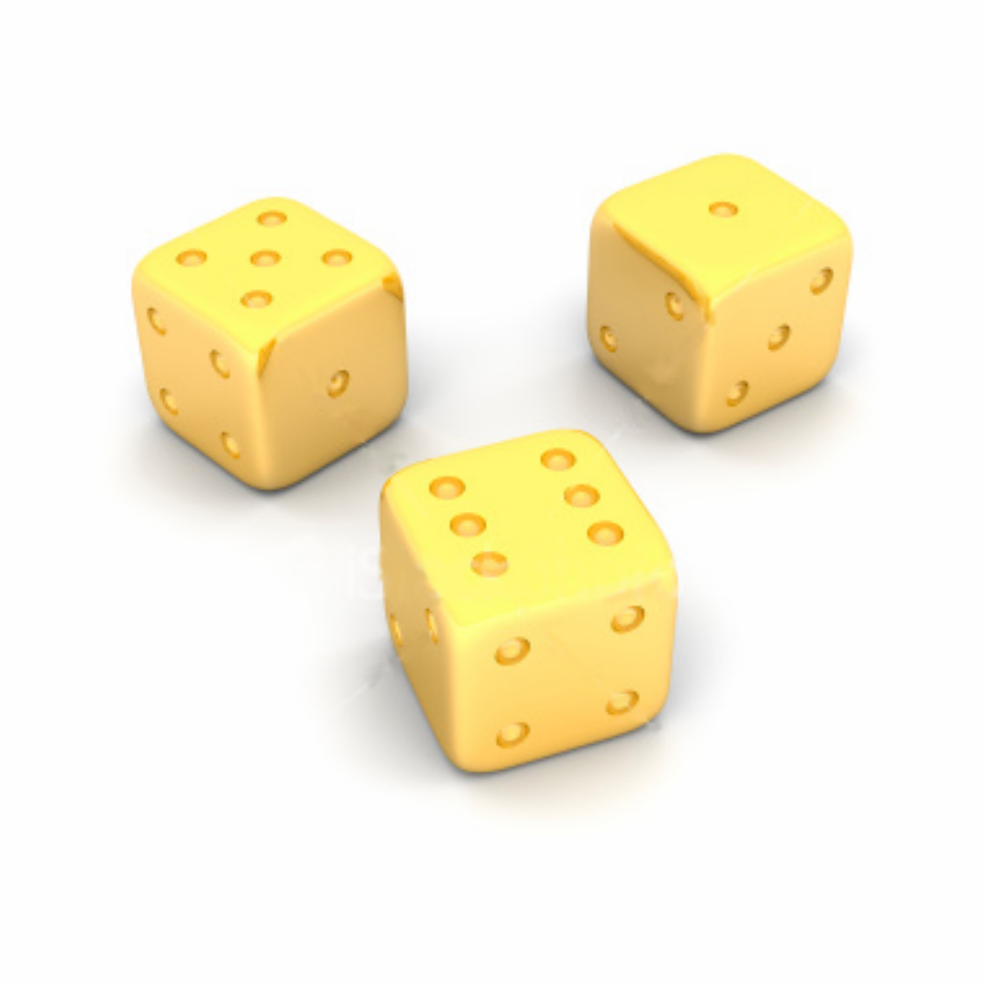
Design for Testability in Software Systems



Emmanuel Mulo

Design for Testability in Software Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Emmanuel Mulo
born in Kampala, Uganda

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Philips Medical Systems Nederlands B.V
Veenpluis 4-6
Best, The Netherlands
www.medical.philips.com

Design for Testability in Software Systems

Author: Emmanuel Mulo
Student id: 1290363
Email: E.O.Mulo@student.tudelft.nl

Abstract

Building reliable software is becoming more and more important considering that software applications are becoming pervasive in our daily lives. The need for more reliable software requires that, amongst others, it is adequately tested to give greater confidence in its ability to perform as expected. However, testing software becomes a tedious task as the size and complexity of software increases, therefore, the next logical step is to make the task of testing easier and more effective. In other words, improving on the *testability* of the software. Improvement of testability can be achieved through applying certain tactics in practice that improve on a tester's ability to manipulate the software and to observe and interpret the results from the execution of tests.

Thesis Committee:

Chair:	prof. dr. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	dr. Andy Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Nico van Rooijen, MR Software, Philips Medical Systems
Committee Members:	dr. ing. Leon Moonen, Faculty EEMCS, TU Delft dr. Koen G.Langendoen, Faculty EEMCS, TU Delft

Preface

When confronted with the kind of work that I have been confronted with while doing this thesis, I start to comprehend one famous quote that I have seen written down many times; *"If I have seen further it is by standing on ye shoulders of Giants"*. If there is one lesson I have learnt from the experience, that is it!

I am grateful to the people who have contributed their efforts in bringing this work together. Nico, who struck me from the very first time I listened to him speak and was constantly pushing and steering my thinking when I veered off track, while at the same time leaving up to me to go as far as I wanted; Andy who gave (very detailed) insightful reviews and encouragement when things seemed not to be progressing; Arie for answering my call as I was searching through potential graduation projects; Maria, Marco and Christian who gave peace of mind and great company in my last months in the wonderful town of Best; all my great "kamergenooten" Jaap, Roel, Eric, Martin and Danny for the informal "inburgeringscursus" as well as getting involved in my work; and finally, *very heartfelt* thanks to my parents and sisters; every time I spoke to any one of them I was completely recharged with the energy I needed to push on.

Emmanuel Mulo
Best, the Netherlands
November 15, 2007

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
- 1.1 Background and Problem Statement . . . 1
- 1.2 Software Testability . . . 2
- 1.3 Research Question(s) . . . 6
- 1.4 Objectives, Scope and Methodology . . . 6
- 1.5 Overview of Chapters . . . 7

2 Philips Medical Systems 9
- 2.1 Organisation overview . . . 9
- 2.2 MR Software . . . 9
- 2.3 Software Testing at PMS . . . 10
- 2.4 Concluding Remarks . . . 12

3 Related Work 13
- 3.1 Design Testability . . . 13
- 3.2 Built-in Test . . . 17
- 3.3 Test Frameworks . . . 19
- 3.4 Concluding Remarks . . . 21

4 Built-in Test Design 23
- 4.1 Introduction . . . 23
- 4.2 Requirements for Built in Test Scheme . . . 23
- 4.3 Design Scheme of a Built in Test Infrastructure . . . 26
- 4.4 Process Approach to Incorporating Built in Test Infrastructure . . . 30
- 4.5 Analysis of Built in Test Approach . . . 31

4.6 Concluding Remarks 32

5 Case Study: Applying to Philips Medical Systems **33**
5.1 Overview of New System UI Component 33
5.2 Applying Built in Testing to New System UI 35
5.3 Evaluation of Built in Testing in NSUI 37
5.4 Concluding Remarks 41

6 Conclusions and Future Work **43**
6.1 Contribution 43
6.2 Conclusion 44
6.3 Future work 44

Bibliography **47**

A Code Snippets **51**
A.1 PackageCommand class 51

List of Figures

1.1 Multi-dimensional view on Testability [26] 3

2.1 Architecture of the MR System 10

3.1 Testability Tactics [1] 15
3.2 BIT Fishbone [5] 17
3.3 Class Structure of BIT-Embedded Framework Extension [25] 19
3.4 JUnit Testing Framework Architecture [3] 21

4.1 Overview of Built in Test Concept 24
4.2 Class Diagram of Proposed BIT Scheme 27
4.3 Setting up BIT Infrastructure 29
4.4 Illustration of a Class Hierarchy with BIT Classes attached 29
4.5 Test Execution Sequence 30

5.1 NSUI and Interacting Components 34
5.2 UIBackbone Class Structure 34
5.3 Setup of NSUI BIT Objects 36
5.4 A component diagram for BIT in NSUI 37

Chapter 1

Introduction

1.1 Background and Problem Statement

Building reliable software is an important issue considering that computer applications are now used in all kinds of environments, including some where human life depends on the computer's correct functioning. Software is considered reliable if it has a low probability of failure while it is being used [23]. These failures only occur if, for example, faulty code in the software is executed. Identification of these faults in the software can be done through formal proofs or through selective or exhaustive testing [30]. Formal proofs are rather complicated to perform and exhaustive testing is not feasible because of the large number of execution paths that exist even in software comprised of relatively few lines of code. Selective testing also known as dynamic testing (or simply software testing) is the most common method of improving confidence in the reliability of software.

1.1.1 Software Testing

Software testing involves executing the software and observing how it reacts. If the software behaves as expected, confidence in its reliability is increased. Software testing has progressed from being a post-development activity to an activity that is integrated throughout the development phases. In the traditional waterfall software development life cycle, software testing was done after a software system had already been built. More recently defined software development methodologies and/or processes[1] recommend that testing be done at various stages throughout the development cycle. The V-model, for example, recommends that testing should be done at unit, integration and system levels [7]. At each testing level of the V-model, a different level of granularity of software component[2] is subjected to testing activities.

Test driven development, another recently defined software development methodology, advocates for the development of unit tests *even before* the functional code is written [4].

[1] The term *process* as used in this document refers to the software development lifecycle and is not to be confused with process referring to how work flows are structured in an organisation.

[2] Unless otherwise specified, the term *component* is used throughout this document to refer to classes, modules or subsystems

After creating unit tests, the developer writes the functional code necessary to ensure that all the previously created unit tests are passed. If the unit tests have all passed and the developer cannot think of any more to create, then the development work is considered complete. Test driven development is said to improve the quality of software by forcing developers to take certain design decisions based on the testing need. Also the tests written provide a form of documentation for the software.

Regardless of how testing is integrated into a software development cycle, there are a number of difficulties incurred when testing software. Software testing is based on assumed inputs that may give false test results if the assumptions are wrong [36]. Testing may reveal failures but does not necessarily help with pointing out where the faults in the software are located. Moreover, testing requires that there is an oracle present to specify what the expected results from the test are. In worse cases, testing may not even reveal failures because the faults either do not get executed, or get executed but deceptively give correct results for the inputs that are chosen [6].

The size of software systems also affects testing. As systems get larger testing becomes more expensive. Estimates of the cost of software testing range from 40% to 80% of the entire development costs [41]. Unit testing, for example, is very expensive in terms of effort to maintain the test code. Besides test code maintenance, relatively large parts of the entire system have to be (re)tested even for small additions to the overall source code base. Finally, the additional software units may seem correctly implemented, only to discover malfunctions after they (units) have been composed together and tested at sub system level.

Software testing can, therefore, be viewed as an economic problem that is driven by either a predefined reliability target or by resource constraints [5]. In the former instance resources are devoted to the testing until the reliability target is achieved, whereas in the latter instance, testing is performed until the resources dedicated to it (testing) are depleted.

1.2 Software Testability

Voas and Miller [38] introduce *software testability*, as the third piece of the *reliability puzzle*, alongside the previously mentioned formal verification and software testing. They define testability as the probability that the software shall fail on its next execution [38].

Testability is defined in the IEEE glossary of software engineering [24] as;

> *(1) degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and the performance of tests to determine whether those criteria have been met.*

Testability can also be thought of as a characteristic or property of a piece of software that makes it easier to test. The authors of [5, 15] adapt the notions of *controllability* and *observability* from hardware testing into the software context. Controllability is the ability to manipulate the software's input as well as to place this software into a particular state, while observability deals with the possibility to observe the outputs and state changes that

occur in the software. From their definitions, a piece of software is said to be testable if it has these two properties. Other work adds extra characteristics like *understandability, traceability* and *test-support capability* [18] to the testability notion.

Besides being a characteristic of software, testability can be defined in terms of the software development process, that is, relative to the testing phase, for example, integration test, relative to an activity of the test phase, for example, test case creation, and as a characteristic of the architecture, design, and implementation of a piece of software [26]. Binder [5] talks about six major factors that result in testability in the development process. The factors are: characteristics of the design documentation, characteristics of the implementation, built-in test capabilities, presence of a test suite, presence of test tools and the software development process capability/maturity.

Looking at testability in the context of the development process, one can talk about high testability of java classes (implementation) in a system as far as creation of test cases (test phase activity) during unit testing (testing level). This high testability at implementation level might not necessarily imply high testability of the system modules for the same system when tested at integration level. Figure 1.1 illustrates this explicit, multi-dimensional view of testability. Such an explicit definition of testability is assumed to be understood implicitly by individuals based upon the level and activity at which they are working in the software development process.

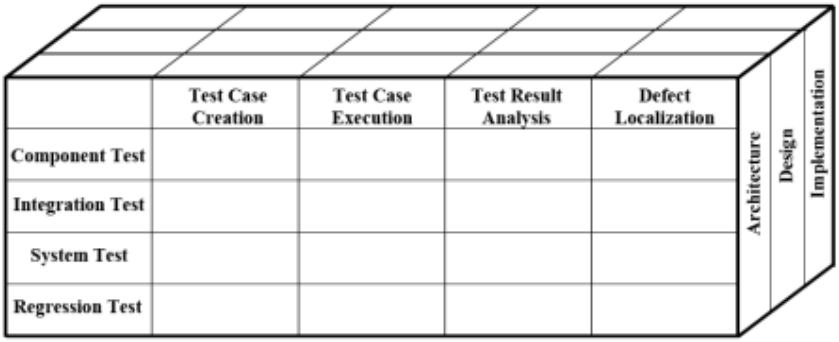


Figure 1.1: Multi-dimensional view on Testability [26]

Testability acts as a complement to software testing in that it eases discovery of faults in software by helping to focus the testing efforts on areas that are most likely to yield these faults [38]. Typically, resources for developing software are limited. Testability enables achievement of greater reliability in such a resource-constrained development environment. Besides this, testability also reduces the overall costs needed to attain a predefined target in a reliability-driven development environment. In other words, testability helps to make testing more effective and efficient. The testability characteristic of software leads to the notion of design for testability. Binder [5] defines design for testability as “a strategy to align the development process so that testing is maximally effective”. This strategy involves, for example, incorporating certain features, in software implementations and/or software designs, that shall improve on the testability of software.

In the following sections, two key issues in software testability (controllability and observability) are discussed. We also look at testability in the context of activities in the software development process.

1.2.1 Controllability and Observability

In this section we zoom in on two themes that form a common thread throughout testability namely: controllability and observability.

Controllability

Controllability is concerned with the ease of manipulating a software component in terms of feeding values to its *inputs* and, consequently, placing the component in a *desired state*. Inputs into components can be in the form of parameters to class methods or functions. These inputs come from user interaction with the software's user interface or from other interactions of components within or external to an application. The inputs are stored and manipulated in variables and are modifiable depending on the scope in which they are located. In object-oriented programming, for example, techniques like encapsulation prevent that variables are modified by external unapproved methods.

State of software can be considered as a value set [6], that is, the set of variables and corresponding values at a particular instance in time. Whereas variables in a software module or unit can have unlimited combinations of values, only a subset of combinations, that have a significant meaning, are of interest to developers of software. Take the case of software that is operating on bank accounts. When an account holder withdraws money, the bank software marks his/her account's state as *active* as long as the balance is still sufficient. The account's state change is only interesting when the user no longer has a positive balance. The account might then be marked as *overdrawn*. The ability to manipulate software components' inputs and states implies that one has greater control of the software and can, therefore, generate efficient test sets [34].

Observability

Observability is the ability to view the reactions of software components to the inputs that are fed in and also being able to watch the changes to the internal states of the software. Typically software outputs would provide observability, however, there are some erroneous internal states or interactions that do not surface as failures and, as such, are hard to trace back and solve. Voas [36] mentions three necessary conditions for a failure to occur: there must be execution of faulty code in software, this execution should lead to a corrupt state of data and finally the corrupt state should be propagated to the output.

By having measures in place to enhance observability of internal states and interactions, hidden faults can be detected. Observability is, therefore, linked to interpretation of results from test runs [34] or executions of the software. Software observability can be enhanced through various ways including; having logging facilities, tracing facilities, code instrumentation, probes [11] and, using assertions [37].

1.2.2 Testability Factors

Testability should also be considered in the context of the software development process. In [5], six testability factors are listed that can either facilitate or hinder testing. We discuss these factors briefly.

Characteristics of Software Design Documentation

Software structure and behaviour can be documented in form of requirements specifications, architectural views, detailed design models and pseudo-algorithms. The clarity and conciseness of this documentation can be a facilitator for testability. Some of the characteristics of the documentation that can enhance testability include: unambiguity in requirements, thoroughness of detailed design, traceability of the requirements to the implementation and, separation of concerns [5]. These characteristics are not only related to testability but also refer to fundamentals of design practice.

Characteristics of the Implementation

Software is implemented with a variety of programming languages and paradigms. The features available in these languages and paradigms make for easier or more difficult testing depending on how they are used. For example, in object oriented systems, inheritance as a feature can cause problems for testing if there is a very deep inheritance tree. This is due to the fact that classes that are lower in the hierarchy need to behave similar to base classes higher up in the hierarchy, and as such, these classes need to be retested in case of changes in a parent class.

Built-in Test Capabilities

Building test capabilities into software implies that certain features, that are not necessarily in the functional specifications of the software, are added to the software to ease testing. For example, having extra set/reset methods [5] that allow classes in object oriented software to be placed in a desired state. Besides having extra methods, built-in test capabilities could also involve hard coding test cases into the software [19, 25, 39].

Test Suite

The test suite is a collection of test cases and plans to use them. Along with an oracle for predicting the test results, a test suite can make testing of a system much easier. The test suite is also valuable because it can be reused for regression tests every time a system has been modified. If this regression testing is efficiently automated it represents a saving in test efforts.

Test Tools

Having tools in place that (automatically) execute test scripts, record execution traces, log state and performance information and, report test results makes it a lot easier to handle testing.

Software Development Process Capability/Maturity

There should be a constant commitment to software testing by an organisation producing software. This should be reflected in the resources dedicated to testing, like capable staff, presence of a systematic testing strategy and integration of testing throughout the entire development process.

1.3 Research Question(s)

The research on testability has been prompted by issues that are raised in the following questions.

Question 1*: How do decisions made at design time influence the testability of software?*

During the design stages of software, it is represented in terms of requirement specifications, architectural and detailed design diagrams. These representations capture the structure and behaviour of the software before it is implemented. The representations are then transformed into the actual software implementation. The challenge is to study how these representations impact the final implementation of the software, with the aim of identifying characteristics and/or patterns in the representations that may enhance or perhaps impair testability. Identifying such characteristics and/or patterns would enable one to create representations of software that evolve into better testable implementations and thus improve on the time- and effort-efficiency during software testing.

Question 2*: How can built-in tests and test frameworks be effectively incorporated in a software system to enhance testability?*

Built in testing involves adding extra functionality within system components that allow extra control or observation of the state of these components [5]. In conjunction with a testing framework, these might provide a great enhancement to the testability of a system. It is, therefore, interesting to investigate how these features can be applied to large systems at higher level of granularity, specifically, during integration testing, to see their impact on testability in such a system.

1.4 Objectives, Scope and Methodology

The aim of this document is to report on an investigation into methods of improving testability in large software systems. The investigation was done at Philips Medical Systems; one of the business units of Royal Philips Electronics. The investigation had two main phases that are; a literature study in the area of software testability, and a practical project to investigate some of the theoretical work from the literature.

The literature study resulted in a document providing an overview in the area of software testability. This document was compiled by reviewing literature, mostly in the area of software testability. Areas specifically covered included, how testability can be measured and, how it can be incorporated into a software system. Also covered was literature about software testing, reliability and, design and architectural patterns among others. As far as software testing is concerned, the focus of the literature (and later the practical project) was mainly on integration testing. Test driven development strategies were left out of the survey due to the fact that the organisation, in which the investigation was done, has a defined business process model of how software development activities are performed. More over, this research aimed to find out what options are available for improving testability within the context of this already existing model of work.

The project work, following the literature study, focused on built in testing as an area to investigate further. The objective of this project work was to investigate the impact of built in testing on testability of a large software system. A design was made for incorporating built in tests in such a system. This design was experimented with on a large system. The issues observed during the design and experiment phases are reported in this document.

Throughout the document, wherever systems or programming paradigms are discussed, object-oriented systems or programming techniques are the main focus.

1.5 Overview of Chapters

The remaining chapters in this document are as follows. Chapter 2 gives a background of the problem of software testability in the specific context of Philips Medical Systems where the literature study and practical project work were done. Chapter 3 discusses work done by other researchers that is related to the subject matter of this thesis. Chapter 4 proposes a design for incorporating testability into a software system. Chapter 5 reports on the experience of applying this design to a large software system. Finally, some conclusions are made in chapter 6 along with some suggestions of how this work can be followed up.

Chapter 2

Philips Medical Systems

This chapter presents a brief overview of Philips Medical Systems and the need for improving on testability.

2.1 Organisation overview

Philips Medical Systems (PMS) is one of the business units that comprises Royal Philips Electronics. PMS is involved in the production of medical imaging systems like X-ray machines, computer tomography (CT), magnetic resonance (MR) and ultrasound imaging equipment. These systems are used to create images of various parts of the body in varied detail for radiologists and cardiologists among others.

Magnetic resonance (MR) scanner technology is developed by the MR department within PMS. MR technology uses a combination of magnetic fields and radio frequency (RF) waves to create clinical images of body parts like the brain, spine and joints, that are used in analysis and diagnosis of problems with these body parts. Within the MR department is a group focused on the creation of software that controls the MR hardware and provides other necessary functionality to the cardiologists, radiologists and all other medical personnel who work with these scanners.

2.2 MR Software

The MR software comprises a number of components that are distributed over the entire MR system shown in Figure 2.1. The system comprises of the following components each located on a different machine:

B-DAS/C-DAS: B/C Data Acquisition System[1] responsible for controlling the scanner hardware and generating a stream of raw image data.

Host: computer that provides user functionality like exam card creation, patient data viewing, and scan preparation to radiographers in hospitals.

[1] the letters "B" and "C" indicate the version of the data acquisition system. C-DAS is the more recent version that is currently in use.

Reconstructor: computer that receives a stream of raw image data from the B/C DAS and creates images to be used on the host computer.

DVD-box: is used to write images onto a DVD.

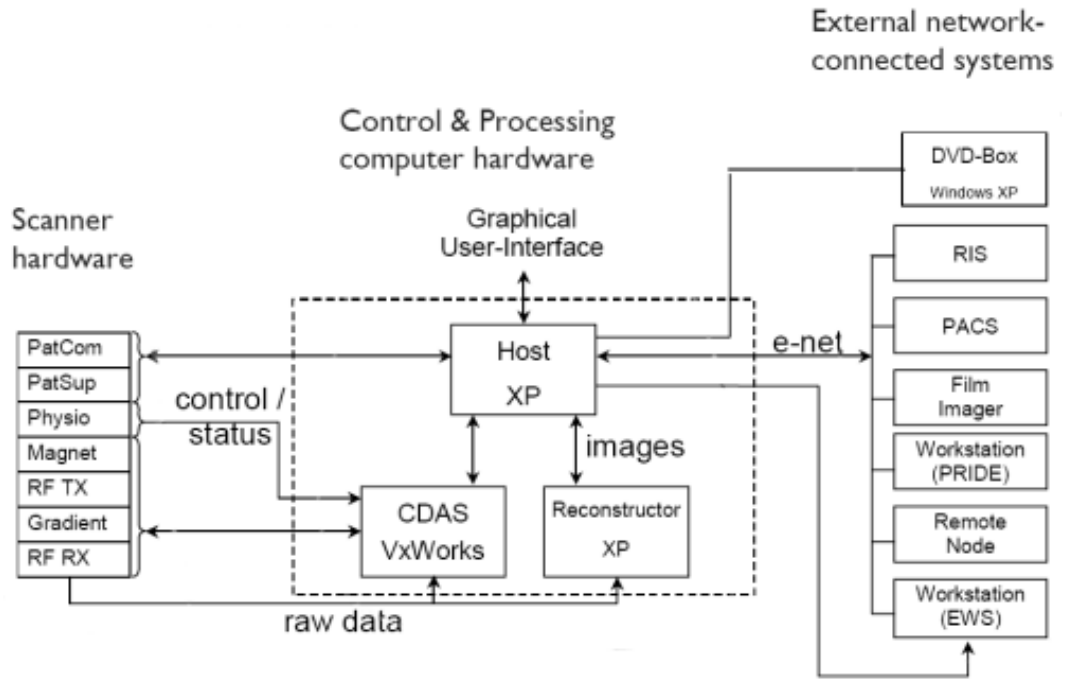


Figure 2.1: Architecture of the MR System

All the source code for the various system components is stored in one software archive. The software archive is approximately 8,000KLOC, consists of over 80 processes when executing and is implemented using a variety of programming and scripting languages (C/C++, C#, Perl), user interface frameworks (C# Winforms, MFC) and data storage mechanisms (Sybase, XML). For ease of management of software development tasks, the entire archive is divided into a number of smaller (sub) modules known as building blocks.

The MR software archive has provisions to handle a number of different configuration options depending on; the need of the clients, the (computer and scanner) hardware being used and the hospital environment in which the system shall operate. These configurations make the archive very versatile but also very difficult to test adequately.

2.3 Software Testing at PMS

2.3.1 The State of Software Testing

Software testing in the MR software department is currently performed using a number of different configurations of test environments. These test environments are made available

based on the type of tests that need to be carried out. In order of increasing costs and scope of testing, the test environments available are; developers/SDE PC (virtual machine), work stations (WS), software test model (STM), reconstructor test model (RTM) and "ontwikkel" test model (OTM).

The test environments mentioned enable the different levels of testing. Levels of testing identified include; developer unit testing, integration testing, system testing and clinical/acceptance testing.

Each developer uses his/her PC to perform immediate tests on units that he/she has produced. This can be done through an instance of a virtual machine on the developer's PC. This configuration behaves similar to the host computer from figure 2.1, providing the developer with the same graphical user interface that would be available to a radiographer in a hospital. All external system components are simulated in this environment.

Integration tests are performed on the WS, STM and RTM test environments. These environments are available to developers to perform tests on a more realistic system configuration with varying combinations of simulators and actual components. In these environments, software components that have been developed or modified are patched on the systems and tested to verify that they interact correctly with other existing components.

System tests and clinical/acceptance tests are performed on OTM environments. The OTM comprises a full working system with all components in place. It is only reserved by developers in special test situations that require all components to be present. The OTMs are used, by the software integration team, to perform a daily (updated components) and weekly (entire system) integrator test. The integrator test involves a minimal number of system functions being tested to ensure that the system still operates correctly. This test determines whether or not the updated source code shall be consolidated with the existing archive. Besides the integrator test, the OTMs are used to perform clinical test procedures. These tests are carried out by qualified application specialists and simulate the actual use of an MR system in a hospital environment.

Much of the testing described above is carried out manually. However, there are a number of other specialised automated tests for example, the reconstructor regression tests, and scanning hardware functionality tests.

Some PMS organisational roles specifically defined to handle software testing responsibilities are the test coordinator and the test engineer. The test coordinator is responsible for the software test activities in a particular development project/increment. He/she focuses on planning tests and tracking these plans. He/she also gives input to a segment leader with respect to the required resources, like test engineers and test systems. The test engineer is mainly responsible for the creation of tests, the execution of tests and reporting of the test results.

2.3.2 Software Testing Vision

The Software Architecture Team (SWAT) at PMS, has a vision to make testing more efficient by taking testability into account when developing software. The objectives of this vision are;

1. to reduce on the amount of test cases required to prove quality

2. to reduce on the effort required to execute the test cases and analyse problems discovered to their root cause

This vision is driven by a number of factors including; the need to test multiple configurations of the MR software, the high cost of test systems, the fact that a lot of time is spent in (regression) testing and bug fixing (tracing problems to their root cause), and the need to ensure that the entire archive is constantly functioning as it is supposed to. The SWAT vision places special emphasis on integration testing as a lot of defects are discovered at this level of testing.

The SWAT team believes that, to ease testing, most testing should be performed on the developer PC, there should be automated regression testing and, most of the problems should be analysed in an offline environment.

2.4 Concluding Remarks

The state of software testing at PMS currently creates a need to improve on the testing in terms of automating it, as well as making it more effective in terms of (relatively) minimal effort put in and maximal results achieved.

Chapter 3

Related Work

In this chapter, we present techniques and investigations from other research on the improvement of software testability.

Design for testability is defined as "a strategy to align the development process so that testing is maximally effective" [5]. The primary concern of design for testability is removing the obstacles to controllable input and observable output. We consider certain tactics or measures, at various phases of the software development process, through which we can attain better controllability and observability hence leading to better testability.

We look at the design and implementation phases of the development process. During the design, the testability of the resulting architecture should be considered relative to the test strategy and the architectural pattern used. In other words, for testability to be high, the choice of test strategy should match the architectural pattern to which the strategy is being applied. Testability in detailed designs can be enhanced through adding constraint information in the designs that prevent ambiguity for the implementers. For the implementation level, testability can be enhanced through providing a testing framework and adding some built-in test code along with the application code.

3.1 Design Testability

Software design is documented as requirements specifications, architectural views and detailed design models. These documents are eventually used in the testing of the software [5] since they spell out the expected behaviour and/or characteristics of the software. In this way there is a relationship between the testability of software and the different forms in which it is represented during design.

3.1.1 Requirements Specification

One of the early stages of software design is the requirements engineering phase. According to Sommerville [33], requirements engineering "is concerned with establishing what the system should do, its desired and essential emergent properties and the constraints on the system operation and software development processes". A distinction is made between user requirements and system requirements. User requirements are defined as the more

ambiguous statements regarding what the system should do. The system requirements are expanded user requirements that describe system functionality in a more detailed and concise way. However, they describe the system's external behaviour and constraints but not its implementation. These (system) requirements can be written as structured language specifications using template forms with standard fields. Also, use-case descriptions and sequence diagrams are used for documenting system requirements. The output from requirements engineering is the software requirements specification (SRS).

System requirements are used for further design and implementation of software, however, they are also used for testing even before any implementation exists [22]. To enhance testability, requirements should be stated in clear, unambiguous terms so that they cannot be misinterpreted both in implementation and test design [5]. Graham [22] recommends that testers should be involved in the requirements engineering phase as this can help with the specification of requirements. At this phase, testers should already be able to spot possible gaps in a specification that could affect testing at a later stage.

Interface faults are one of the most common problems in software [29]. For this reason, there should be a special focus on interface specification during requirements engineering. It is recommended that these specifications are explicitly included in the requirements documents. Interface specifications should include: procedural specifications (application programming interfaces, APIs) and data structures. Interfaces should be specified for both the system that is being designed, and existing systems that are supposed to interact with this system.

Requirements eventually result in artefacts in the software implementation. Being able to link the requirements to artefacts (and vice versa) in the phases leading up to and including the implementation is what is known as traceability [5, 21]. Wright [40] takes the view that traceability is closely related to perceived product quality; it is a means to prove that requirements have been understood, complied with, and no unnecessary features or functionality have been added. Traceability implies that test cases already developed at the requirements stage can be exercised on the final implementation.

In development of software, organizations normally choose between starting either with requirements engineering or architecture specification [28]. This results in either production of artificially frozen requirements documents, or constrained architectures that restrict users and handicap developers by resisting inevitable and desirable changes in requirements. However, since most software-development projects address requirements specification and architecture/design issues simultaneously, then it makes sense to evolve the two incrementally and iteratively. In the next section testability is considered from an architectural point of view.

3.1.2 Architecture and Testability

Software architecture of a computing system is the structure of the system, comprised of a collection of software components, the externally visible properties of these components, and a description of the interactions between these components [1, 20]. Architecture can be looked at as a very high-level abstraction of a system. It is influenced by a number of forces: system stakeholders, the architect's background and experience, and the technical

environment surrounding the architecture at the time of its creation. Each of these forces push for certain properties or quality attributes to exist in the software. Testability is an example of a quality attribute that is required in a piece of software.

Considering that quality attributes are not operational, Bass *et al.* [1] propose the use of *quality attribute scenarios* to translate these attributes into operational notions. A quality attribute scenario is a "quality-attribute-specific requirement" that should typically contain six elements: a stimulus, source of the stimulus, the environment, the artefact, expected response, and the response measure. An example of a testability quality attribute scenario is: within a development/testing environment (environment), an integrator (source of stimulus) performs the integrator test (stimulus) on a subsystem (artefact). The integrator is able to automatically control the subsystem and observe its behaviour (expected response) and at the end of the test 70% coverage is achieved (response measure).

Achievement of quality attributes relies on application of fundamental design decisions, referred to as tactics [1]. Figure 3.1 illustrates application of tactics to influence testability. Upon completion of a software increment, if one has applied testability tactics to the software, it is expected that the software faults are more easily discovered. Testability tactics are divided into two categories: providing input/capturing output, and internal monitoring.

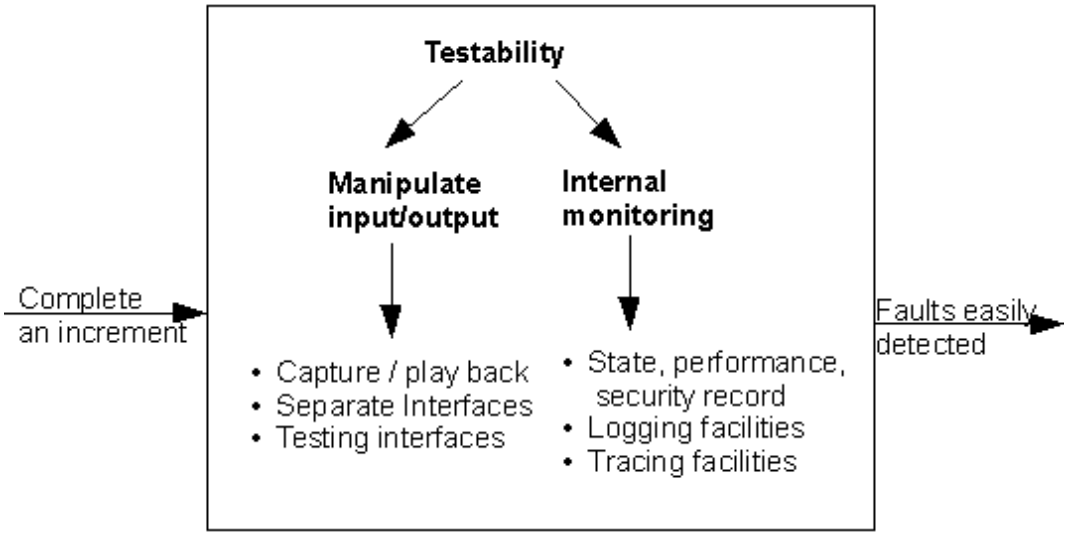


Figure 3.1: Testability Tactics [1]

Capture/Playback techniques involve recording information passing across interfaces in the system during normal operation. Information captured can be stored in a repository and used for two purposes: to compare outputs of a component at a later stage, and to provide inputs to another component. *Separating interfaces* from implementation implies that internal parts of a component can be modified as long as one ensures that the modification adheres to the interface specifications. Using *testing interfaces* allows specialised access to the internals of system components independent from the normal operation. *Built-in mon-*

itors are embedded in system components and can store information like the component state, performance and security.

Architects usually choose from (a collection of) architectural patterns to create a system design [1]. Architectural patterns provide a set of predefined structural organization schemas for software subsystems, along with responsibilities, rules and guidelines for organizing the relationships between the subsystems [10]. The patterns implement a number of design decisions that support certain software qualities. Examples of architectural patterns include: pipes and filters, layers, blackboard and model-view-controller [10, 20].

Testability of an architecture is considered in [13]. The authors propose that architectural decisions should be informed by test strategies, that is, testability of an architecture is a combination of the architectural pattern and the testing strategy chosen. Some strategies are better suited to testing certain architectural patterns as opposed to others, for example, a thread testing strategy is more appropriate to testing pipe and filter architectures, whereas a critical-module strategy is more appropriate to testing an abstract data type (object-oriented) architecture pattern.

3.1.3 Detailed Design

When a more detailed structure of software artefacts is created, it is termed detailed design [10]. The detailed design refines higher level (sub)systems that are defined in the software architecture. The detailed design, as the architecture, specifies components of a subsystem and the interactions between these components. The difference between the two is in the level of granularity considered. For example, classes in object-oriented analysis and design would be used to represent a detailed design, whereas components would be used to represent an architecture.

Similar to the case with architecture, there are situations where design solutions at the detailed design level can be reused with some adaptations. These solutions are known as design patterns [10, 16]. Design patterns offer refinements that transform a collection of classes into a group of logical subsets [2]. However, these refinements may also result in relationships between classes, termed testability anti-patterns, that can lead to problems with testing. Examples of anti-patterns are: self-usage relationships, where a class might indirectly use itself to process a particular task, and class interactions where a class has multiple ways in which it can access methods within another class. These anti patterns are made more complex by the inheritance and polymorphism characteristics of object oriented systems.

Baudry [2] proposes augmenting designs with constraint information such that during implementation, it is clear how to implement relationships between classes. Using the UML as an example, one can take advantage of the UML stereotypes extension mechanism to include constraint information in the design. The stereotypes are used to constrain the relationships between classes to avoid ambiguity. Constraints can be based on whether a relationship between a class is a *uses* and/or *creates* relationship. This shall avoid that the implementation allows multiple paths through which class states can be altered.

3.2 Built-in Test

Testing is mainly targeted at the implementation; therefore, testability of the implementation is a key factor in test effort [9]. Built in test (BIT) capabilities imply inserting test code and test interfaces as part of a component's implementation. This built-in test code, while contained in a component, is distinctly separate from the component's application code and functional interfaces [19, 25, 35, 39]. Component errors can occur within the component, or due to interactions between components [35]. The test interfaces found in BIT-enabled components help with detection of both types of errors either by supporting internal testing of the component, or by enabling external testing facilities to access the component internals. Figure 3.2 is the BIT fishbone from [5] that lays out the different issues to be considered when using BITs. Each of these elements are discussed briefly.

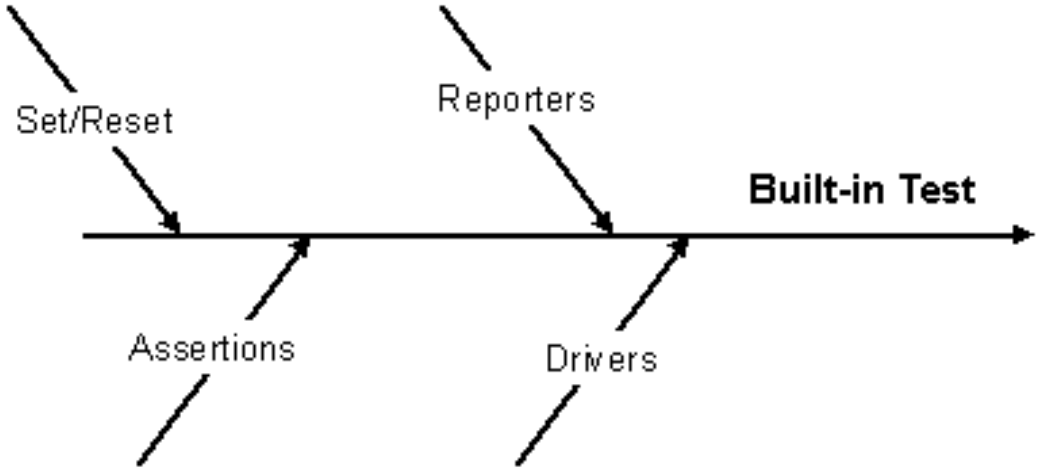


Figure 3.2: BIT Fishbone [5]

(Re)set methods provide the possibility to set a class to a desired state regardless of what its current state is. Since a state is a value set, this implies initialising the class variables to certain values. Gao *et al.* [19] propose an architecture, for a component with BITs, that consists of: a testing interface, and BIT code to facilitate interactions between the component and the external testing tools.

Reporters return information about the state of a component. State information can be returned through various means including: state method calls, logging, tracing and probes. Logging records certain events and actions that occur and stores this information in a log file along with a time stamp. This provides a trail of key activities that can help later on with debugging the system [8]. However, while logging should be sufficient to provide some observability of system behaviour, it should not be too detailed that it becomes cluttered and unmanageable. Too much logging can also consume system resources like hard disk space and memory.

Tracing facilities [19] enables monitoring of the behaviour of a software component during its execution. This monitoring can be in terms of the component's state, performance

or operations. [17] recommends that software components should have a tracking interface and embedded code to interact with external applications that do tracing.

Whereas not all errors that occur within software are observable through outputs, using assertion statements in key parts of the software can help to observe error states that might go undetected [37]. Assertions should, ideally, be placed in areas where it is most likely that the faults in software shall neither corrupt data nor be propagated to the output. According to the studies done by Perry and Evangelist [29], common faults include: inadequate error handling, missing functionality, inadequate post processing and data structure alteration. They find that, in total, 66% of faults identified are attributed to interface problems. Rosenblum [31] specifies two broad assertion classifications to detect these common faults, namely: assertions about the specification of function interfaces, and assertions about the specification of function bodies. Assertions about interface properties can trap issues like mismatched arguments and side effects on global variables whereas assertions about variables within the function body can trap data errors in long-running algorithms [5, 31]. Closely related to these ideas, is the concept of design by contract where preconditions, post conditions and invariants are used to verify expected parameters and states in a system [27].

Test drivers take advantage of the previously mentioned techniques like (re)set methods and reporters. These drivers are responsible for actual execution of tests on a component under test (CUT). They initialise the CUT to a desired state, feed it with appropriate inputs, control and monitor its execution and give a report of test results [24]. In situations where ad-hoc testing strategies are employed, a driver is usually non-existent [5]. In more structured testing strategies, a separate driver may exist for each CUT. Such a driver to CUT ratio, however, may have implications for maintenance efforts of testing code. A more scalable solution would be a generic driver which uses test specifications to automatically generate a test suite.

The techniques discussed above should, ideally, be used in combination to make a piece of software BIT-enabled. Jeon *et al.* [25] propose a pattern for embedding BITs to test the extensible parts of an application framework. The solution proposed is to attach a number of test classes to the (extensible) part of the framework that contains a CUT (see Figure 3.3; classes dedicated to testing are in the gray area). The test classes have an inheritance relationship with the extensible class of the framework. When the application is in test mode and calls are made to methods in the CUT, these calls are forwarded to the CUT through the test classes. Test classes included are: a test driver, test logger, a CUT sensitizer for tracing errors and state changes, CUT monitor for observability and CUT oracle for predicting test results.

In the setup proposed in [12], a CUT provides "hook" interfaces on which to attach the BIT capabilities. Test wrappers are then attached to the CUT through these hooks. The test wrappers' external interface(s) are identical to those of the CUT such that the wrapper is transparent to clients of the CUT. The wrappers, however, monitor any interactions that happen between the CUT and its clients. They detect errors and are able to pinpoint locations where these errors occurred. The downside to this method is that there might be performance penalties due to the wrapper code. Also, it is expected that the wrappers are removed during deployment. This means the system is not released as it was tested; therefore, confidence that it shall behave the same, when deployed, is reduced.

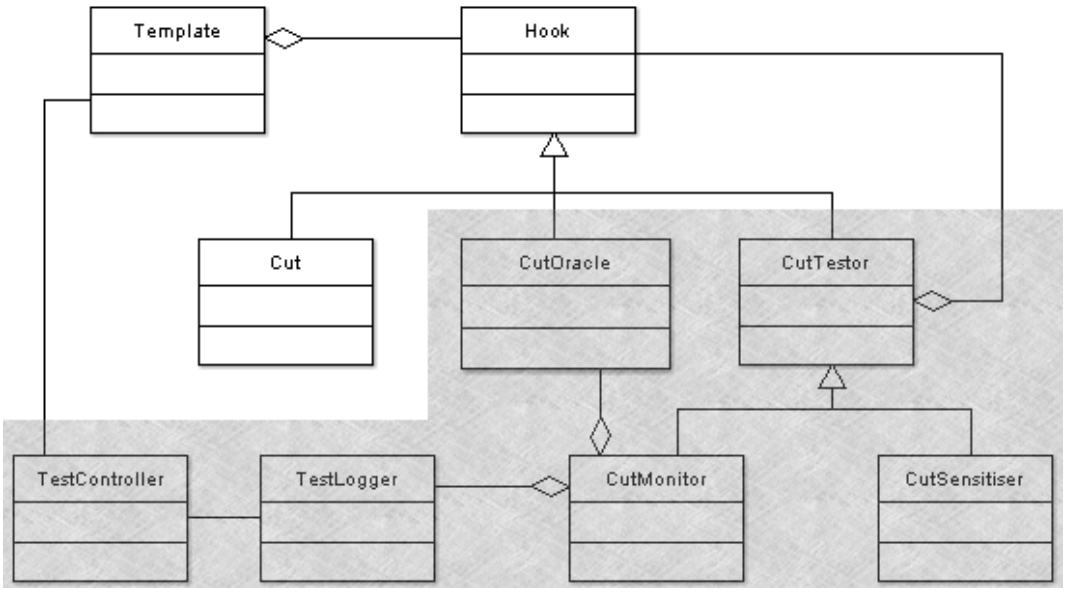


Figure 3.3: Class Structure of BIT-Embedded Framework Extension [25]

Wang *et al.* [39] propose that built in tests, in object oriented systems, should be added as member functions at class level and that on the system level a test subsystem should be built that takes advantage of built in test code at class level. One advantage of this technique is that the test code can be inherited and thus reused by the child classes. It is also possible to reuse the built-in test code on a framework level.

A common thread through the examples mentioned is that there exists some external framework that interacts with the built-in tests. In the next section test frameworks are looked at to give an idea as to how they can be implemented.

3.3 Test Frameworks

Testability of implementations can be enhanced by having a test framework. Test frameworks should provide the infrastructure with which to exercise test cases and collect test results. The test framework is an application separate from the application under test. We investigate how a test framework can be implemented to enhance testability. First, application frameworks are discussed in general and then we focus on frameworks in the software testing domain.

A framework is “a reusable, semi-complete application that can be specialised to produce custom applications” [14]. The applications created from a particular framework fall in an application (sub)domain [32]. Some common examples of frameworks include Microsoft .NET framework and CORBA.

Frameworks have the following properties:

i they are designed and implemented in a modular fashion hence making it easy to understand and maintain software built with the framework,

ii they define generic components that can be (re)used, eliminating the need to recreate common solutions,

iii it is possible to extend the framework's interfaces through hook methods that the framework provides,

iv frameworks have a dispatching mechanism that determines the appropriate application-specific methods to call when an event occurs.

Frameworks can be classified as white-box frameworks, black-box frameworks or a hybrid of the two techniques [14, 32]. These frameworks differ in the way they are extended to create an application. White-box frameworks use object oriented features like inheritance for extensibility. Blackbox frameworks support extensibility by defining interfaces; through these interfaces, extra components can be plugged into the framework to create an application.

The structure of frameworks consists of the frozen part that remains fairly static and the variable part that changes based on the specific application [25, 32]. The variable part of the framework is called a hotspot. Hotspots are typically implemented by hotspot subsystems that consist of: an abstract base/hook class containing hook methods, a number of derived classes each representing different alternatives in functionality, and other additional classes and/or relationships [32].

The framework hotspot subsystems can be classified, based on their properties, as: non-recursive subsystems where services are provided by one object, chain structured recursive subsystems where services are provided by a chain of subclass objects, and tree-structured recursive subsystems where services are provided by a tree of subclass objects [32].

Test frameworks fall under the domain of software testing. Adapting the earlier definition of a framework to the testing domain, a test framework can be considered as *a reusable, semi-complete software testing application that can be specialised to produce custom testing applications*. Variability in the test framework can be either based on the application that one needs to test and/or based on the different (sub)components and their characteristics that are encountered in a system under test (SUT).

Testing of a software system should be automated [5] considering that a lot of effort is required to develop and run test cases. It is, therefore, useful to have a test framework infrastructure that implements reusable and extensible test functionality. When testing software, the following elements should be present: a collection of test cases, test drivers, stubs, test loggers and test oracles [6]. As far as a testing framework, some of these elements are relatively stable and some change based on what (part of the) system is being tested.

As proposed in [12] a test framework should have: an automated generator of built in test wrappers, an automated generator of test drivers based on requirements and test specifications, and automated generation of test cases where applicable and/or possible. The CUTs should provide hooks to which the BIT capabilities are automatically attached by the test framework. For some preliminary testing there should be BIT wrapping, whereas for

normal acceptance testing, no wrapping; however, during normal acceptance testing, there should be access to certain BIT capabilities for the CUT.

Figure 3.4 shows the architecture of the JUnit unit testing framework [3], that is very similar to the *incremental testing framework* as described in [6]. The framework consists of three main classes: TestResult, TestCase and TestSuite. *TestResult* records events from a particular test run, regardless of what kind of test it is. This information includes: number of tests run, which tests passed, and of greater interest, which tests failed. The *TestCase* class represents a test. It provides a typical test case interface that enables setup, execution, evaluation and teardown instructions. The *TestSuite* class enables the aggregation of test cases into one object such that multiple test cases can be run with invocation of the single TestSuite object. While designed for unit testing, this framework can also be used for integration testing as proposed in [6].

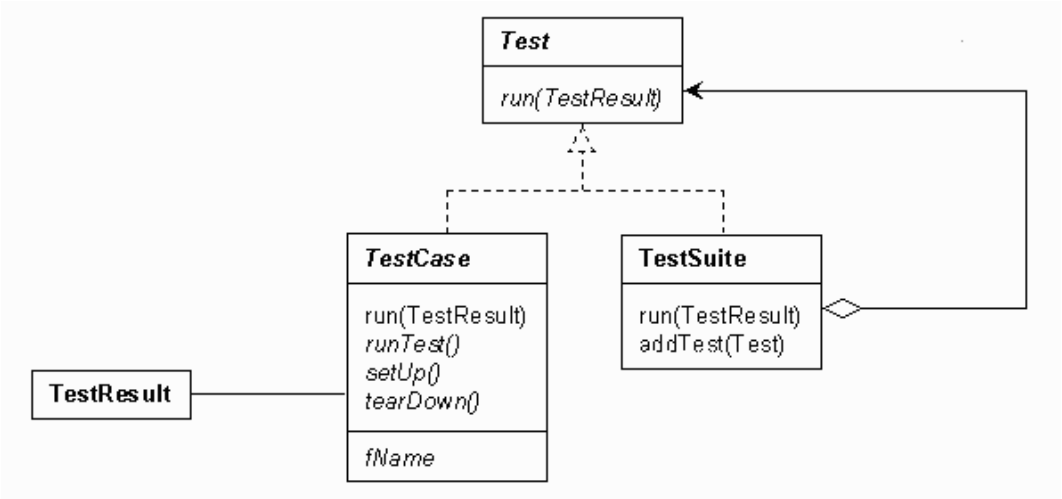


Figure 3.4: JUnit Testing Framework Architecture [3]

3.4 Concluding Remarks

Testability should be considered throughout the development process for it to be effective. At the architecture level, considering the notion of testability and tactics to achieve this already helps when developing relationships between components or choosing what architectural style(s) to use. In this way, architectural considerations are critical to the realisation of testability, however, by itself, the architecture is unable to achieve testability. As one refines the designs with greater detail, testability is usually related to the relationships/dependencies between the smaller components. The implementation determines the final results of all the considerations that start at the architecture level.

Hard-coding test code in the units themselves can cause problems because the units become bulky and complex. On the other hand, a testing framework can have a high programming overhead in terms of effort to develop the framework, and effort to convince

programmers to include the framework library code into their units. An intermediate solution might be to have automatic wrapping of the components being tested [19]. Each component would then have a standardised test interface and a few lines of test code for interaction with a test framework. With this solution, the standardised test interface and test framework interaction code does not require a lot of effort from the developers.

With BITs, concerns arise from the fact that the test code can be used maliciously since it is built-in alongside the functional source code. Measures to prevent this include: having mode switches in the software that can place it in normal or test mode [39] during execution, and having the ability to compile software with or without BITs; this depends on whether one is creating testing or operational versions of the software[5].

Chapter 4

Built-in Test Design

4.1 Introduction

As mentioned in chapter 1, this project work is divided into a literature study and a practical project part. The objective of the practical project is to investigate methods of improving software testability. One of the research questions under consideration is "*How can built-in tests and test frameworks be effectively incorporated in a software system to enhance testability?*". The practical part of the project tries to answer this question through creation of a design scheme for built in testing; this is followed up by applying this design to an existing system.

Built in testing involves inserting test code in the production code of a system. The aim of inserting this test code is to improve on the controllability and observability of the code. In chapter 3, we discussed a number of methods to improve on controllability and observability like: placing assertions in code, use of set/reset methods and having reporters (logging and tracing). In this chapter, we present a design to incorporate built in test (BIT) infrastructure within software (components). This BIT infrastructure is coupled with an interface to allow external testing tools to manipulate and observe a component under test. The design aims to improve on testability of a component, with a focus on integration testing. Figure 4.1 illustrates the overall concept of this design.

Before making the BIT design, however, a number of requirements from such a design are stated. The requirements are documented in section 4.2, followed by a description of the design for the BIT classes (section 4.3) and finally, an approach description for applying these BIT classes to improve on testability of a system (section 4.4). The approach is described in this chapter in a generic way and a specific case of applying it is discussed in chapter 5.

4.2 Requirements for Built in Test Scheme

In order to make a design for a BIT scheme, a number of requirements of such a scheme are defined. These requirements are defined from two points of view; a test interface to the system under test (SUT), and the BIT infrastructure that is embedded in the SUT. For both

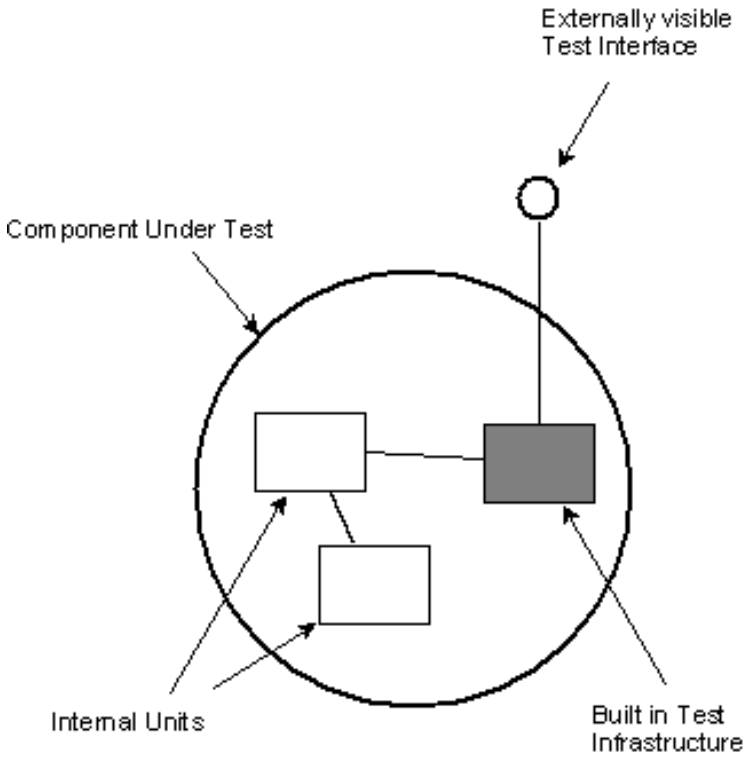


Figure 4.1: Overview of Built in Test Concept

the test interface and the BIT infrastructure, we take into account requirements (functional and non-functional) and some design constraints.

Requirements for the Test Interface

A number of (non-functional)requirements and/or design constraints are defined initially for the test interface:

- Genericity: there should be no tight coupling to any particular implementation technology
- Robustness: the interface should remain relatively unchanged regardless of changes to the BIT infrastructure or the external test tool.
- Independence: the interface should allow for independent evolution of the software under test, the BIT infrastructure, and any external testing tool/framework that shall be used to feed tests to the system.
- Simplicity: The interface should be easy to use through an external test tool. It should require a simple structure of query commands and minimal effort to set up the testing environment. Parameters received by the testing interface should be primarily string

based. This makes it possible to replace/exchange the external testing tool, provided the formatting of data passed to the interface stays consistent.

- Granularity: The size of component to which the interface is attached should be at the level of services / processes[1]. This implies that external access by a test tool shall be granted at the level of processes.

Besides the design constraints, a number of functional requirements are defined for the test interface. The interface is required to provide the following capabilities:

- controllability and observability of the SUT
- query of available commands, existing states and existing object instances.
- allow execution of test cases
- enable select parts of the system to be placed in a certain state

Requirements for the Built in Test Infrastructure

The non-functional requirements and/or design constraints for the BIT infrastructure are defined as follows:

- the BIT infrastructure has to be implemented with object oriented technology
- the infrastructure should be extensible
- it should, as much as possible, be separate from the SUT implementation

Besides these constraints, the BIT scheme has to fulfill the following functional requirements:

- should provide a simple naming scheme for the object instances of the classes under test (CUT) and map these names provided to actual object instances in system.
- should create and register internal test infrastructure.
- should route commands that are fed through the test interface to the correct CUT object instances.

With these requirements and constraints in mind, a design is made for a BIT scheme. The next section gives details of this design.

[1] Here a process is distinguished by the fact that it is capable of running as a standalone executable running its own thread(s)

4.3 Design Scheme of a Built in Test Infrastructure

4.3.1 Built in Test Classes

The design of the BIT infrastructure is such that a number of classes interact with each other to improve on the controllability and observability of the SUT. The design consists of core test classes (IBITest interface, BITester and BITCoordinator) and test support classes (BITCommand, CommandFactory). The core classes are generic and can be reused in different parts of the system, whereas the test support classes are abstract class definitions that have to be extended into concrete classes. The extension of these abstract classes is specific to the SUT. The class diagram in figure 4.2 shows how the classes interact with each other.

A more detailed description of the classes and their functions follows.

BITester Class

This is the core class of the BIT scheme. It is a generic class that is attached to a CUT, when a test that one desires to perform can be invoked from within that CUT. The BITester class is responsible for: executing test commands on the particular CUT that it is attached to, and routing test commands to other BITester classes incase the CUT that it is attached to is not a target for a particular test run. To facilitate the identification of object instances in the system, each BITester object is assigned a unique name when instantiated. This name is used by the external test tool to identify an object instance targeted for testing.

IBITest interface

This interface definition enables access to the SUT by external testing tools. External tools may vary in implementation as long as they use the interface as it is defined. The external tool should perform functions like; invoking test runs, check on the status of the SUT as tests are being run or after tests have been run, checking what commands are available to be executed or what object instances are available for execution of the tests (that is, objects that have a BIT class attached to them).

BITCoordinator

This is a specialised version of the BITester class that implements the IBITest interface. Therefore, it is responsible for providing access to the SUT, enabling external testing tools to query, execute test commands, and extract information about the states and available BITester classes within the SUT.

CommandFactory

A CommandFactory is responsible for instantiating appropriate BITCommand objects based on parameters specified through the BITCoordinator class. The abstract CommandFactory class defines the class structure and is extended by ConcreteFactory classes. The ConcreteFactory classes and the BITCommand classes that they instantiate are specific to the SUT.

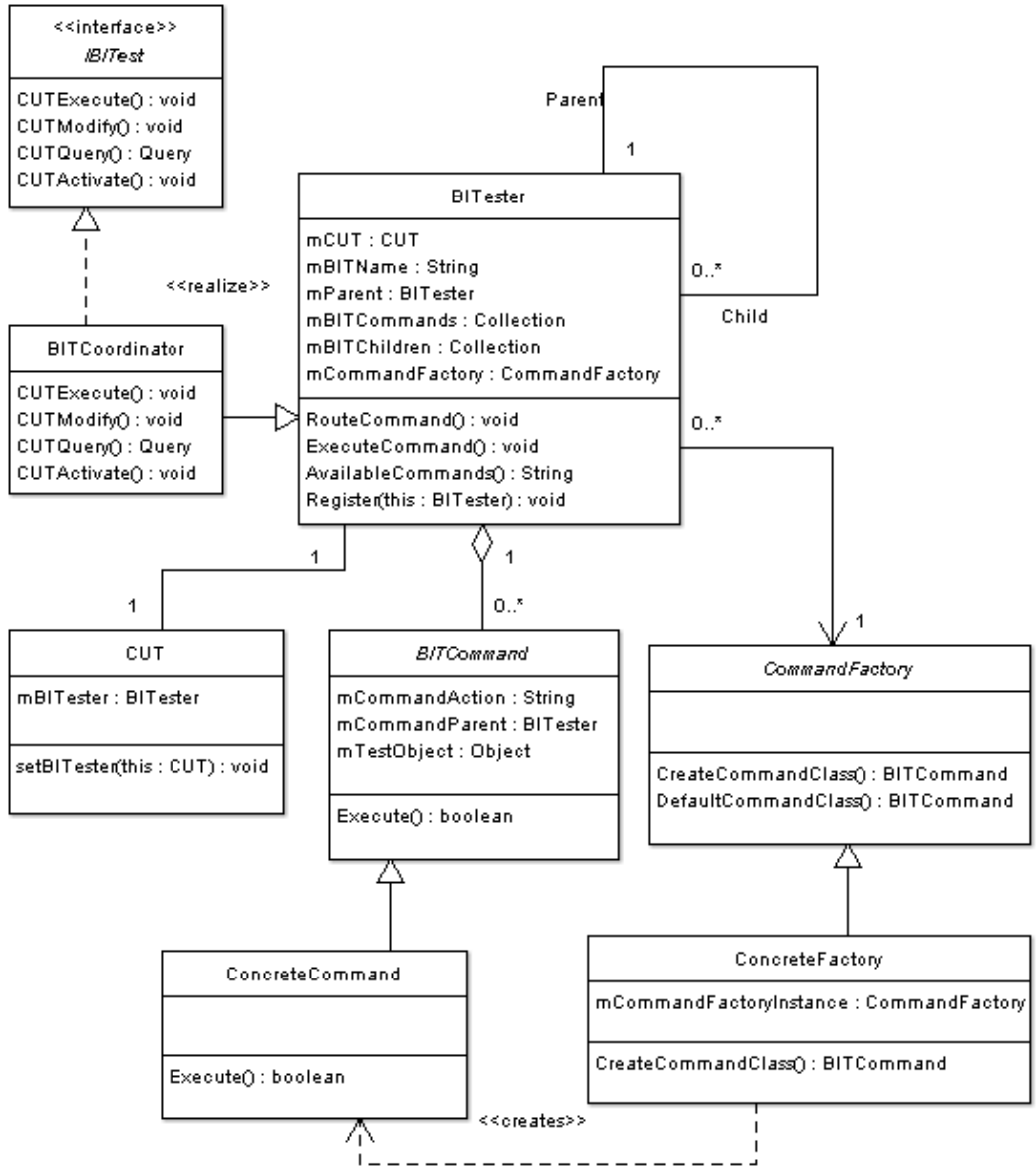


Figure 4.2: Class Diagram of Proposed BIT Scheme

BITCommand

This is an abstract class definition that should be extended into concrete command classes. The concrete versions of this class define and invoke the actions to be executed when a test case is run. These actions are specific to the SUT. The actions are invoked by calling an `Execute()` method, which every concrete command class must implement. The `Execute()` method is called by the BITester class and returns a boolean value: `true` if the method runs successfully and `false` otherwise.

Class Under Test (CUT)

The CUT is the software module targeted for testing. In relation to built in testing, the CUT is responsible for initialising the BIT infrastructure. During initialisation, the CUT provides the BIT classes with a reference to itself. The term CUT does not to imply that the tests are isolated to a class but rather that the tests are initiated from within this class. However, when run, the tests involve a number of other classes. This process is explained in more detail in section 4.4.

TestCaseElements

This class encapsulates the parameters from a test case into a single object. An object of this type is passed to the BITCoordinator through its external interface. The parameters stored in a TestCaseElements object are used to route it to the correct BITester class, to determine which concrete command class is instantiated, and which command action is invoked on the CUT.

4.3.2 Usage Scenarios for the BIT Classes

In this section, two scenarios are described that illustrate typical execution sequences while running tests using the BIT infrastructure.

Setting Up the Built in Test Infrastructure

The BIT infrastructure consists of a BITCoordinator object and a number of BITester objects. The top level class (that contains the `Main()` method) in the class hierarchy of the SUT should have a BITCoordinator class attached to it. This BITCoordinator class maintains a reference to the top level class. Every other class (lower in the class hierarchy) that is targeted for invoking tests should have a BITester class attached to it. These BITester classes also maintain references to the classes to which they are attached.

To setup the BIT infrastructure, a typical flow of events, illustrated in figure 4.3, is as follows. During startup of the software, the top level class instantiates a BITCoordinator class. A number of lower level classes, from which tests shall be invoked, also instantiate BITester classes. Whenever a BITester class is instantiated, such an instance maintains a reference to the class that instantiated it. The BITester class also maintains a reference to a parent BITester class, one level up in the hierarchy. For example, in figure 4.3 the BITCoordinator shall be the parent of the BITester class attached to LowerLevelClass since the BITCoordinator is one level up in the hierarchy. The parent BITester classes in turn maintain a reference to their child classes.

Eventually, when the entire startup sequence of the software has been completed, there should exist a hierarchy of BITester classes attached to each other and mimicking the hierarchy in the SUT as shown in figure 4.4.

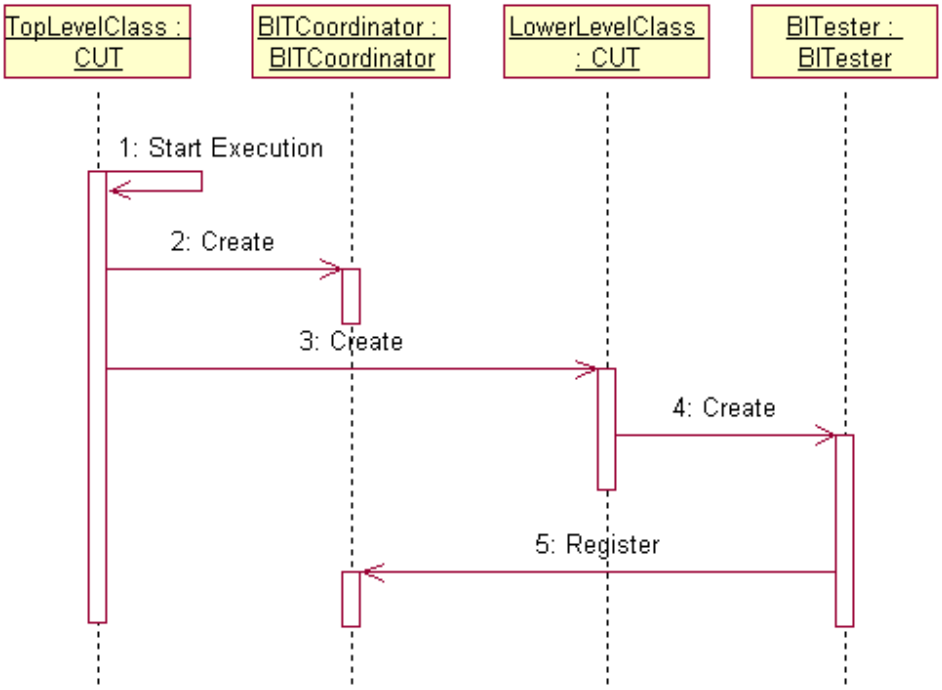


Figure 4.3: Setting up BIT Infrastructure

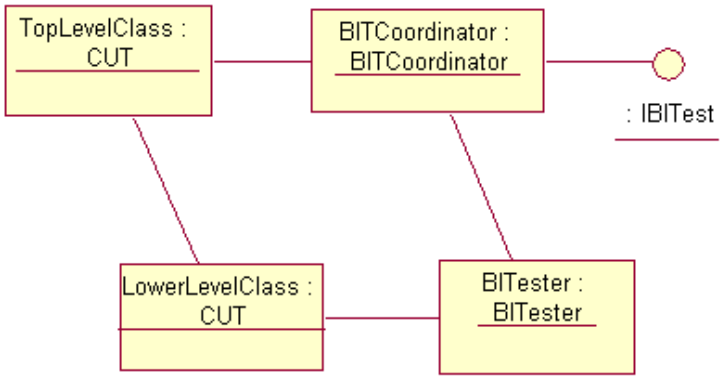


Figure 4.4: Illustration of a Class Hierarchy with BIT Classes attached

Executing a Test Case

The sequence of events for executing a test case assumes that the BIT infrastructure has already been setup as described in section 4.3.2. A test case involves feeding test case parameters to the SUT through its external test interface. These parameters determine how routing to the correct target CUT is done, and what test case is executed. Figure 4.5 shows

a typical sequence of test execution.

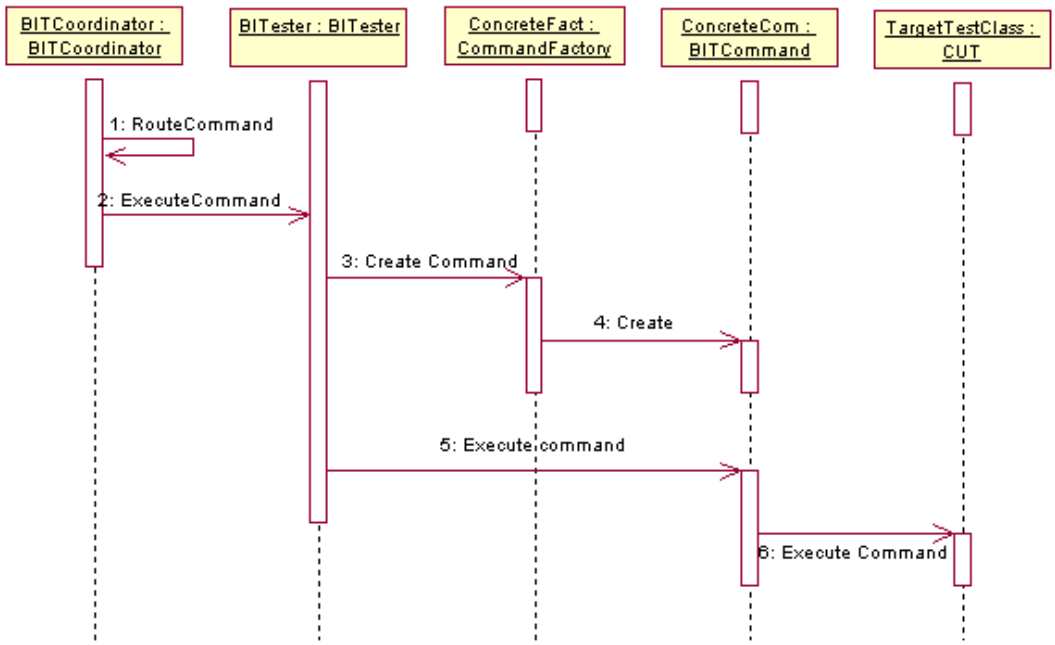


Figure 4.5: Test Execution Sequence

The external testing tool invokes the BITCoordinator class through a test interface and passes to it the parameters required to run a test. If the BITCoordinator class is the target CUT, it executes the test, otherwise it routes the test to the correct BITester class. Similar behaviour is exhibited by the BITester classes if they are invoked by a parent class.

Executing a test case involves querying a concrete command factory for the appropriate command object. The concrete factory instantiates the appropriate command object, which is then executed by invoking the command's `Execute()` method. The execute method of a command invokes actions (initiated in the CUT) required to accomplish the defined test.

4.4 Process Approach to Incorporating Built in Test Infrastructure

Section 4.3 describes the infrastructure and interacting components that are involved in the BIT scheme. A generic description is now given of how this scheme can be employed to enhance the testability of a software component. Based on experimental work from a case study, a more context specific description is given in chapter 5.

Before incorporating BIT infrastructure into a system, it must be decided which parts of the system are targeted for testing. System test specifications are a good starting guide in determining which parts of the system should be targeted. These specifications describe

tests that should be run to verify the system's functionality. Based on these specifications, one can trace where, in the system, the BIT infrastructure will be incorporated.

The approach to incorporating BIT infrastructure can be broken down into two main phases, that is, instrumenting the production code with test code, and defining a set of BITCommands and a concrete command factory.

Instrumenting the production code involves embedding test code in the production code. The purpose of this test code is to instantiate a BITester class. A reference to this BITester class is maintained by the CUT in which the BITester is instantiated. Each BITester class, in turn, maintains a reference to the CUT to which it is attached. The top-level class in the SUT's class hierarchy should instantiate a BITCoordinator class.

Creating a command set involves defining command classes and a concrete factory to instantiate these classes. Command classes derive from the BITCommand abstract class and must implement an `Execute()` method. When a command class is instantiated, it is passed a TestCaseElements object as a parameter. Later, when the command's `Execute()` method is called, it invokes actions on the CUT based on parameters specified in this TestCaseElements object. These actions are defined when implementing the `Execute()` method of the command class.

Besides command classes, a concrete factory needs to be defined. The concrete factory is implemented as a singleton and has a method `CreateCommandClass()` that returns the appropriate command class instance when invoked. The concrete factory follows the factory method design pattern [16] of instantiating different (in this case command) objects depending on which one is needed.

Besides the embedding of the test code and the definition of command classes and factories, there is need for an external test tool that is responsible for feeding test parameters to the SUT. This tool should have access to the external interface of the BITCoordinator class. The tool should wrap the test parameters in a TestCaseElements object and pass this object as a parameter to the BITCoordinator class. The tool should also be able to retrieve and process results from querying the BIT infrastructure.

4.5 Analysis of Built in Test Approach

The approach to using BIT classes to test software components has been described from a generic point of view. During the design phase of the BIT approach, a number of issues are noticed that are now presented here.

A typical (automated) testing scenario involves setup of classes/components under test, execution of test cases and teardown/cleanup of the system after tests are done. These steps are performed by the testing framework on the binaries of an application that is not executing in its "normal" environment. With the BIT approach, testing is performed on the binaries of an application that is executing in its normal environment/manner. This implies that the setup and teardown steps are mostly carried out by code that is already implemented within the system, as part of its normal functionality; as opposed to the setup and teardown steps being performed in the testing framework. Once the code is instrumented, therefore,

the tester can focus on definition of appropriate set of command test classes rather than details of setting and teardown/cleaning up system components.

The BIT approach makes use of text strings to do routing of test commands and identification of classes that are targeted for testing. Therefore, details of the components being tested are abstracted to a level of simple string names and commands to perform certain actions. This makes it intuitive to define and run test cases on the system being tested. This also implies that, while a lot of work needs to be done in terms of the effort to create a comprehensive set of test cases, it might not necessarily be a complicated task.

Considering that programming languages have string processing libraries already included, the BIT approach takes advantage of these already existing facilities, with no extra effort needed to maintain them.

Defining a set of test cases can eventually lead to the problem of test code becoming outdated as the production code evolves. For the BIT approach as described in this chapter, the problem is noticed *sooner*, if there are any changes in production code that could invalidate the test code. This is due to the fact that the test code is compiled within the system as part of the regular code. If there are mismatches between the test and production code, errors shall be raised during compilation to point out these mismatches. This specifically relates to syntactical differences like removing/editing method names, return types and parameters. Errors in program logic, however, are not detected; neither are any code additions that do not have corresponding tests.

The BIT approach can enable automated (regression) testing based on how one implements the external testing tool and defines a command set. The approach gives access to the internal parts of the system and provides a naming scheme for the internal components. This can be used by the external tool to select components and run tests on them.

4.6 Concluding Remarks

This chapter gives the overview of the design of the BIT scheme. We describe the components required for setting up a BIT infrastructure, as well as the typical execution sequences when testing using the BIT technique. We also discuss the approach to incorporating BIT into an existing system. In chapter 5, we consider how this design is applied to a case to improve on testability of a system.

Chapter 5

Case Study: Applying to Philips Medical Systems

The BIT design specified in chapter 4 is applied in a case study at Philips Medical Systems' MR software department. The component chosen for the study is known as the New System User-Interface (NSUI) and is part of the software that is deployed on a host computer (refer to the MR system architecture in figure 2.1). This chapter gives a brief overview of the NSUI component, describes how the BIT design is applied to this component, and presents an evaluation of the BIT experiment.

5.1 Overview of New System UI Component

NSUI is a graphical user-interface framework based on .NET technology. NSUI contains a number of packages whose functions include viewing and post processing of images scanned during patient examinations. As an example, post processing might involve enhancing the quality of an image, or merging a set of scans done at different times so that changes in a body organ can be viewed over time.

The core component of NSUI is an executable named `SystemUI.exe`. In order to perform the functions of viewing and post processing of images, `SystemUI.exe` depends on a number of other components (libraries). Figure 5.1 shows the relationship of NSUI's main component to the other components. Only components that were studied during the experimental work are shown in this figure.

The PictorialIndex component displays a series of examinations for a particular patient. The PackageBase component is responsible for implementing functionality that is shared across a number of viewing and post processing packages in the system, and the PicturePlus component, an example of a package, is responsible for enhancing image quality.

As mentioned in chapter 2, software development work in Philips Medical Systems is divided into modules known as building blocks. The building block where NSUI's main component is implemented is known as UIBackbone. A number of classes in UIBackbone are modified while applying the BIT concept. Figure 5.2 illustrates these classes and how they are related to each other. The central class in UIBackbone is the Workspot class that

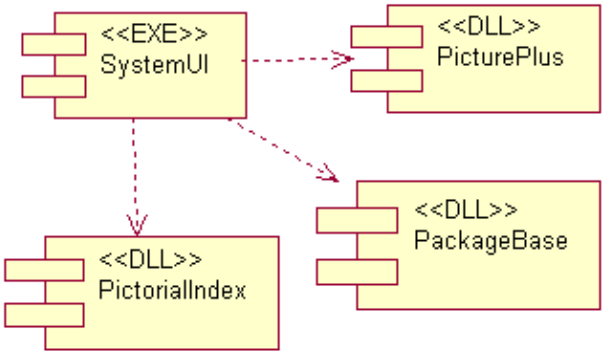


Figure 5.1: NSUI and Interacting Components

starts up the main execution thread of NSUI. This class represents a work environment, that is, a system node where patient data is viewed and/or processed. This Workspot class comprises a number of cases (Case class references), where a case represents a patient whose scan data for a particular examination is being viewed and/or processed. The Case class contains a PackageController class that is responsible for handling package actions like starting up and stopping. The PackageController class also keeps track of all packages (PackageViews) that are started. The PackageView class, which is derived from the View class, controls the UI for image viewing components like PicturePlus.

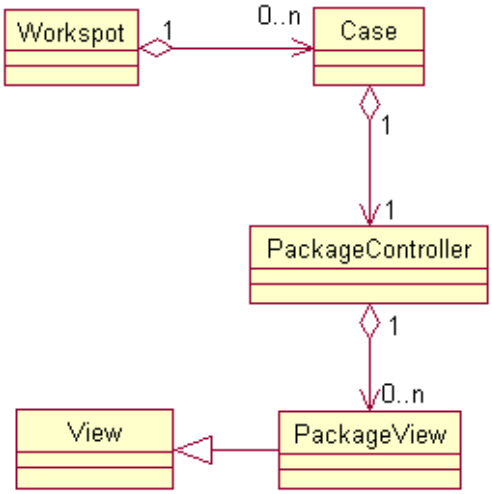


Figure 5.2: UIBackbone Class Structure

These classes are central to performing the basic tasks of the image viewing and post processing functions of the UI. In the next section we see how the BIT concept is applied to these classes.

5.2 Applying Built in Testing to New System UI

5.2.1 Experimental Work

As described in section 4.4, setting up a BIT scheme involves two phases namely: instrumenting the system under test (SUT) with BIT code, and defining a set of command classes and a command factory to instantiate these command classes when needed. Here we describe what is done in these phases for the practical work.

Before setting up the BIT infrastructure, however, we have to determine which kind of tests are run on the NSUI component. The knowledge of which tests are run can guide the phases of instrumentation and definition of commands.

The test specifications for NSUI include, amongst others, the *ability to start and stop packages*. For the components that the NSUI interacts with, the PicturePlus package is among those studied to perform tests on package functionality. The test specifications for this package list *submitting a (processing) job* as part of the functionality to be tested. We now look at how we setup the BIT infrastructure in NSUI guided by these test specifications.

Instrumentation

In this phase, the source code of certain classes is instrumented with statements to instantiate BITester classes and pass to them references to the classes under test. In the case of UIBackbone, the Workspot class is the top level class in the hierarchy. This class is instrumented with code to startup a BITCoordinator class and pass to it a reference to the Workspot class.

From the system documentation, the PackageController of each Case class is responsible for *starting up and stopping packages*. The Case and PackageController classes are, therefore, instrumented with code to attach BITester classes to them. The *submit job* command is initiated by a package's UI. Since the package UIs are derived from the PackageView class, we also instrument the View class (parent class to PackageView) with a BITester class. By instrumenting the View class we can reuse the same BITester code to perform test actions on other packages that derive from the View class.

The effect of instrumenting the system classes with the BIT code is that at initialisation of the system, the BIT infrastructure is also started up. Figure 5.3 shows the object structure of the NSUI component after the BIT infrastructure is initialised during system startup.

Defining a Command Set

During the phase of defining a set of command classes, code is written that executes the functions that one desires to test. These functions are eventually invoked when commands are issued (to the test interface) from the external testing tool. For the case study a command class *PackageCommand* is defined that specifies what actions are taken when *start package, stop package* and *submit job* commands are issued from the external test tool. These commands perform test actions on the NSUI component. Appendix A shows some code snippets from the PackageCommand class to illustrate how test actions are invoked.

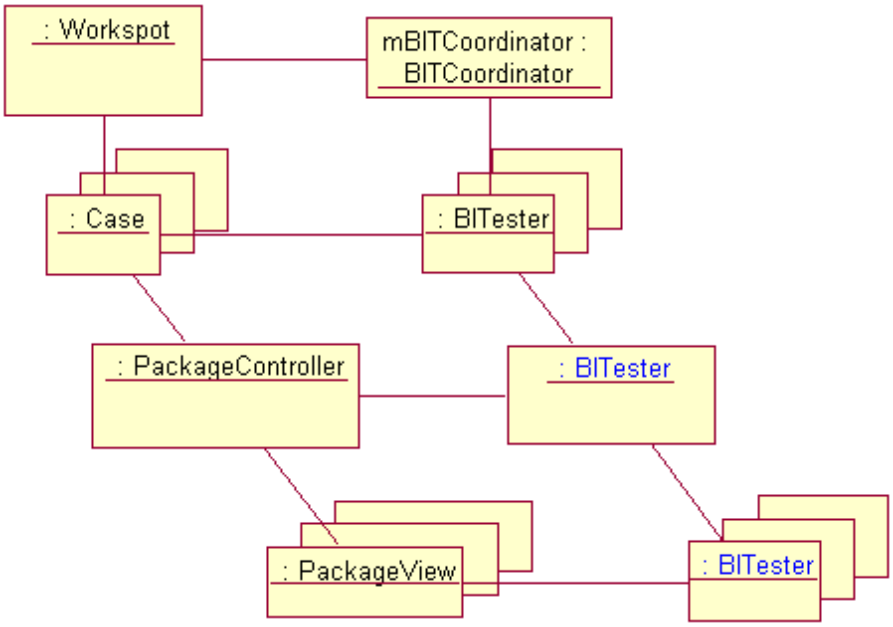


Figure 5.3: Setup of NSUI BIT Objects

Along with the PackageCommand class, a command factory, *SystemUICommandFactory* is defined. This factory is responsible for instantiating command classes when they are needed. For the experiment, the SystemUICommandFactory only instantiates the PackageCommand class.

With the BIT code inserted in the system and the command set defined, a number of tests on NSUI are run to see whether the command set performs the functions expected of it. As the experiments are being run, some issues are identified that are discussed in the next section.

5.2.2 Discussion

During the instrumentation phase the first issue noticed is that not all CUTs, lower down in the class hierarchy, have a reference to their parent class. This reference is needed to create the BITester class hierarchy. To avoid making non-prescribed changes to the setup of the production code, the instrumentation is done in a non-uniform way. In general, the code for instantiating a BITester class is embedded in the CUT that the BITester is attached to; in some specific cases, however, the code is embedded in the CUT's parent. For example, the Case class contains a reference to the Workspot class and so the code to instantiate the BITester class is inserted in the Case class' constructor. On the other hand a PackageController does not have a direct reference to its (parent) Case class. Therefore, the code to instantiate a BITester for a PackageController is instead inserted in its (parent) Case class.

When defining the test commands, one needs to decide the point at which a test command is invoked in the system. For NSUI, certain actions can be initiated from various points in the system. Without in depth knowledge of how the code is implemented it can be a problem to identify the appropriate point at which to invoke a command. For example, to submit a job, one can trigger an event, or simply call a job submit method. Depending on what is being tested it might be the case that it is more appropriate to trigger an event.

Due to technology (C#) restrictions, it is required that the PackageCommand class and the SystemUICommmandFactory are compiled as part of the NSUI executable. This restriction is due to the fact that within the command class, references are made to certain class types that are part of the NSUI executable. These references cannot be resolved if the command class is part of an external class library. The rest of the BIT components, however, are compiled as an external library that can be reused later by other parts of the system. A component diagram for the component setup of built in testing facilities on NSUI is shown in figure 5.4.

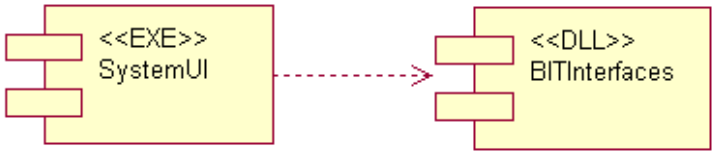


Figure 5.4: A component diagram for BIT in NSUI

The BIT approach as applied here is not automated. For example, while performing trial runs with the BIT classes in place, the system start up, along with certain actions that need to be performed before executing a test, are executed manually. The BIT approach does not provide a mechanism for automatically setting up the system and fulfilling certain pre-conditions before a test is run. Besides setting up the required pre-conditions, there are some tests, like image quality verification, that cannot be fully automated as they require visual inspection of the results by a human operator.

Also while performing trial runs on the system, it is observed that timing issues come into play. For example, two consecutive commands are issued on the system: the first command to start up a PackageView and the second command to do some image processing on the view that has just been started. The second command, however, results in an error because a lot of internal processing happens before the BITester class of that view is instantiated. In order to function correctly, there must be a pause between the start up of the view and the execution of a test on that view.

5.3 Evaluation of Built in Testing in NSUI

To perform an evaluation of the built in testing technique, a presentation, outlining the experiments performed, was made to a number of possible users/beneficiaries of the technique

within Philips Medical Systems. The group of individuals comprised: managers, software designers, software engineers, test coordinators and development tool experts. The technique was presented as explained in chapter 4 and then the specific experiment carried out on the NSUI component was described. A number of reactions to the BIT technique were gathered and are presented in this section. Also included in this section are some evaluations based on observations made during the experimental work. The evaluation is categorised into technical aspects of the BIT setup, and organisational issues that influence applying this technique in the organisation. Also briefly stated are some possible improvements and/or extensions that could be made to the BIT technique in further experiments.

5.3.1 Technical Issues

Intrusiveness of BIT to System

The BIT approach is intrusive to the SUT in a number of ways.

The BIT structure proposed exposes the internals of the SUT through an interface. This might allow for the misuse of the system since test actions can be run on the SUT through this interface.

During the setup of the BIT infrastructure, the BIT objects need to keep references to their parents in the BIT class hierarchy. This reference to the BIT object's parent is normally passed through the parent object of the CUT. There are some cases, however, where a CUT is designed in such a way that it does not have a reference to its parent. Whereas one could add a parent reference in the CUT, this could potentially be intrusive given that the system might have been intentionally designed in a way that such a reference to a parent class is absent. During the experimental work, this exact problem was encountered between the PackageController and the Case classes. The PackageController does not have a direct reference to its parent. In this case it is more appropriate to place the code that initialises the PackageController's BITester in the Case class.

The issue of having multi-threading in the software system also brings into question how intrusive the BIT technique is. Since test interfaces have to be defined that allow access to a running process/thread, the BIT implementation needs to ensure thread safety.

Choice of Query/Test Language

The design proposed is composed of an external testing tool that is able to access the externally visible testing interface. The external testing tool essentially executes commands and queries the system for information about internal components. For this particular experiment, a rudimentary query/test language setup was used. However, in a large scale deployment, it might be the case that such a language shall need to be maintained as well. In this case it is a wise choice to reuse already existing query/scripting languages. Reusing these languages results in benefits like: a shorter learning curve for the testing tools considering that some users of the BIT tools might already have experience in that language, and the query/scripting language is probably already documented as well as maintained by its creators.

System Performance

It is possible that system wide implementation of the BIT facilities shall have an impact on the performance of the system. The MR software requires a tuning mechanism to control the consumption of system resources even without BIT facilities. With BIT facilities implemented, the contention for system resources would require that this tuning also takes into account the BIT facilities. This impact perhaps depends on how the choice is made to attach BIT objects to parts of the system.

Related to performance is the issue of how system and BIT objects are handled when built in testing is in place. One of the tests run in the experiment is starting up a package (view) from a PackageController, resulting in the instantiation of a PackageView object. After these package views are stopped, however, the query function shows that the PackageView objects still exist in the system. Since the BIT objects have a reference to the PackageView objects, they (PackageView objects) are not automatically erased by the garbage collector once the package views are stopped. The BIT approach, therefore, needs to incorporate a way to release the reference to the object instance that it is testing. In fact, the BIT object itself should be destroyed once the CUT object is no longer needed in the system. Without such functionality the system resources could be consumed since objects that are not used anymore are not destroyed.

Required Domain Knowledge

For the exercise of instrumenting the system, it is difficult to choose an exact spot to attach the test code to and to invoke test commands from. The BIT approach, therefore, requires that the persons instrumenting the system and defining the test commands have in depth knowledge in parts of the system that they are responsible for testing. Without this knowledge it takes longer than necessary to identify how and at which point to invoke test commands on the system.

Processing of Test Results

An aspect of the BIT concept that is not implemented fully in the case study experiment, is how to process results from test runs. The current implementation of the `Execute()` method of the BIT command classes returns a boolean value to indicate that a function execution completed or failed, however, no further processing is done with this information. Considering that most of the parameters passed back and forth between the BITester classes are string type values, it is possible to have more information indicating which test failed. This particular aspect is connected to improving the observability of the system under test.

Standardisation of Test Interfaces

There is need for the standardisation of the test interface such that once it is implemented in the various parts of the system, it is not changed (frequently). This standardisation should be considered from system architecture level if it is to stay uniform throughout the system.

Genericity of the Approach

The experiment is carried out on the NSUI component in the system. However, for the approach to be useful, it must be applicable to different system components and different technologies. Based on the design of the BIT infrastructure, it can be implemented using various technologies and in various settings/components. This possibility exists because the BIT design is object oriented and the BIT facilities mostly deal with string type parameters to perform their duties. The parts of the testing infrastructure that are specific to a system component are the command classes and the command factories. These need to be created based on the system (component) being tested.

5.3.2 Organisational Issues

Creation and Maintenance of Test Tooling

Despite the fact that the built in test concept provides the ability to view the internal structure of a running system, a lot of other software tools are needed for test management. For example, test tools are needed to store and compare results from previous test runs, to create and manage the test cases that are run on the system, and to process and present test results in useful formats. Such tools need to be maintained and as a result it might be more desirable that they are available and maintained externally to the organisation, such that the internal organisation is mainly responsible for defining what kind of tests are run on the system and system specific details.

Software Development Process and Procedures

Depending on the software development procedure and work flows in an organisation, there is need to agree on issues like, how the BIT concept shall be fitted into the existing software development processes, which persons shall implement and maintain the BIT infrastructure and at what stages during the development should the organisation take advantage of the BIT infrastructure for testing.

5.3.3 Possible Improvements and Extensions to Built in Test Experiment

The experiment on NSUI sought to investigate whether the built in test method improves on the testability of a large software component. However, while the experiment succeeds in illustrating some of the issues that arise when applying such a technique, there are some aspects of the BIT technique that can be improved to make it more useful. We now discuss some of these possible improvements and/or extensions to the BIT technique.

The BIT experiment needs to be carried out on an even larger scale than NSUI. This would involve defining test interfaces in other components and defining a larger set of command classes. If these BIT classes and interfaces are incorporated into multiple components in the system using different technologies, other problems that are overlooked in the NSUI experiment might be identified.

Besides having information about which classes are available to be tested within the system, it would be handy to retrieve state information of these classes. This way one could

identify the state that a component is in while it executes, and narrow in closer on why/when certain failures occur.

5.4 Concluding Remarks

We have looked at the built in test technique as applied to the NSUI component in the MR software system. In the experimental work the structure of the NSUI and some other building blocks that NSUI is dependent on are studied. These portions of the system are instrumented with built in test code and tests of basic features of NSUI are executed using the BIT code. The BIT technique as applied here, therefore, provides the ability to run high level system tests on a piece of software to determine that the prescribed functionality of the system actually works. A number of short comings in the BIT experiment are also identified and can be improved with further work.

Chapter 6

Conclusions and Future Work

This chapter discusses some of the project's contributions and gives a summary of the entire report. Finally, some ideas for future work will be discussed.

6.1 Contribution

During the literature study phase of this project, a lot of research in the area of software testability was reviewed. A very large percentage of this research, however, does not take into account testing/testability of (components of) large scale software systems; much of it focused on testability of classes and unit testing techniques. The practical part of this project was conducted as an experiment to demonstrate the (in)effectiveness of BITs and test frameworks in improving testability in (components of) a large software system. A design was made for incorporating BIT into a system component. This design was applied to an existing system and a rudimentary testing tool was built to verify the operation of the BIT infrastructure.

A number of issues are observed while making the design for the BIT approach. The approach enables *abstraction of the work of defining and executing test cases* to a higher level. Instead of viewing test cases as source code, they are viewed more as natural language query statements. In some cases, parameters for the tests can even be tweaked from an external testing tool. Besides the ability to execute test cases, the approach also enables a dynamic view of the state of the system. The entire setup creates the possibility for others, who are not developers, to use the test tool; also resulting in improvement in the controllability and observability of a system during testing. The possibility to create and execute commands, however, is dependent on the test command set that is defined for a specific application/system under test.

The BIT approach *facilitates discovery of out-dated test cases* as a system is evolving; though this is at the syntactical level. With some test approaches where the test and production code are kept separate, should there be situations where the production code is changed but not tested for a while, then the test suite could become out dated with time.

The BIT approach promises the *possibility to run automated (regression) tests at system level*. BIT classes give access to the internal parts of a system, combined with a simple

naming scheme for these parts. Besides this, the existence of an external test tool implies that tests can be (re)run in a fixed order, thus improving on the reproducibility of conditions that reveal system errors.

6.2 Conclusion

Testability is concerned with the ease of testing software; it is defined by the terms controllability and observability. Software that has these two properties is said to be testable. Controllability and observability can become operational, in practice, through strategies that are employed in the software development process to make testing easier. These strategies in the development process should be considered in requirements and architecture specification, through to implementation.

At architecture level, controllability and observability are useful terms that help to reason about testability. Since architecture is a more stable asset in a software development process, considering testability at an architectural level can ensure longer term focus on testability. At the design level, architectural patterns need to be matched with a choice of testing strategies, requirements need to be reviewed and refined by testers, and detailed designs need to be analysed for inappropriate relations between classes. For the implementation, having built-in test capabilities coupled with a testing framework would greatly enhance testability.

Built in testing, in software, involves instrumenting the production code of the software with some test code. The purpose of this test code is to enable better controllability and observability of the software as it is being tested. Techniques for built in testing include: using assertions, having set and reset methods, and having logging facilities in the software.

Another approach to built in testing, is to have a built in test infrastructure responsible for exercising tests on a piece of software. This built in test infrastructure should provide facilities like: allow external test tools access to the system internals, execute test commands on the system, query the system to discover its current state, and modify the states of certain parts of the system. In combination with an external test tool this technique can result in a full test suite for automated testing at system level. The BIT infrastructure should be designed, carefully, to fit within the technology of the system in which it is used, and should not affect the performance and behaviour of the system.

6.3 Future work

The work done as part of this project focused more on improving on the controllability of a software system with some minimal addressing of observability features. Extension to this work would involve improving on the observability features of the BIT scheme, thus allowing better observation of the software under test.

Besides this, the experiments were performed on a single process of an entire system. Considering that a large system runs multiple processes and multiple threads, it would be interesting to discover what issues arise when the BIT concept is applied across multiple processes; in other words, a larger scale trial of the BIT concept.

During the practical work, it was observed that the same software archive was reused for different configurations of the software. It is interesting to test whether the BIT technique can improve on testability in situations where software developed needs to be tested in different configurations.

Bibliography

[1] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2nd edition, April 2003.

[2] Benoit Baudry, Yves Le Traon, Gerson Sunyé, and Jean-Marc Jézéquel. Measuring and improving design patterns testability. In *Proceedings of the 9th International Symposium on Software Metrics (METRICS '03)*, pages 50–59. IEEE Computer Society, 2003.

[3] K. Beck and E. Gamma. Junit: A cook's tour, August 1999. JavaReport.

[4] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[5] Robert V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101, 1994.

[6] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] Rex Black. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. John Wiley & Sons, Inc., 2007.

[8] Pettichord Bret. Design for testability. In *Proceedings of Pacific Northwest Software Quality Conference (PNSQC)*, pages 243 – 270. PNSQC/Pacific Agenda, October 2002.

[9] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *Journal of Software Systems*, 79(9):1219–1232, 2006.

[10] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996.

[11] Gupta S. C. and Sinha M. K. Impact of software testability considerations on software development life cycle. In *Proceedings of the First International Conference on Software Testing, Reliability and Quality Assurance*, pages 105–110. IEEE, December 1994.

[12] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Journal of Software Testing, Verification and Reliability*, 11(2):97–111, 2001.

[13] Nancy S. Eickelmann and Debra J. Richardson. What makes one software architecture more testable than another? In *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*, pages 65–67. ACM Press, 1996.

[14] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

[15] R. S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[17] Jerry Gao, Zhu E.Y, Shim S, and Lee Chang. Monitoring software components and component-based software. In *Proceedings of the 24th Annual International Computer Software and Applications Conference (COMPSAC '00)*, pages 403–412. IEEE Computer Society Press, 2000.

[18] Jerry Gao and Ming-Chih Shih. A component testability model for verification and measurement. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC '05)*, pages 211–218. IEEE Computer Society, 2005.

[19] Jerry Z. Gao, Kamal K. Gupta, Shalini Gupta, and Simon S. Y. Shim. On building testable software components. In *ICCBSS '02: Proceedings of the First International Conference on COTS-Based Software Systems*, pages 108–121. Springer-Verlag, 2002.

[20] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[21] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of International Conference on Requirements Engineering (ICRE)*, pages 94–101. IEEE, 1994.

[22] Dorothy Graham. Requirements and testing: Seven missing-link myths. *IEEE Software*, 19(5):15–17, 2002.

[23] Dick Hamlet. What is software reliability? In *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS '94) 'Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security'*, pages 169–170, July 1994.

[24] IEEE. IEEE glossary of software engineering terminology, ieee standard 610.12. Technical report, IEEE, 1990.

[25] Taewoong Jeon, Hyon Woo Seung, and Sungyoung Lee. Embedding built-in tests in hot spots of an object-oriented framework. *ACM SIGPLAN Notices*, 37(8):25–34, 2002.

[26] Ronny Kolb and Dirk Muthig. Making testing product lines more efficient by improving the testability of product line architectures. In *Proceedings of the ISSTA 2006 workshop on Role of Software Architecture for Testing and Analysis (ROSATEA '06)*, pages 22–27. ACM Press, 2006.

[27] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[28] Bashar Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(2):115–117, 2001.

[29] D. Perry and W. Evangelist. An empirical study of software interface faults. In *International Symposium on New Directions in Computing*, pages 32–38. IEEE Computer Society, 1985.

[30] C. V. Ramamoorthy and S. F. Ho. Testing large software with automated software evaluation systems. In *Proceedings of the international conference on Reliable software*, pages 382–394. ACM Press, 1975.

[31] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[32] Hans Albrecht Schmid. Systematic framework design by generalization. *Commun. ACM*, 40(10):48–51, 1997.

[33] Ian Sommerville. *Software Engineering*. Addison Wesley, 7th edition, May 2004.

[34] Yves Le Traon and Chantal Robach. Testability measurements for data flow designs. In *Proceedings of the 4th International Symposium on Software Metrics (METRICS '97)*, pages 91–98. IEEE Computer Society, November 1997.

[35] J. Vincent. *Built-In-Test Vade Mecum Part I, A Common BIT Architecture*. Component+, 2002.

[36] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Software Eng.*, 18(8):717–727, 1992.

[37] Jeffrey M. Voas. Quality time: How assertions can increase test effectiveness. *IEEE Software*, 14(2):118–122, 1997.

[38] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[39] Yingwu Wang, Dilip Patel, Graham King, Ian Court, Geoff Staples, Maraget Ross, and Mohamad Fayad. On built-in test reuse in object-oriented framework design. *ACM Comput. Surv.*, pages 7–12, 2000.

[40] S. Wright. Requirements traceability - what? why? and how? In *Proc. IEE Colloquium on "Tools and Techniques for Maintaining Traceability During Design"*, pages 1–2. IEEE, 1991.

[41] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. On how developers test open source software systems. Technical Report TUD-SERG-2007-012, Delft University of Technology, Software Engineering Research Group, 2007.

Appendix A

Code Snippets

A.1 PackageCommand class

Code snippets from the PackageCommand class.

```
public class PackageCommand : BITCommand
{
'
'
public override bool Execute()
{
switch(mCommandAction){
case "Start":
'
'
case "SubmitJob":
'
'
case "StopAll":
mPackageController = (PackageController) mTestObject;
mPackageController.StopAllPackages();
return true;

default:
return false;
}
}
}
```