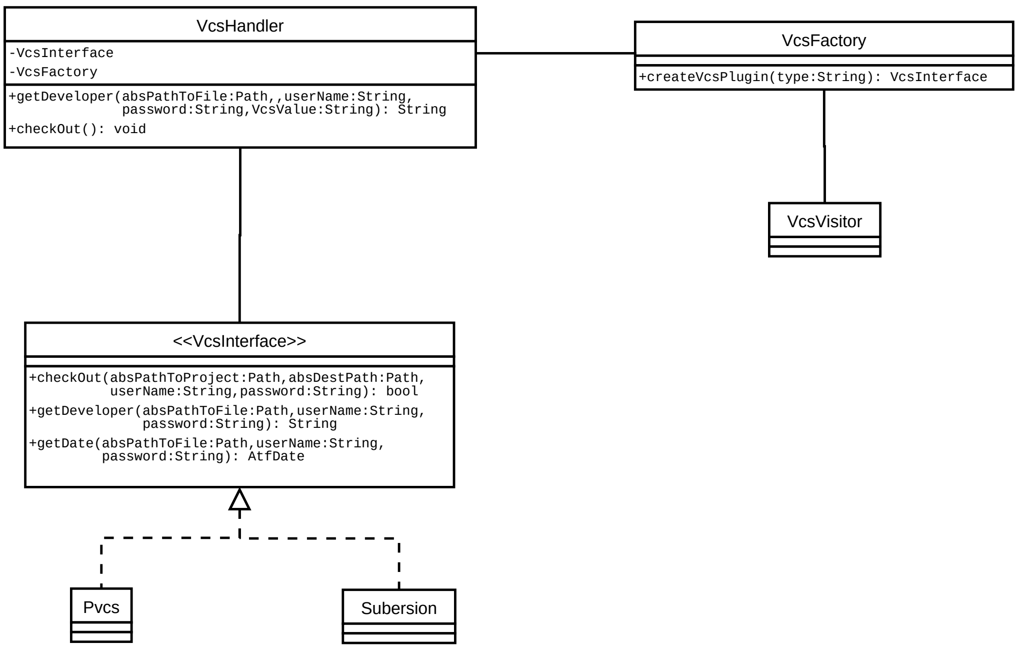

VcsHandler
-VcsInterface
-VcsFactory
+getDeveloper(absPathToFile:Path,,userName:String, password:String,VcsValue:String): String
+checkOut(): void
VcsFactory
+createVcsPlugin(type:String): VcsInterface
VcsVisitor
<<VcsInterface>>
+checkOut(absPathToProject:Path,absDestPath:Path, userName:String,password:String): bool
+getDeveloper(absPathToFile:Path,userName:String, password:String): String
+getDate(absPathToFile:Path,userName:String, password:String): AtfDate
Pvcs
Subersion

Figure B.1: The VCS module

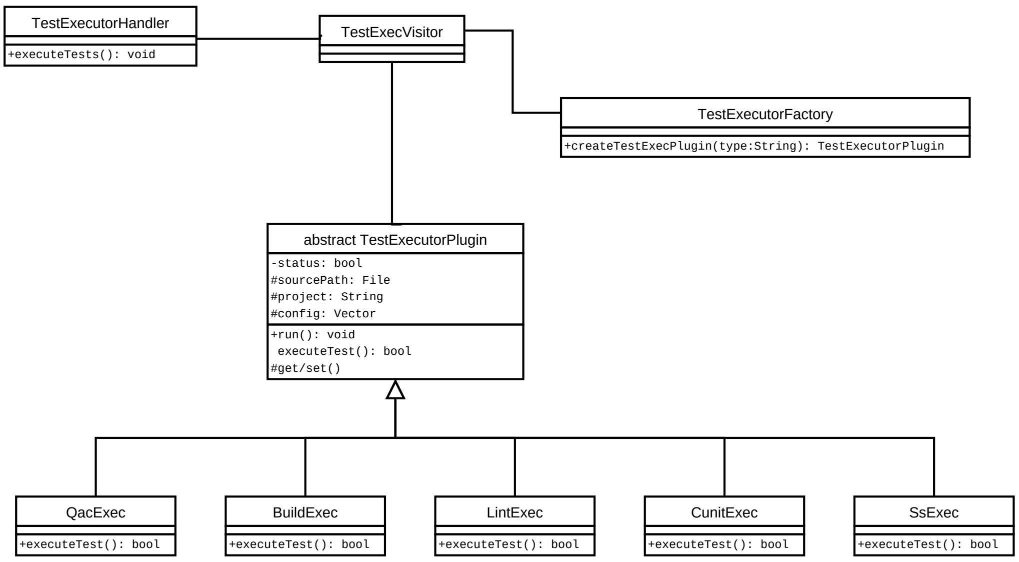


Figure B.2: The TestExec module

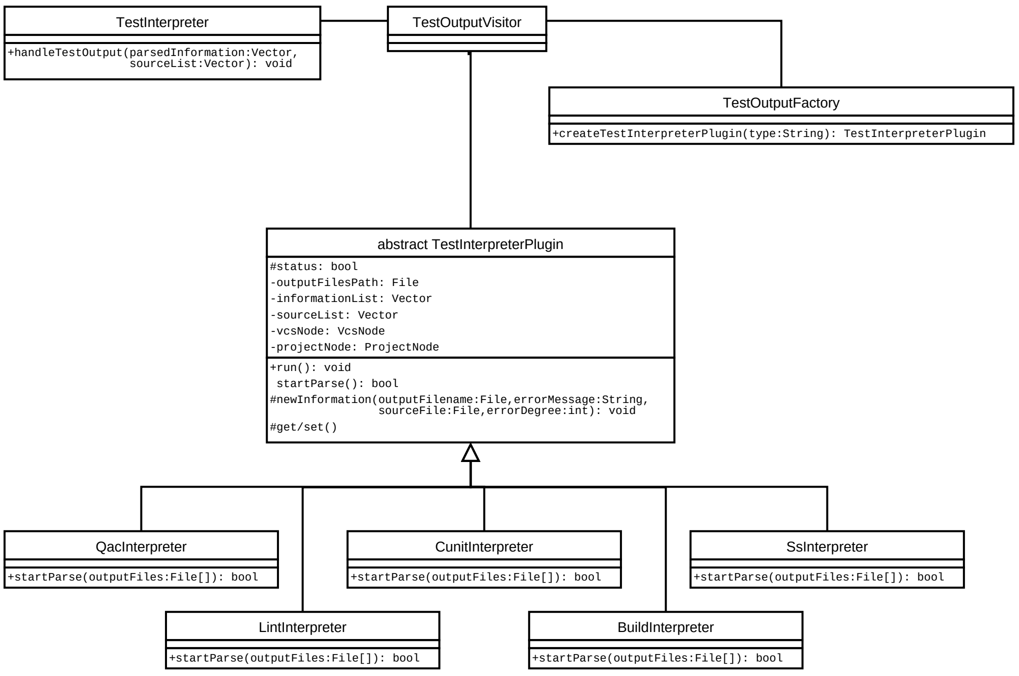

TestInterpreter
+handleTestOutput(parsedInformation:Vector, sourceList:Vector): void
TestOutputVisitor
TestOutputFactory
+createTestInterpreterPlugin(type:String): TestInterpreterPlugin
abstract TestInterpreterPlugin
#status: bool
-outputFilesPath: File
-informationList: Vector
-sourceList: Vector
-vcsNode: VcsNode
-projectNode: ProjectNode
+run(): void
startParse(): bool
#newInformation(outputFilename:File,errorMessage:String, sourceFile:File,errorDegree:int): void
#get/set()
QacInterpreter
+startParse(outputFiles:File[]): bool
CunitInterpreter
+startParse(outputFiles:File[]): bool
SsInterpreter
+startParse(outputFiles:File[]): bool
LintInterpreter
+startParse(outputFiles:File[]): bool
BuildInterpreter
+startParse(outputFiles:File[]): bool

Figure B.3: The TestInterpreter module

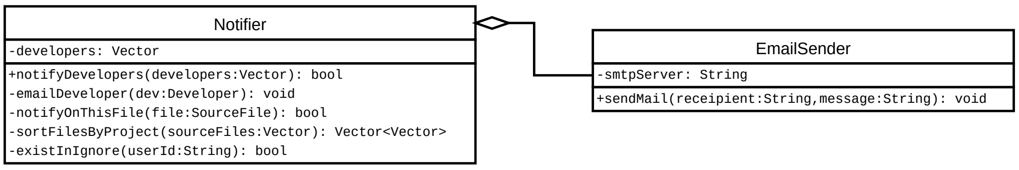

Notifier
-developers: Vector
+notifyDevelopers(developers:Vector): bool
-emailDeveloper(dev:Developer): void
-notifyOnThisFile(file:SourceFile): bool
-sortFilesByProject(sourceFiles:Vector): Vector<Vector>
-existInIgnore(userId:String): bool
EmailSender
-smtpServer: String
+sendMail(receipient:String,message:String): void

Figure B.4: The Notifier module

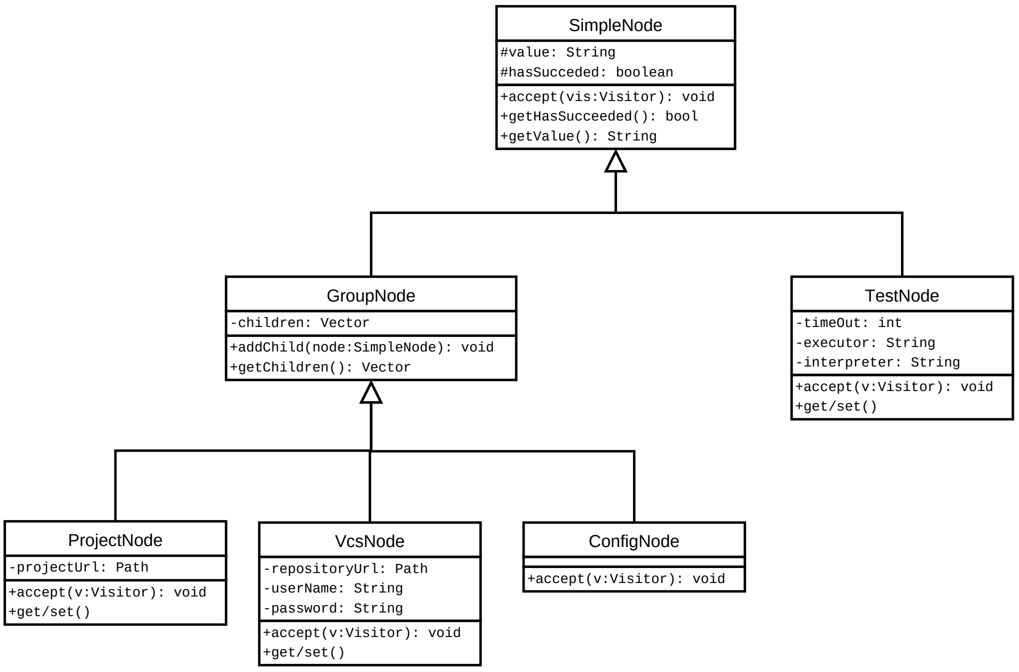


Figure B.5: The Node hierarchy

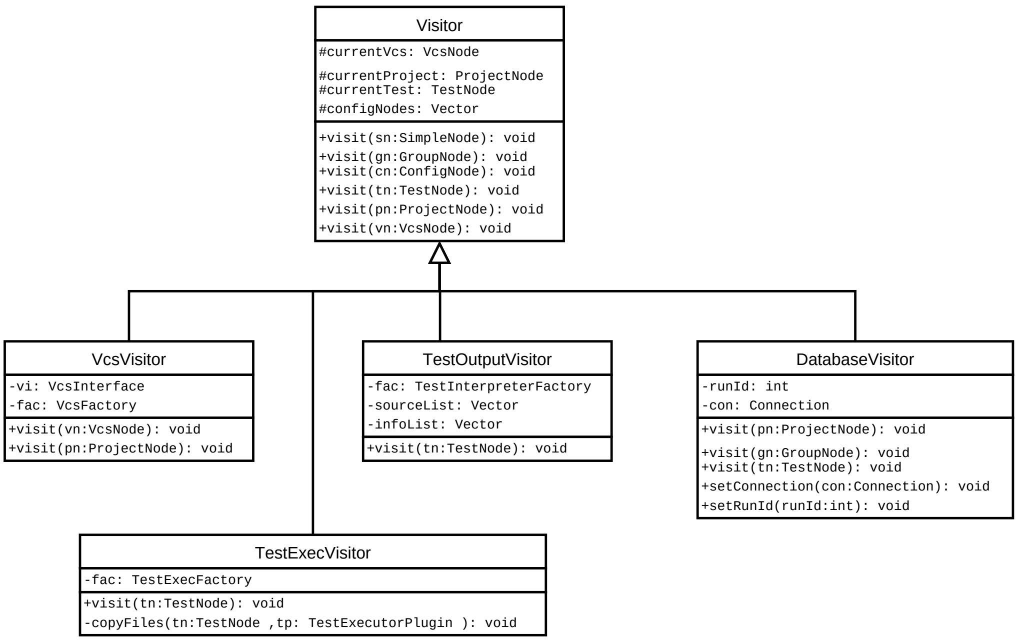

Visitor
#currentVcs: VcsNode
#currentProject: ProjectNode
#currentTest: TestNode
#configNodes: Vector
+visit(sn:SimpleNode): void
+visit(gn:GroupNode): void
+visit(cn:ConfigNode): void
+visit(tn:TestNode): void
+visit(pn:ProjectNode): void
+visit(vn:VcsNode): void
VcsVisitor
-vi: VcsInterface
-fac: VcsFactory
+visit(vn:VcsNode): void
+visit(pn:ProjectNode): void
TestOutputVisitor
-fac: TestInterpreterFactory
-sourceList: Vector
-infoList: Vector
+visit(tn:TestNode): void
DatabaseVisitor
-runId: int
-con: Connection
+visit(pn:ProjectNode): void
+visit(gn:GroupNode): void
+visit(tn:TestNode): void
+setConnection(con:Connection): void
+setRunId(runId:int): void
TestExecVisitor
-fac: TestExecFactory
+visit(tn:TestNode): void
-copyFiles(tn:TestNode ,tp: TestExecutorPlugin ): void

Figure B.6: The Visitor hierarchy

# Appendix C

# User's Guide

## Add a new test

When a new test is to be added, there are a few steps that have to be taken. In the list below these steps have been summarized. The list can be seen as a guide for how to proceed when adding a new test.

1. Write the actual test executor plugin. This class must extend the abstract class: `TestExecutorPlugin`. Two methods must be implemented:

   ```
   public boolean executeTest()
   public File[] getOuputFilePath()
   ```

   The first method is called from the platform when the test should be executed. The second gets called after the test is finished and its purpose is to report the files (generated by the test) that are of interest.

   The `executeTest` method return a boolean indicating the success or failure of the test. If the boolean is set to false, no interpretation of the test result will be done. The code inside this method should contain the actual execution of the test. The test itself could be written completely in java, or it could be an external program that gets started from here.

   The super class `TestExecutorPlugin` has some useful methods that are available for the plugin. For instance, returning the location of the source code and which project it belongs to.

2. Modify the `TestExecFactory` so that it recognizes the new test and can create the newly made plugin. For instance if a plugin named `CodeVerificatonExec` (related to the executor named CodeVerification) has been created, the following lines should be added to the `createTestExecPlugin` method:

   ```
   else if(type.equalsIgnoreCase("CodeVerification"))
    testExecPlugin = new CodeVerificationExec();
   ```

3. When the first two steps are done, the implementation of the test execution is completed. What remain is to create the interpreter. For this, a test interpreter plugin must be created. This plugin should extend the `TestPluginInterpreter` class. Only one method must be implemented:

`boolean StartParse(File[] absOutputFiles)`

This is the method that gets called from the platform when the actual interpreting of the test should start. The parameter `absOutputFiles` contains a list of all files generated by the test (the files reported by the executor plugin). The returned boolean indicates if the interpreting was successful or not.

All that remains now is to do the actual interpreting of the files. When an error or warning has been found, the method:

```
newNotification(File outputFile, String errorMessage,
File sourceFile, int errorDegree)
```

should be called. The first parameter is the path to the file interpreted and the second is the actual error message.

`SourceFileName` is the path to the file the error was discovered in. This could be an absolute path, or just the file name.

The last parameter is the error degree. There are four levels of error degrees to use: `ERROR_DEGREE_UNKNOWN`, `ERROR_DEGREE_MIN`, `ERROR_DEGREE_MEDIUM`, `ERROR_DEGREE_HIGH`.

By calling this method a new notification will be created which later can be viewed on the web site.

4. Modify the `TestInterpreterFactory` so that it recognizes the new test and can create the newly made plugin. For instance if a plugin named `CodeVerificatonInterpreter` (related to the interpreter named CodeVerification) has been created, the following lines should be added to the `createTestInterpreterPlugin` method:

```
else if(type.equalsIgnoreCase("CodeVerification"))
 testInterpreterPlugin = new CodeVerificatonInterpreter();
```

5. The last step is to add the test to the config file. The syntax of this file is described in the next section

## C.1 Platform settings

All specific paths and constants have been gathered in a static class called `PlatformSettings`. It is recommended to use this whenever a new constant or a new path to a program is to be added. By having all such settings collected in a single class it makes it easier to move our program from one computer to antoher.

## Configuration file

This is a description of the configuration file. The structure is shown below and a description of each parameter then follows.

```
<CONFIG>
   <TESTSTRUCTURE>
      <VCS Value="[VCS name]" RepURL="[Path to the VCS]"
         User="[Login name]" Pass="[Login password]">
         <PROJECT Value="[Project name]"
          ProjURL="[Path on the VCS to THIS project]">

            <TEST Name="[Test name]" TimeOut="[Timeout time]"
             Executor=[Executor name]
             Interpreter="[Interpreter name]"
          >
            </TEST>

            <TEST ...>
            [May contain more tests]
            </TEST>

         </PROJECT>

         <PROJECT ...>
         [May contain more projects, with tests]
         </PROJECT>

       </VCS>

       <VCS ...>
       [May contain more VCS systems, with projects]
       </VCS>
   </TESTSTRUCTURE>

   <GENERAL>
      <BASEPATH>[Path for SAFE output]</BASEPATH>
      <IGNORELIST>
         <IGNORE>[User name to ignore]</IGNORE>
         <IGNORE ...>
         [May contain more]
         </IGNORE>
      </IGNORELIST>
   </GENERAL>
</CONFIG>
```

– `[VCS name]`

  This is the name of the VCS. Must be the same as the corresponding name in the VCSFactory.
  Restrictions: Do NOT use white space in the name.
  Examples: “subversion”, “PVCS”.

– `[Path to the VCS]`

  The path to the VCS.
  Restrictions: Do NOT use white space in the path.
  Examples: “pvcs://computer1/”, “svn://mapc09/DistDIMA”.

– `[Login name]`

The VCS login name.
Restrictions: None known.
Examples: "sss745".

- `[Login password]`

  The VCS password for this login name.
  Restrictions: None known.
  Examples: "scania".

- `[Project name]`

  This can be any name that describes a project. It does not have to match anything else, however changing the project name will be interpreted as a completely new project.
  Restrictions: Do NOT use white space in the name.
  Examples: "S6", "MyTruck".

- `[Path on the VCS to THIS project]`

  The path on the VCS where this projects files are. This does NOT include the path TO the VCS.
  Restrictions: Do NOT use white space in the path.
  Examples: "trunk/work".

- `[Test name]`

  This can be any name that describes a test. It does not have to match anything else, however changing the test name will be interpreted as a completely new test.
  Restrictions: Do NOT use white space in the name.
  Examples: "QAC", "Lint".

- `[Timeout time]`

  The timeout time for both the executor and the interpreter, respectively, in seconds.
  Restrictions: Only use intergers.
  Examples: "3600".

- `[Executor name]`

  This is the name of the executor for this test. Must be the same as the corresponding name in the TestExecutorFactory.
  Restrictions: Do NOT use white space in the name.
  Examples: "QacExecutor".

- `[Interpreter name]`

  This is the name of the interpreter for this test. Must be the same as the corresponding name in the TestInterpreterFactory.
  Restrictions: Do NOT use white space in the name.
  Examples: "QacInterpreter".

- `[Path for SAFE output]`

  This is an absolute path where all output from the SAFE system (test output, log files etc.) get. Subfolders are however created for each new run.
  Restrictions: Do NOT use white space in the path.
  Examples: "C:\SAFE\Output\".

– `[User name to ignore]`

  This is a user name that will not receive automatic e-mail from the SAFE system. Must be the same as stored on the VCS used, for that person.
  Restrictions: None known.
  Examples: “sss745”