Developing a Maintainable Test Case Generator for Automatic Testing of Computer-Based Interlocking Systems

MIKAEL KRYDZINSKI

KTH Electrical Engineering

Degree project in
ICS
Master thesis
Stockholm, Sweden 2015

XR-EE-ICS 2015:005

Abstract

Developing software without considering the potential changes it might have to undergo in the future can be a costly mistake. This is because its maintenance costs can become very expensive as they can consume over 90% of the total life-cycle costs. Incorporating maintainability in software has for this reason become highly attractive since it can significantly reduce the maintenance costs and therefore save companies and software developers a fortune. This thesis presents a software tool that has been developed to aid Bombardier in the verification of computer-based interlocking (CBI) systems. The tool automatically generates test cases which represent the different tests that verify the interlocking system. The paper is divided into two parts. The first part focuses on the maintainability of the tool while the second part investigates whether the tool can speed up the testing process of CBI-systems at Bombardier. The results show that the tool is highly maintainable and that tests on CBI-systems can be performed significantly faster with it.

Keywords: Computer-based interlocking system, Software, Maintainability, Test cases, Test case generator

Sammanfattning

Att utveckla mjukvara utan hänsyn till de eventuella förändringar som den måste genomgå i framtiden kan bli ett kostsamt misstag. Detta beror på att dess underhållskostnader kan bli mycket dyra eftersom de kan konsumera över 90% av de totala kostnaderna. Det har därför blivit väldigt attraktivt att lägga fokus på underhållbarhet när man utvecklar mjukvara eftersom det kan avsevärt minska underhållskostnaderna och därmed spara mjukvaruutvecklare och företag en förmögenhet. I denna uppsats presenteras ett verktyg som har utvecklats för att hjälpa Bombardier med verifiering av datorställverk. Verktyget automatiskt genererar testfall som representerar de olika testen som verifierar ställverket. Uppsatsen är uppdelad i två delar. Den första delen fokuserar på verktygets underhållbarhet medan den andra delen undersöker om verktyget kan påskynda testprocessen för datorställverk på Bombardier. Resultaten visar att verktyget är väldigt underhållbart och att test på datorställverk kan utföras mycket snabbare med hjälp av den.

Nyckelord: Datorställverk, Mjukvara, Underhållbarhet, Testfall, Testfallsgenerator

Acknowledgments

Firstly, I want to express my utmost appreciation to my examiner Robert Lagerström at the Royal Institute of Technology for all the support, feedback and guidance that he has given me throughout the whole thesis. Olof Sundqvist at Bombardier has been an outstanding supervisor for the project, and I would like to thank him for the valuable time and help that he has given me. I would also like to thank Henrik Yngvesson for providing me with everything needed for the project and for giving me the opportunity to do my master thesis at Bombardier. I also had the opportunity to collaborate with Mohamed Bakhiet during the thesis. The many interesting discussions and conversations we had were very enjoyable and will not be forgotten. Lastly, I would like to thank my family and friends for their endless support.

List of abbreviations

CBI Computer-based interlocking

CC Cyclomatic complexity

GA General application

GP Generic product

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

MI Maintainability index

Contents

Chapter 1

Introduction

Today trains are transporting thousands of passengers all over the world at high speeds through a complex network of railways. To ensure the safety of the passengers, stations are equipped with interlocking systems. These are safety-critical systems that control the railway traffic and prevent trains from colliding and derailing [1, 2]. There is variety of different interlocking systems such as relay and computer-based interlocking systems (CBI-systems). Although there are some differences between them, they all operate on the same principle that certain conditions must be fulfilled before a train is allowed to enter a route [3].

There have been a number of train accidents in the recent past. Table 1.1 shows the train accidents caused by collisions and derailments in Sweden between 2008-2012 [4]. With new and better methods of verification the interlocking system becomes less error-prone, thus reducing the probability of train accidents. This makes the railway a safer and more reliable mode of transportation.

Accident	**2008**	**2009**	**2010**	**2011**	**2012**
Derailments of trains in motion	14	7	8	7	10
Collisions of trains in motion	4	1	3	2	4
Collisions at level crossings	6	16	16	9	12
Derailments and collisions when shunting	6	4	5	6	4
Total	**30**	**28**	**32**	**24**	**30**

Table 1.1: Train collisions and derailments in Sweden between 2008-2012 [4]

1.1 Background

Bombardier has over the years been verifying CBI-systems manually by running a range of different tests on them in a simulated train yard. They have now begun using an automatic method of testing by expressing the tests in the form of test cases. A test case is a set of instructions and conditions which verify that a certain interlocking function works as expected. The way a CBI-system is automatically tested is by writing down a set of test cases and sending them into a test engine. The test engine then executes all the test cases and delivers a report of what tests the CBI-system passed or failed. The biggest advantage with this method of testing over the manual one is that CBI-systems can easily be retested if any changes are made in the interlocking logic by simply resending the test cases into the test engine again. This saves Bombardier resources as well as time for more important tests.

1.2 Thesis description

Bombardier are now looking to develop a software tool that automatically generates the corresponding test cases to an interlocking test as it may be a faster alternative than that of manually writing them down. In doing so, it is important to ensure that the tool is able to cope and adapt to changes in a cost efficient manner. This is because a software product can be very expensive to modify and keep updated after its development. For example, correcting an error during the operations/maintenance phase can be more than 50 times as expensive to fix than during the development phase, see figure 1.1 [5]. For this reason, the tool also has to be developed in a way which keeps its maintenance costs at a minimum.

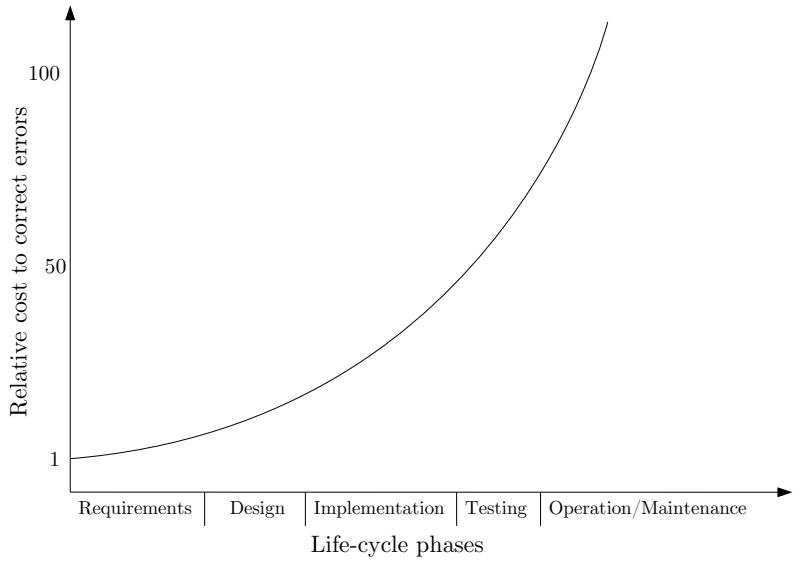


Figure 1.1: Relative cost to correct errors [5]

1.3 Thesis purpose

The purpose of the thesis was to reduce the testing time of CBI-systems at Bombardier by developing a maintainable software tool that automatically generates test cases.

1.4 Objectives

The objectives of the thesis were to:

- Develop a software tool that automatically generates test cases for CBI-systems.
- Ensure that the tool is easy to maintain.
- Compare the time it takes to complete tests with and without the tool.
- Write a user-manual.

1.5 Method

The thesis was a 20 week project performed at Bombardier. The work carried out to fulfill all the thesis objectives was as follows:

- **Pre-study**
  The thesis began with a pre-study focusing on gaining a deeper understanding of the tool that was to be developed. This was done by studying its requirements and functionality as well as all the different data formats that were to be used in it.
- **Literature study**
  A literature study was carried out in order to find which factors affect a software's maintainability. Different naming conventions and various programming methods were also looked at.
- **Tool development**
  The knowledge acquired from the pre-study and literature study was put into practical use to develop a maintainable tool that generates test cases. Once the tool was complete, a user manual was written to accompany it. The development of the tool was the most time consuming part of the thesis.
- **Evaluating the tool**
  The performance of the tool was evaluated by comparing the time it takes to complete tests with and without the tool on a project at Bombardier.

1.6 Delimitations

Software maintainability is affected by numerous factors [6]. Due to time constraints of the thesis, only five factors were taken into consideration when developing the tool i.e. modularity, cohesion, coupling, readability and complexity. The reason why these factors were chosen was because they were found to be most relevant for this project and therefore would most likely have the greatest impact on the maintainability of the tool.

1.7 Thesis outline

The rest of the thesis is organized as follows:

Chapter 2 introduces software maintenance and maintainability as well as their relation to one another. The maintainability factors that were considered in this thesis are also described in this part.

Chapter 3 presents the tool that was developed and the results that were obtained from the complexity and maintainability measurements.

Chapter 4 compares how much faster tests can be performed with the tool than without it.

Chapter 5 discusses the maintainability of the tool and interprets the results from chapter 4.

Chapter 6 concludes the thesis and gives some final thoughts for future work.

Chapter 2

Software maintenance & maintainability

2.1 Software maintenance

A software product is not complete after its delivery. It usually has to be modified numerous times after its initial development [7], partially to correct various bugs and errors that are discovered, but also to make the necessary enhancements and adaptations due to changes in its environment. This process of modification is known as software maintenance. A generally accepted definition of software maintenance is [8, 9]:

> *Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt a product to a modified environment. ([8], p.4)*

There are several reasons as to why software has to undergo maintenance. A few examples are listed below:

- Anti-virus software continuously has to be updated and improved to be able to handle new computer threats.
- Changes in tax laws will necessitate that the software used at tax offices is modified [5].
- If the operating system is upgraded, certain modifications in the software might be necessary to accommodate the new operating system [9].
- Improving a software's functionality in order to meet user requests [5].

2.2 The costs of maintenance

Software cannot always be maintained as desired due to budget constraints. In fact, various studies over the past decades show that maintenance costs are increasing and can consume over 90% of the total software life-cycle costs, see table 2.1 [10]. Therefore, it has become increasingly important to develop software that is easy to maintain as it may save a fortune in the long run. Increasing a software's maintainability has arguably become the best way to lower the maintenance costs.

Year	**Proportion of software maintenance costs**	**Definition**	**Reference**
2000	$> 90\%$	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	$> 90\%$	Software cost devoted to system maintenance & evolution / total software costs	Moad (1990)
1990	$60 - 70\%$	Software maintenance / total management information systems (MIS) operating budgets	Huff (1990)
1988	$60 - 70\%$	Software maintenance / total management information systems (MIS) operating budgets	Port (1988)
1984	$65 - 75\%$	Effort spent on software maintenance / total available software engineering effort.	McKee (1984)
1981	$> 50\%$	Staff time spent on maintenance / total time (in 487 organizations)	Lientz & Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz et al. (1979)

Table 2.1: Proportional software maintenance costs for its supplier [10].

2.3 What is software maintainability?

Over the years there have been a number of definitions given to software maintainability. Three well known definitions of software maintainability are shown below.

Federal Information Processing Standards defines software maintainability as [11]:

> *Maintainability is the ease with which software can be changed to satisfy user requirements or can be corrected when deficiencies are detected. ([11], p.9)*

ISO/IEC international standard has the following definition of software maintainability [12]:

> *The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. ([12], p.10)*

The definition of software maintainability described in IEEE Standard Glossary of Software Terminology is [13]:

> *The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. ([13], p.46)*

Although these definitions are differently phrased, the general consensus is that software maintainability is an attribute that reflects how easy it is for software to undergo maintenance. Therefore, the higher the maintainability is, the easier maintenance becomes.

2.4 Factors affecting maintainability

Various authors have identified a range of different factors that affect a software's maintainability. This study takes a closer look at five of them, see figure 2.1. These factors are frequently found in the literature as well as in different papers in the subject matter [5, 6, 9, 15, 18–20].

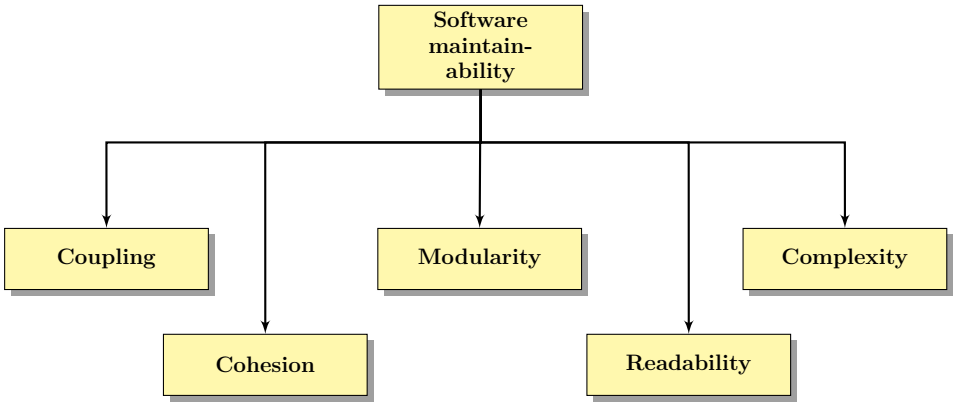


Figure 2.1: Factors affecting maintainability

2.4.1 Modularity

A design technique used in software engineering is to build a program from modules. In this context, a module is a part of a program that performs a specific function [14]. An example of a modular system is the computer. It consists of a processor, hard drive, graphics card, mother board etc. All these components are modules that make up all the functionality in the computer. Developing software in this manner brings about a number of benefits [15]:

- Modular systems are easier to understand as they can be understood piece-wise.
- Modular systems are easier to describe and therefore easier to document.
- Modular systems are easier to test as modules can be tested individually as well as together with other modules in the program.
- Modules can be developed in parallel with each other.
- Modules can be changed without affecting the rest of the program.

The above points are however dependent on internal and external qualities of the modules such as coupling and cohesion. These two concepts are explained in more detail below.

2.4.2 Coupling

Coupling is the degree of interdependence between modules [13]. An illustration of loosely and highly coupled modules can be seen in figures 2.2 and 2.3 [16]. With highly coupled modules, a change in one module is likely to affect other modules. On the other hand, loosely coupled modules are less likely to affect each other [5]. This makes loosely coupled modules easier to modify.

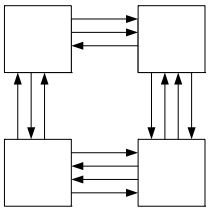


Figure 2.2: Highly coupled modules [16]

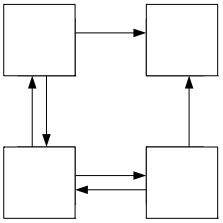


Figure 2.3: Loosely coupled modules[16]

2.4.3 Cohesion

Cohesion is a measure of the functional strength in a module. A module with high cohesion performs only a specific task and requires a minimal amount of help from other modules [16]. Benefits of a cohesive module include a simpler correction of errors and other modifications. Think of cohesive modules as independent systems in a car. Each system is responsible to notify the driver if there is something wrong with it. For example, a fault in the charging system is indicated by the battery light on the dash board. This way the whole car does not have to be inspected as the source of error is known.

2.4.4 Readability

Readability is the degree of ease with which the source code can be read and understood [15]. This is an important factor as programmers are spending 40-60% of their time on reading and understanding the source code [9]. Increasing the readability can therefore greatly reduce the maintenance costs. Two ways to make a program more readable is by documenting it and using an appropriate naming convention.

2.4.5 Complexity

Complexity is a measure of how difficult a program is to understand and verify [13]. It is affected by numerous factors such as the degree of coupling and the structure of the code. If the complexity is high, a program is harder to understand and therefore also harder to maintain. In this thesis the complexity will be quantified by measuring the cyclomatic complexity (CC), which is one of the most popular code

complexity metrics. CC is a measure of the linearly independent paths through a program and its value can be used to:

- Help identify complex programs.
- Predict the amount of time it will take to understand and modify a program.

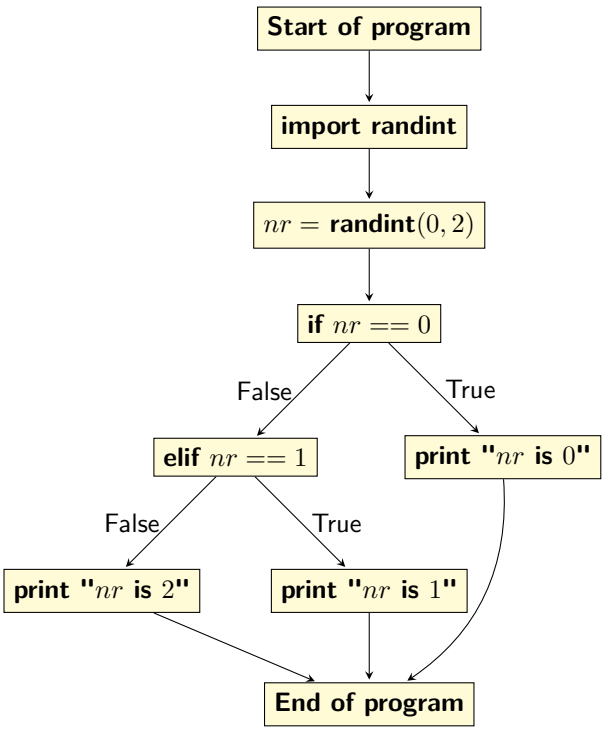
CC can be calculated by representing the code as a graph with the code statements as nodes and the flow of control between the statements as the edges with the following formula:

$$CC = e - n + 2$$

where e is the total number of edges and n is the total number of nodes. Alternatively, it can be calculated by counting the number of decision points in the code with [17]:

$$CC = \psi + 1 \tag{2.1}$$

where ψ is the total number of decision points. Just to show a concrete example of how CC is calculated, a short program with its corresponding graph is illustrated below. The graph has 9 nodes and 10 edges which gives us $CC = 10 - 9 + 2 = 3$. Using the alternative method, we see that there is one if-statement and one elif-statement which gives us $\psi = 2$ and $CC = 2 + 1 = 3$. Therefore, there are 3 independent paths from start to end in the program which is obviously the case. The information regarding CC is retrieved from [5].



Random number generator

```
#Generate number between 0 and 2
from random import randint
nr = randint(0,2)
if nr == 0:
    print "nr is 0"
elif nr == 1:
    print "nr is 1"
else:
    print "nr is 2"
```

Chapter 3

The tool

3.1 Overview of the tool

The tool that was developed consists of two parts. The first part is the *test case generator* which extracts, stores and writes information from different input data files. The second part is the *custom script* where the user specifies how the extracted information should be used to construct test cases for a test. When the *custom script* is executed, information from different data files is extracted and stored in *test case generator*. The stored information can then be accessed through the *custom script*, where it is structured according to the 'test case format'. Once all the information has been structured, it is sent back into a writer in the *test case generator* and a file containing test cases is generated. The CBI-system is then be tested by simply sending the generated file into the test engine where all the test cases are executed. An overview of the tool can be seen in figure 3.1.

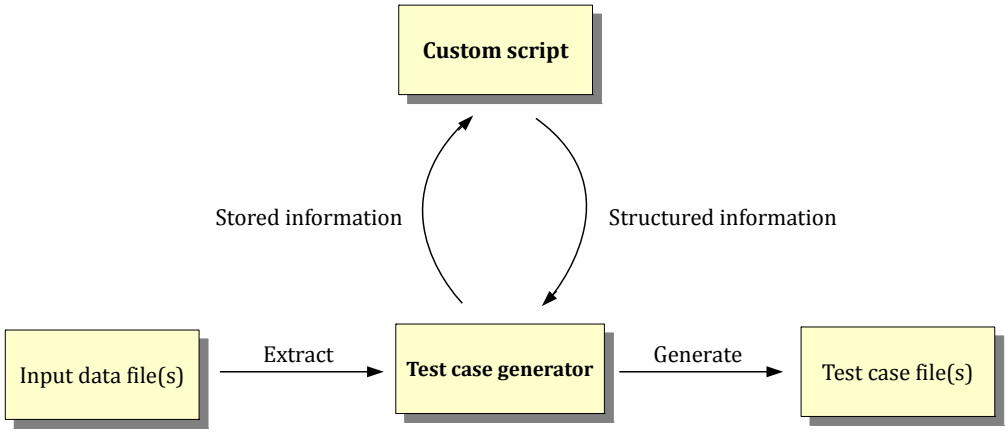


Figure 3.1: Overview of the tool

3.2 Development method

The tool was developed top-down. Setting up the most general aspects of the tool first, such as the architecture, allowed us to implement and test most of the functionality in parallel i.e. run tests for a certain module while developing another one. This did not only save development time, but also allowed us to discover errors at an earlier stage.

3.3 Programming language

It was requested by Bombardier that the tool be implemented with *Python*. There might be several reasons as to why that was, but one important reason was that it had to be able to run on both UNIX and Windows operating systems (*Python* is platform independent [21]). Although the programming language was already chosen by Bombardier, *Python* has a number of desirable qualities when it comes to developing maintainable software which makes it an appropriate programming language for this project [21]:

- *Python* is a high-level programming language. These languages are easier to understand since they are closer to human language than machine language.
- Developing programs in *Python* is faster than in many other programming languages. They are also shorter. This implies that less code has to be maintained, but also that new functionality can be developed and implemented faster if a program is written in *Python*.

Additionally, the syntax in *Python* is also highly readable. To show this, a comparison is made between *C#*, *Java* and *Python* with a simple program that prints a set of candles based on the user's year of birth. The output of the program can be seen in figure 3.2, and the corresponding code for each language can be seen in figures 3.3-3.5 [22, 23]. It might be worth mentioning that *C#* and *Java* are also high-level languages [24, 25].

Program output

```
Födelseår: 1998
Minns du ljusen på dina födelsedagstårtor?
1999 |
2000 ||
2001 |||
2002 ||||
2003 |||||
och i år...
2004 ||||||
```

Figure 3.2: Output from the candle program [22]

Program written in *Python*

```
born = input("Födelseår:")
print "Minns du ljusen på dina födelsedagstårtor?"
year = born + 1
candles="|"
while year < 2004:
    print year, candles
    year += 1
    candles += "|"
print "och i år..."
print year, candles
```

Figure 3.3: Code in *Python* [22]

Program written in *Java*

```
import java.io.*;
class Candles{
    public static void main(String[] args){
        BufferedReader in = new BufferedReader(
                            new InputStreamReader(System.in));
        System.out.print("Födelseår:");
        try{
            String born = in.readLine();
            int year=Integer.parseInt(born)+1;
            System.out.println("Minns du ljusen på dina födelsedagstårtor?");
            String candles="|";
            while(year<2004){
                System.out.println(year+" "+candles);
                year++;
                candles+="|";
            }
            System.out.println("och i år...");
            System.out.println(year+" "+candles);
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
}
```

Figure 3.4: Code in *Java* [22]

Program written in *C#*

```
using System;
public class Candles{
    public static void Main(string[] args){
        Console.Write("Födelseår: ");
        int born = Convert.ToInt32( Console.ReadLine()) +1;
        Console.WriteLine("Minns du ljusen på dina födelsedagstårtor?");
        string candles = "|";
        while (born < 2014) {
            Console.WriteLine("{0} {1}", born, candles);
            born++;
            candles += "|";
        }
        Console.WriteLine("och i år...");
        Console.WriteLine("{0} {1}", born, candles);
    }
}
```

Figure 3.5: Code in *C#* [23]

3.4 Naming convention, variable names & comments

When developing software, it is important to write the code in a clear and structured way. This is because poorly written code can be very difficult to maintain since the programmer will have a tough time understanding it. To illustrate this, consider the two blocks of code in figure 3.6. The code on the left hand side is extremely hard, if not impossible, to understand. Using meaningful variable names and an appropriate naming convention we get the code to the right which is perfectly sensible. Although this is a rather simple example, it illustrates the point quite well. Therefore, a large amount of time was spent on structuring and commenting the code well, finding a good naming convention and choosing descriptive variable names to make the code as readable and understandable as possible. The naming convention used in the tool, and in the example below, is the naming convention used in *C++* [26].

Code comparison

```
if x >= y:                          |  if money >= bookPrice:
    z = x - y                       |      moneyLeft = money - bookPrice
    print "You have %s kr left" % z |      print "You have %s kr left" % moneyLeft
else:                               |  else:
   w = y - x                        |      moneyMissing = bookPrice - money
   print "%s kr is missing" % w     |      print "%s kr is missing" % moneyMissing
```

Figure 3.6: Code comparison

3.5 Architecture of the tool

The architecture of the tool is illustrated in figure 3.7 and a short description of each module can be found in table 3.1. All the functionality of the tool is packaged into modules and divided into a set of layers in a hierarchical structure with the custom script at the top. Designing the architecture in this way allowed the modules to be highly cohesive and at the same time loosely coupled. This design is also expandable, meaning that more modules can be added without causing the overall structure to change or deteriorate.

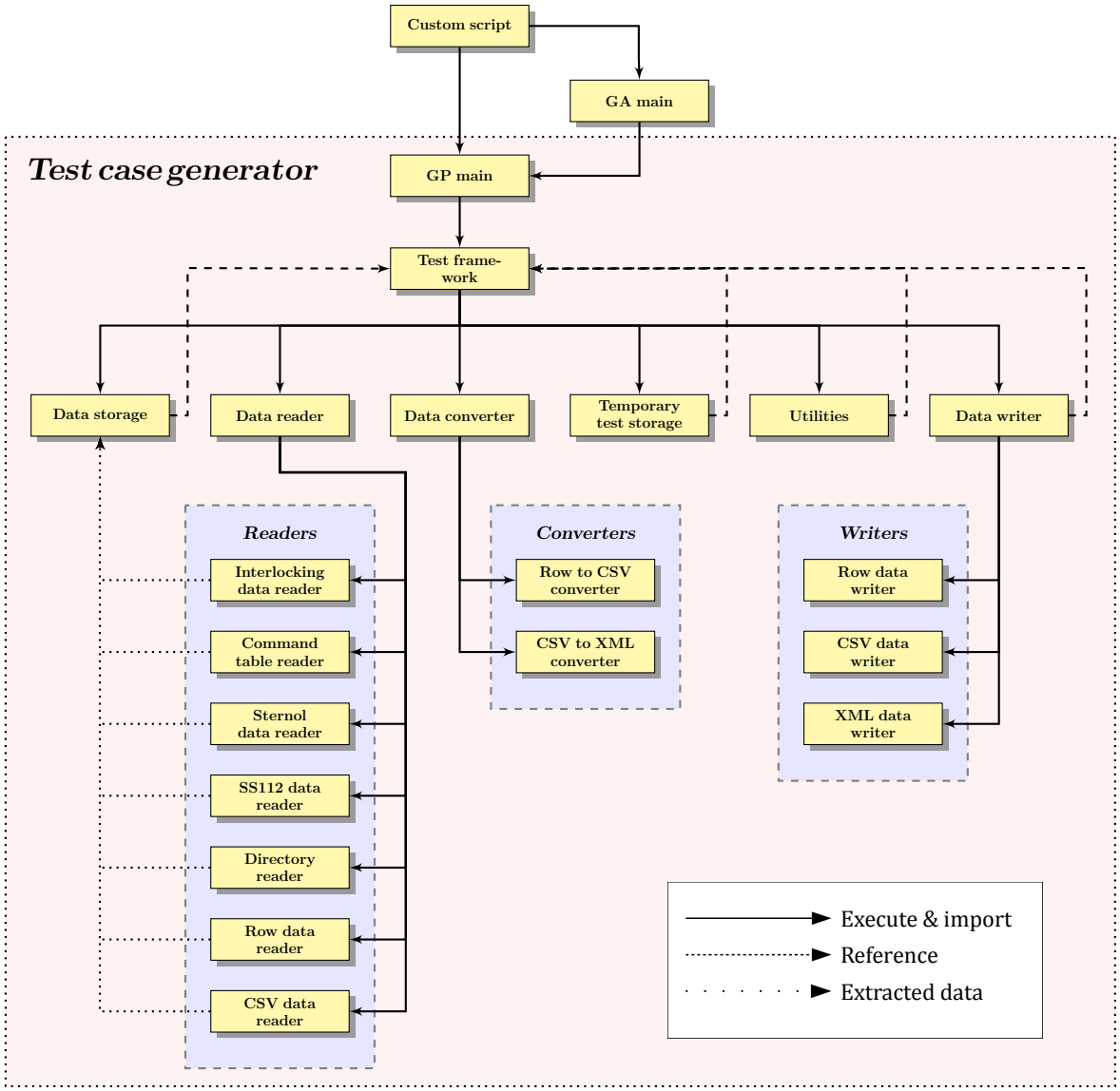


Figure 3.7: Architecture of the tool

Module	Description
Readers	Extract data from input data files
Converters	Convert input data files into desired formats
Writers	Generates the output files
Data reader	Links all the readers
Data converter	Links all the converters
Data writers	Links all the writers
Data storage	Stores extracted data from readers in dictionaries
Utilities	Additional module used to extract specific data
Temp.test storage	Temporarily stores output data until data is written
Test framework	Links the data modules to GP main
GP main	Contains all functions from the data modules used to create tests
GA main (optional)	Contains specific functions for customary tests
Custom script	Executes the test case generator

Table 3.1: Module description

3.6 Tool complexity

The complexity of the tool was measured with a program called *Radon*. *Radon* measures the CC by counting the number of decision points in a block of code (function, method and class) according to equation 2.1 [28, 29]. In each of the modules, the average CC over all the blocks of code was calculated. All the necessary data on how *Radon* ranks the CC score and what counts as a decision point can be found in tables 3.2 and 3.3 respectively [28, 29]. The results from the measurement can be seen in table 3.4.

Construct	**Effect on CC**	**Reasoning**
if	+1	An *if* statement is a single decision
elif	+1	The *elif* statement adds another decision
else	+0	The *else* statement does not cause a new decision
for	+1	There is a decision at the start of the loop
while	+1	There is a decision at the *while* statement
except	+1	The except branch adds a new conditional path of execution
finally	+0	The *finally* block is unconditionally executed
with	+1	The *with* statement roughly corresponds to a try/except block
assert	+1	The *assert* statement internally roughly equals a conditional statement
comprehension	+1	A list/set/dict *comprehension* of generator expression is equivalent to a for loop
Lambda	+1	A *lambda* function is a regular function
Boolean operator	+1	Every *Boolean operator* (and, or) adds a decision point

Table 3.2: Decision points [28]

CC score	Rank	Risk
$1 - 5$	A	Low - simple block
$6 - 10$	B	Low - well structured and stable block
$11 - 20$	C	Moderate - Slightly complex block
$21 - 30$	D	More than moderate - more complex block
$31 - 40$	E	High - complex block, alarming
41+	F	Very high - error-prone, unstable block

Table 3.3: CC score scale [29]

Module	Average CC score	Rank
GP main	1.2	A
Test framework	3.9	A
Data storage	1.1	A
Data reader	1.4	A
Data converter	1.0	A
Temp. test storage	1.0	A
Utilities	2.0	A
Data writer	1.0	A
Interlocking data reader	9.6	B
Command table reader	7.2	B
Sternol data reader	4.9	A
SS112 data reader	4.0	A
Directory reader	2.7	A
Row data reader	3.0	A
CSV data reader	3.0	A
Row to CSV converter	1.7	A
CSV to XML converter	4.7	A
Row data writer	1.3	A
CSV data writer	1.3	A
XML data writer	1.2	A
Average score	2.9	A

Table 3.4: CC measurement results

3.7 Maintainability of the tool

After the tool was developed and thoroughly tested, its maintainability was measured through the maintainability index (MI). MI is a metric that was first introduced in 1991 by Oman and Hagemeister [27]. It has since then been validated by Hewlett Packard in an extensive trial and was for over a decade successfully used on large-scale military and industrial systems [17, 27]. The MI measurement was also carried out with the *Radon* program and is calculated from [28, 29]:

$$MI = max\left[0, 100\frac{171 - 5.2\ln(V) - 0.23G - 16.2\ln(L) + 50\sin(\sqrt{2.4C})}{171}\right]$$

where V is the total Halstead volume, G is the total cyclomatic complexity, L is the number of source code of lines and C is the percent of comment lines(converted to radians). The Halstead volume V is given by:

$$V = (N_1 + N_2)\log_2(\eta_1 + \eta_2)$$

where N_1 is the total number of operators, N_2 is the total number of operands, η_1 is the number of distinct operators and η_2 is the number of distinct operands. How the MI score is ranked is described in table 3.5 and the results for each module can be seen in table 3.6.

MI score	**Maintainability**	**Rank**
$20 - 100$	Very high	A
$10 - 19$	Medium	B
$0 - 9$	Extremely low	C

Table 3.5: MI score scale [29]

Module	MI score [0-100]	Rank
GP main	23.2	A
Test framework	54.7	A
Data storage	42.4	A
Data reader	71.8	A
Data converter	100.0	A
Temp. test storage	100.0	A
Utilities	87.5	A
Data writer	100.0	A
Interlocking data reader	35.9	A
Command table reader	57.9	A
Sternol data reader	54.9	A
SS112 data reader	60.7	A
Directory reader	96.0	A
Row data reader	84.5	A
CSV data reader	84.6	A
Row to CSV converter	94.2	A
CSV to XML converter	69.7	A
Row data writer	79.5	A
CSV data writer	79.5	A
XML data writer	65.2	A
Average score	**72**	**A**

Table 3.6: MI measurement results

3.8 User manual

All of the tool's functionality was also documented in a user manual. Technical details such as the architecture and various sequence diagrams illustrating how the tool executes were also included to give its users a quick, but basic understanding of the program. The user manual was also read and approved by Bombardier.

3.9 Tool summary

The tool has been carefully tested and it seems to work properly. Its maintainability was measured with the MI metric and the results show that it is highly maintainable. A summary of the tool with regard to the maintainability factors can be found in table 3.7.

Maintainability factors	Summary
Modularity	With regards to the tool's modularity, the program was divided into several modules making it easier to understand, document and test.
Coupling	The modules were divided into several layers to keep them loosely coupled. This reduced their dependency on each other hence, making them easier to modify.
Cohesion	The modules were divided in such a way that most modules could perform a specific task. Errors in the tool can therefore be identified and corrected faster.
Readability	It is important that the program is readable. Using a standard naming convention as well as descriptive variable names and comments made the source code easier to understand. Furthermore, a user manual was written to assist the users.
Complexity	A great amount of emphasis was put on the structure of the code to increase the tool's simplicity. To verify this, the tool's complexity was measured with CC, and results indicate that it is low.

Table 3.7: Summary of the tool

Chapter 4

Testing with the tool

The aim of the tool is to reduce the testing time of CBI-systems at Bombardier. To address this, three tests were carried out. The time it took to complete each test with and without the tool was measured and then compared. However, the time for tests without the tool had to be approximated due to time constraints. This was done by measuring the time it took to write down a hundred test cases and based on that time, the total writing time for all of the test cases could be estimated [30].

The test cases generated by the tool were also sent into the test engine to see if any of them failed. This is because failed test cases reveal the source of error whether it be from the tool, CBI-system or input data. A test case fails if all the conditions specified in it are not met by the CBI-system. The results and a short description of each test can be found in sections 4.1-4.3.

4.1 Track circuit test

Along the railway there are devices called track circuits. These devices indicate where a train is located. All the track circuits were tested by occupying and releasing them. The results show that the track circuit test was performed $43/16 \approx 2.7$ times faster with the tool.

Method	**Test cases**	**Failed test cases**	**Time[h]**
With tool	7782	0	16
Without tool	7782	-	43

Table 4.1: Results from the track circuit test

4.2 Approach-locking test

Approach locking prevents trains from suddenly braking by disallowing a signal to indicate stop if a train is too close to it. All signals were tested for this. The results show that the approach-locking test was performed $60/24 = 2.5$ times faster with the tool.

Method	**Test cases**	**Failed test cases**	**Time[h]**
With tool	10815	758	24
Without tool	10815	-	60

Table 4.2: Results from the approach-locking test

4.3 Level-crossing test

Intersections between the road and the railway are called level-crossings. The barriers found at the level-crossings were tested to ensure that they function as intended. The results show that the level-crossing test was performed $421/40 \approx 10.5$ times faster with the tool.

Method	**Test cases**	**Failed test cases**	**Time[h]**
With tool	75837	4280	40
Without tool	75837	-	421

Table 4.3: Results from the level-crossing test

Chapter 5

Discussion

The discussion is divided into two parts. In the first part the maintainability of the tool and the MI metric are discussed. The second part takes a closer look at the results from chapter 4 and discusses the performance of the tool.

5.1 Maintainability of the tool

A problem with software maintainability is that it is very difficult to accurately measure. This is because it is affected by numerous factors which, to some degree, can be subjective. In this thesis, the maintainability was measured through the MI formula which showed that the tool was highly maintainable. However, in my opinion the MI metric does not give the full picture regarding maintainability and should merely be used as an indicator. This is because a number of important factors cannot be extensively measured. For example, although variable names and comments are used, there is no indication regarding their quality i.e. if they are good or bad. Furthermore, MI does not take into account supplemental information such as the user manual and other important documents that aid the maintenance process. To further validate that the tool is maintainable, a computer scientist and a software engineer at Bombardier were asked a series of questions regarding the tool's readability, modifiability and life-span. There was a general consensus among them regarding the tools readability and modifiability i.e. they agreed that tool is easy to read and understand as well as easy to modify. Regarding the life-span of the tool, the computer scientist estimated that the tool would last for $5 - 10$ years without any major changes while the engineer estimated that the tool would last at least two decades.

With the results obtained with the MI metric as well as the positive feedback from the computer scientist and software engineer, it is reasonable to conclude that the tool is easy to maintain.

5.2 Tool performance & results

The results from the tests in sections 4.1-4.3 are very positive and clearly show that generating the corresponding test cases to each test with the tool is a faster option than that of writing them down. However, the tests that were performed were rather large. If a test is sufficiently small, it may in fact be faster to manually write down the test cases instead. This is because it takes time to create and test the custom script as well as gather all the necessary input data.

With regards to the failed test cases, there was not enough time to examine all of them. However, none of the examined test cases failed due to errors in the tool. In fact, the source of error was mainly found to be in the modeling of the interlocking logic. This suggests that the tool works correctly.

Another great benefit that comes with the use of the tool are the created custom scripts. When a custom script is written and tested for one project, it can be reused in the next one with minimal modification. This can greatly reduce the testing time as some tests in the next project may run straight away.

Chapter 6

Conclusion & future work

6.1 Conclusion

In this thesis a software tool that automatically generates test cases has been developed to aid Bombardier in the verification process of CBI-systems. The tool has proven to be useful since the results show that it can significantly reduce the testing time of CBI-systems as tests can be performed faster. This in turn may not only save Bombardier resources, but also enables them to increase the degree of verification as more tests can be run using the same amount of time. With the positive results and feedback regarding the maintainability, the tool will also be easier to change and modify to meet Bombardier's needs and expectations for many years to come. Bombardier is very satisfied with the outcome of the project and the tool has now become an official product used at the company.

6.2 Future work

There are some additional enhancements that can be made to the tool in order to increase its maintainability and reduce the testing time of CBI-systems:

- **Documenting the tool**
  Different programmers have different styles. The source code can quickly become confusing if modifications are carried out using different naming conventions. This can probably be avoided by documenting how the naming convention is used in the tool properly. Other aspects of the code could also be documented in more detail as it may facilitate the maintenance work. It is also important to remember to keep the documentation updated and in accordance with the current version of the tool.

- **Improving the source code**
  There might still be some improvements that can be made in the source code. Adding more descriptive comments, improving variable names if needed and optimizing the various functions and methods to reduce their complexity are only a few examples. There is also a possibility of discovering unnoticed bugs which have to be corrected.

- **Keeping the tool maintainable**
  Inspecting the code regularly to see if modifications in the tool have been implemented in a maintainable way might be wise. This can prevent the source code from deteriorating due to quick fixes and preserves the maintainability of the tool.

- **Adding more readers**
  If some of the input data for a test cannot be directly extracted with the tool, it will have to be written down manually which can be very time consuming. Adding more readers to the tool increases the range of different input data that can be extracted by the tool. This in turn allows tests to be performed even faster.

Bibliography

[1] Lutovac, Dejan and Tatjana Lutovac. *Towards an Universal Computer Interlocking System*. Facta Universitatis, Series: Electronics and Energetics 11.1 (1998): 25-49.

[2] Khan, Sher Afzal and Nazir A. Zafar. *Towards the formalization of railway interlocking system using Z-notations*. Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on. IEEE, 2009.

[3] Banguide.
`http://www.jarnvag.net/index.php/banguide/signaler`.
Accessed 02 December 2014.

[4] Trafikanalys. Bantrafikskador 2012.
`http://trafa.se/PageDocuments/Bantrafikskador_2012.pdf`.
Accessed 02 December 2014.

[5] Grubb, Penny and Takang A. Armstrong. *Software maintenance: Concepts and practice*. ISBN 981-238-425-1. World Scientific Publishing Co, 2003.

[6] Oman, Paul and Hagemeister, Jack. *Metrics for assessing a software system's maintainability*. Proceedings Conference on Software Maintenance, pp.337-344, Nov 1992.

[7] Bengtsson, PerOlof, et al. *Analyzing software architectures for modifiability*, 2000.

[8] *IEEE Std 1219: Standard for software maintenance*, 1998.

[9] Pigoski, Thomas M. *Practical software maintenance: best practices for managing your software investment*. ISBN 0-471-17001-1. John Wiley & Sons, Inc., 1996.

[10] Koskinen, Jussi. *Software maintenance costs*. Information Technology Research Institute, ELTIS-Project University of Jyväskylä (2003).

[11] *Guideline on Software Maintenance*. Federal Information Processing Standards Publications(FIPS PUB 106), 1984.

[12] *ISO/IEC 9126-1: Information technology - Software product quality - part 1: Quality model*, 2000.

[13] *IEEE Std 610.12: Standard Glossary of Software Engineering Terminology*, 1990.

[14] The free dictionary
`http://www.thefreedictionary.com/module`.
Accessed 06 December 2014.

[15] Hashim, Khairuddin, and Elizabeth Key. *A software maintainability attributes model*. Malaysian Journal of Computer Science, Vol. 9, December 1996.

[16] Bharat Bhushan, Agarwal and Sumit Prakash, Tayal. *Software Engineering, second edition*. Firewall media, 2009.

[17] Laird, Linda and Brennan, Carol. *Software Measurement and Estimation: A Practical Approach*. ISBN: 0-471-67622-5. John Wiley & Sons, Inc., 2006.

[18] Ghosh, Soumi, et al. *Fuzzy Maintainability Model for Object Oriented Software System*. International Journal of Computer Science Issues, Vol. 9, No. 2, July 2012.

[19] Peercy, David. *A Software Maintainability Evaluation Methodology*. IEEE transactions on software engineering, Vol.SE-7, No. 4, July 1981.

[20] Riguzzi, Fabrizio. *A survey of software metrics*. Universita degli Studi di Bologna DEIS, 1996.

[21] Dawson, Michael. *Python programming for the absolute beginner*. Cengage Learning, 2010.

[22] Eriksson, Henrik. *Pythonkramaren: Del ett*. Royal Institute of Technology, 2009.

[23] Friezadeh, Farhad. Electrical Engineer, 2014.

[24] Oracle. The Java programming language.
`http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html`.
Accessed 12 December 2014.

[25] Microsoft. Visual C# resources.
`http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx`.
Accessed 12 December 2014.

[26] Misfeldt, Trevor, et al. *The elements of C + + style*. ISBN 9780511547027. Cambridge University Press, 2014.

[27] Kaur, Kulwant, and Hardeep Singh. *Determination of Maintainability Index for Object Oriented Systems*. ACM SIGSOFT Software Engineering Notes, Vol. 36, Nr. 2, March 2011.

[28] Radon tool documentation.
`http://radon.readthedocs.org/en/latest/intro.html`.
Accessed 11 December 2014.

[29] Radon tool documentation.
`http://radon.readthedocs.org/en/latest/commandline.html`.
Accessed 11 December 2014.

[30] Bakhiet, Mohamed. *Automated Tests of Computer-Based Interlocking Systems: Developing a test case generator*, 2014.