2092

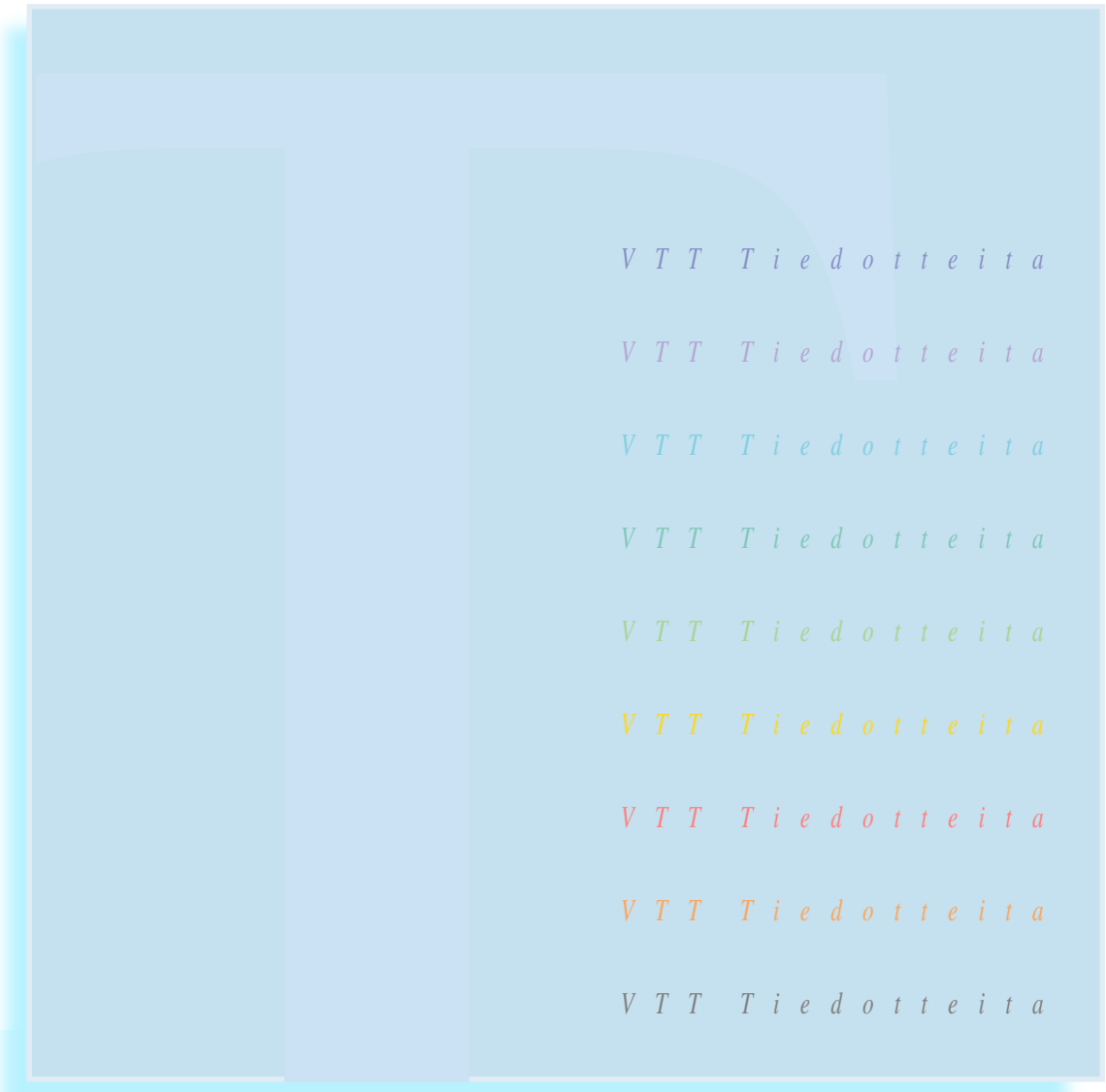
VTT RESEARCH NOTES

VTT TIEDOTTEITA

Matti Kärki

Testing of object-oriented software

Utilisation of the UML in testing



TECHNICAL RESEARCH CENTRE OF FINLAND

ESPOO 2001

VTT TIEDOTTEITA – MEDDELANDEN – RESEARCH NOTES 2092

Testing of object-oriented software

Utilisation of the UML in testing

Matti Kärki
VTT Electronics

TECHNICAL RESEARCH CENTRE OF FINLAND
ESPOO 2001

ISBN 951–38–5816–2 (soft back ed.)
ISSN 1235–0605 (soft back ed.)

ISBN 951–38–5817–0 (URL:http://www.inf.vtt.fi/pdf)
ISSN 1455–0865 (URL:http://www.inf.vtt.fi/pdf)

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Kerttu Tirronen

Otamedia Oy, Espoo 2001

Kärki, Matti. Testing of object-oriented software. Utilisation of the UML in testing. Espoo 2001, Technical Research Centre of Finland, VTT Tiedotteita – Meddelanden – Research Notes 2092. 69 p. + app. 6 p.

Keywords Unifield Modelling Language, software development, software test processes, software test automation

Abstract

The modern software development requires more efficient production methods than ever before. It has been recognised that benefits can be obtained in software development by using object-orientation. Testing, however, has gained less attention, although it is still an important task in the software development to achieve such goals as finding errors and quality.

The goal of this paper is to study how object-orientation affects testing as well as how the testing techniques that are adapted for object-orientation can be used for test design purposes. Utilisation of the Unified Modelling Language (UML) in testing is introduced, and some practical solutions to avoid the obstacles of the testing of object-oriented software are addressed as well. Moreover, these solutions are combined and a test automation system (test driver implementation), which makes it easier to test the object-oriented software, is presented.

Finally, the testing techniques that are studied, are applied to a demonstration system, which is designed and implemented by using a CASE tool called Rhapsody. As Rhapsody provides its own impact to testing and test design, it is shown how the various UML diagrams are used for test design purposes in the context of Rhapsody.

Although object-orientation provides benefits for software development, it can be argued that the testing of object-oriented systems is occasionally more difficult compared to the testing of traditional systems. However, by planning tests carefully and taking the special needs of the testing of object-oriented software into account, these obstacles can partially be avoided. Furthermore, since the UML provides a notation to express software designs, and as object-orientation emphasises functional testing, the UML gives information for test design that should not be overlooked.

Preface

This work was carried out at VTT Electronics during the ASCENT-project in order to define suitable testing techniques for object-oriented software. The project was funded by Kaski Tech, the National Technology Agency (Tekes), Nokia Mobile Phones, Nokia Networks, and VTT Electronics. I appreciate them for arranging the opportunity to write the master thesis (tech) for the Department of Electrical Engineering, University of Oulu.

I wish to express my gratitude to my supervisor at the University of Oulu, Professor Tino Pyssysalo, for his comments, feedback and guidance. I am also grateful to the second supervisor of the thesis, Professor Juha Röning for his comments.

I would like to thank people at VTT as well. I thank Mr. Jukka Korhonen, Mr. Hannu Honka and other co-workers for their support and useful discussions during the project.

I express special thanks to my parents who encouraged me to educate myself and provided the possibility to do that. Finally, my additional thanks go to Auli for her support and understanding during the struggle of writing the thesis.

Oulu, Finland, February 2001

Matti Kärki

Contents

List of symbols

Bug	An error or a fault.
CUT	Class Under Test.
CUT	Component Under Test.
Component	Any software aggregate that has visibility in development environment, r example a method, a class, an object, a subsystem, an executable.
Error	A mistake made by a human.
Failure	A failure occurs when fault is executed.
Fault	A fault is the result from an error. A software fault is missing or incorrect code.
IUT	Implementation Under Test.
LSP	Liskov Substitution Principle.
MUT	Method Under Test.
OUT	Object Under Test.
SUT	System Under Test.
Test case	Collection of test inputs, execution conditions, and expected outcomes.
Test suite	A set of related test cases.
Traditional software	Software that is written in an imperative language (e.g. C), designed by a functional decomposition, developed in a waterfall life cycle and separated into three levels of testing.
UML	Unified Modelling Language.

1. Introduction

It can be asserted that object-orientation was created in the late 1960's when the Simula were introduced [1, p. 6], but only during the last decade object-orientation has gained popularity. Many reasons can be enumerated for this. Nowadays, the software systems are getting larger and more and more complex all the time, and managing these complicated systems has become difficult. In addition, software has to be produced quicker and more efficiently, which consequently emphasises software re-use. It is recognised that object-orientation provides means for making software re-usable, and moreover, it offers a solution for the problems of managing large and complex systems, as it gives an ability to break down systems into smaller ones. Furthermore, standardisations of C++ and the Unified Modelling Language (UML), and especially Java programming language, have all led to an increasing acceptance of object-orientation.

Despite all this favour that object-orientation has gained in recent years, testing has become a neglected area. There are numerous books about developing and implementing the object-oriented software, but only a few books concentrate on testing. Although many software testing strategies and models have been proposed during the last decades, they do not address the special needs of the testing of the object-oriented software, since most of them focus on traditional software. Applying the traditional software testing techniques and strategies to object-oriented development may be inadequate, since the traditional software diverges greatly from the object-oriented software. Apparently, the features that give object-orientation its strength, such as encapsulation, inheritance, polymorphism, and dynamic binding, have an effect on testing. Besides the differences, there are similarities that can be easily recognised. For instance, state-based testing may be useful for object-oriented software, since the objects preserve state information. In this paper, it is studied how object-orientation affects testing, and some practical solutions to avoid the obstacles of the testing of object-oriented software are presented as well. Based on these solutions, a test automation system (test driver implementation) that takes the special needs of the testing of object-oriented software into account, is presented.

In order to succeed, testing has to be designed. Traditionally, software design documents as well as requirements and code are used for test design purposes. The Unified Modelling Language has become the leading modelling language for object-oriented analysis and design. Since the UML provides a notation to express software designs, it gives a source of information for designing tests. Furthermore, the standardisation of the UML has given a necessary foundation for software development and design tools. Such a tool, called Rhapsody, is introduced in this paper. Although Rhapsody is compliant with the UML, it provides its own impact test design, as it puts

some constraints on the use of the UML. In addition, code generation from the UML diagrams has – obviously – an effect on testing.

The objective of this paper is to study how the testing techniques that are adapted for object-orientation can be used for test design purposes. Furthermore, utilisation of the Unified Modelling Language in testing is introduced. Finally, the testing techniques, that are studied, are applied to a demonstration system, which is designed and implemented by using Rhapsody. As Rhapsody provides its own impact to testing and test design, it is shown how the various UML diagrams are used for test design purposes in the context of Rhapsody.

Although the UML supports various diagrams, the focus is on use case, sequence, statechart, and class diagrams, since these are the diagrams supported by Rhapsody. Moreover, C++ is used as an example language in the paper. However, some of the examples are language independent and consequently can be adapted for other programming languages such as Java.

2. Software testing

Traditional software is a well-known area together with the testing of it. Understanding the testing of traditional systems provides a good basis for testing the object-oriented systems, as clearly, there are similarities. Although object-orientation does not alter the purpose of testing, the testing is different from the technical perspective because object-orientation provides new features and emphasises other features. Consequently, some testing strategies need adaptation and some lose their effectiveness. These differences are discussed in the second part of this chapter.

Traditionally, software design outputs (as well as code and requirements) are used for driving the testing, as they give much of the information to test the system. For object-orientation, the Unified Modelling Language (UML) has become de facto standard for modelling the software systems. Since these diagrams provide inputs to test design, the utilisation of UML models in testing is introduced at the end of this chapter.

2.1 Introduction to software testing

One of the goals of a software process is to produce high-quality software products. To achieve this goal, different types of quality assurance methods are needed. Software quality assurance activities encompass reviews, inspections, walkthroughs, and testing [2]. Hence, software is tested, since judgement about quality and acceptability has to be made [3, p. 3]. It can be argued that testing is a complex, challenging, and time-consuming part of a software project. There is no short cut in testing, even though automation can to some extend be applied to testing, and furthermore, testing is necessary because mistakes that lead to system failures are made, and these failures must be discovered [3, p. 3].

With these goals defined for testing, the definitions of testing can consequently be provided. The first definition of testing, which emphasises quality, is:

Testing is the measurement of software quality [4, p. 20].

And since quality means the degree to which a system meets customer needs and expectations, another definition of testing can be given:

Testing is establishing confidence that a program does what it is supposed to do [4, p. 20].

The third definition of testing limits the scope of testing and makes the finding of errors the goal of testing. It emphasises the execution of code and limits the testing to running an implementation. Testing of documents such as specifications is precluded.

Testing is the process of executing a program or system with the intent of finding errors [5, p. 5].

In order to achieve the goals of testing, tests have to be planned and designed. To accomplish this task, test design strategies are needed. These methods have traditionally been black and white box strategies and their combination, a grey box strategy. In addition to different types of test design strategies, we have to distinguish levels of testing allowing us to define different objectives for each level. Thus, testing is different at different levels; different types of errors are searched for at each level.

2.1.1 Levels of testing

Unit testing, integration testing and system testing are the levels of testing on a typical software system [6, p. 20]. These levels of testing can be found in the waterfall model of the software development life cycle: the levels of testing correspond to the levels of design. Although this model has it drawbacks, it is useful to distinguish levels so we can define different objectives for each level. The waterfall model with testing levels is shown in Figure 1 [3, p. 159].

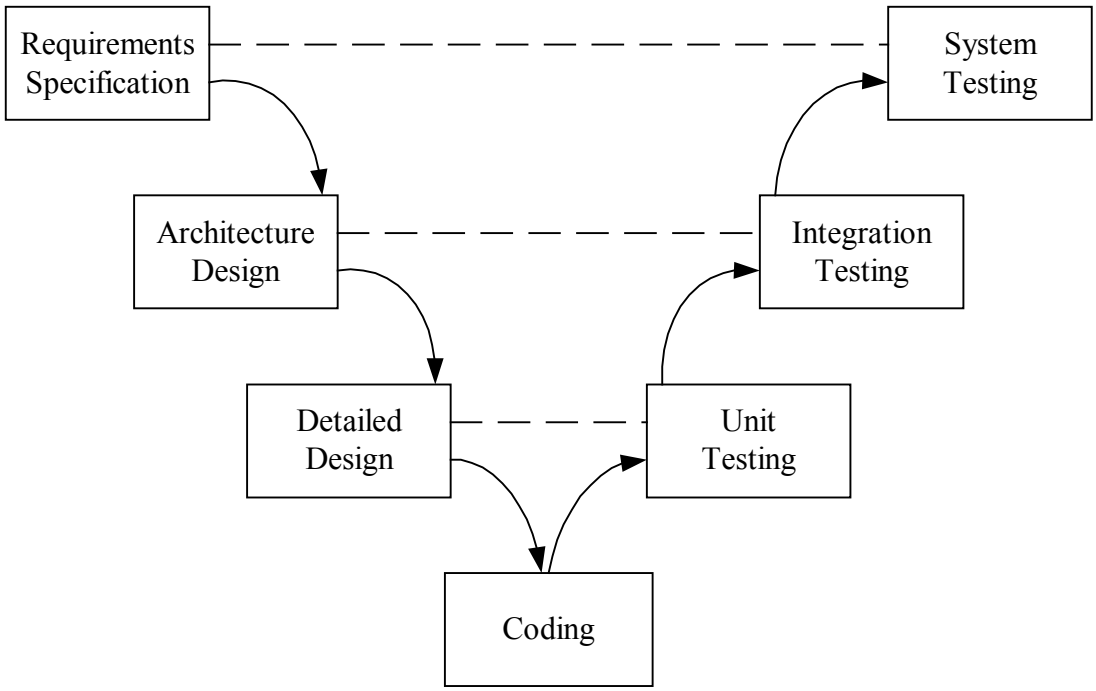


Figure 1. The V-model.

As it can be seen from the Figure 1 there is correspondence between design and testing levels. Requirements specification, architecture design, and detailed design correspond with system testing, integration testing and unit testing respectively. Test planning is done on the correspondent level of design, and the results of testing are verified by comparing them with equivalent design documents [7]. Although the primary goal of testing is to find the faults in the implementation, faults in the requirements and design can be found as well [8, p. 324].

Unit testing is the testing of the smallest testable piece of software, a module or a few modules that are so tightly coupled that testing them individually is impractical. The purpose of unit testing is to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match with the intended design structure [6, p. 21]. Unit testing is usually done by the unit's originator.

To test a module, it is often necessary to simulate the outside world of the module. We need to simulate those modules, whose services are used by the module to be tested, using so called stub modules. These stubs use the subordinate module's interface, may do some data manipulation and return. In addition to stub modules, we have to use test drivers to provide test inputs, control and monitor the execution of the tests, and report the results. Figure 2 depicts the unit test environment [2, p. 514].

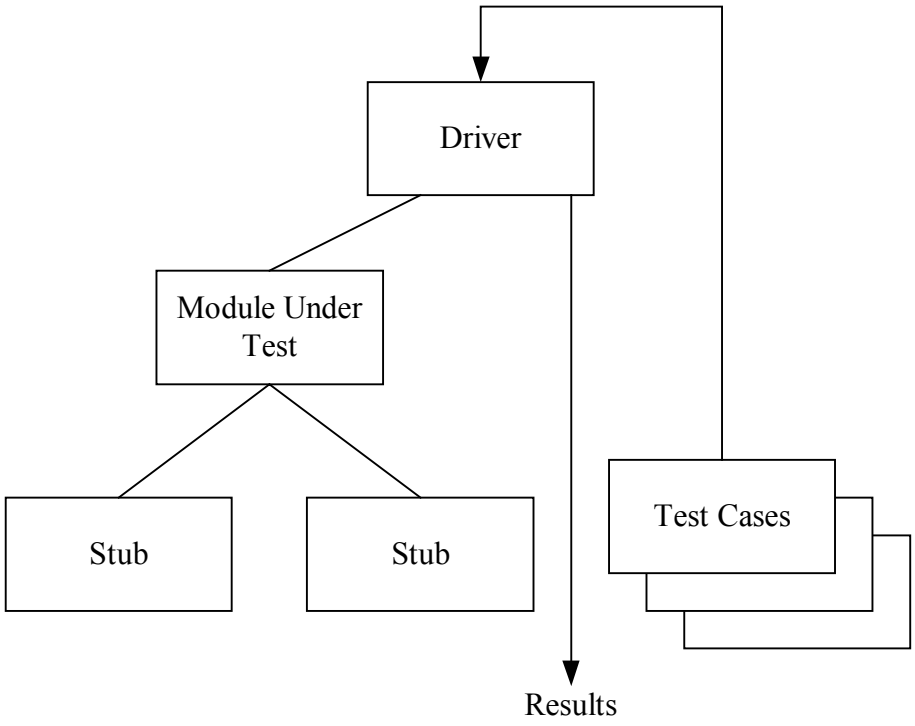


Figure 2. Module test environment.

Integration is a process in which components are aggregated to create larger components. The main focus of **integration testing** is to test the interfaces between components. We want to reveal the components' faults that cause inter-component failures and show that even though the components were individually satisfactory, the combination is incorrect or inconsistent [6, p. 21]. Integration testing is not the same as testing integrated components, which is just a higher level of component testing.

System testing concerns issues and behaviours that can only be exposed by testing the entire system. It is done to explore system behaviours, which cannot be carried out by means of unit or integration testing. It often includes set-up, security, performance and recovery testing and is done by an independent testing personnel. [6, p. 22, 9, p. 5]

2.1.2 Test case identification

There are two approaches to identify test cases: black box testing and white box testing. Both of these test design strategies have several distinct test case identification methods, called testing techniques [3, p. 7]. In addition to white box and black box testing, there is a third testing approach: hybrid testing. Hybrid testing, commonly known as grey box testing, combines white and black box testing.

White box testing techniques are derived from the implementation and the structure of the tested component, i.e. the source code. The source code is tested as extensively as possible, e.g. by executing every statement at least once or executing every branch once [9, p. 8]. White box testing is also called structural, clear box, glass box or implementation-based testing.

In contrast to white box testing, **black box testing** techniques are based on requirements, and therefore test cases are derived from the requirements. The purpose of black box testing is to test the component to ensure if it meets its functional and non-functional (e.g. performance) requirements. Synonyms for black box testing are: functional, behavioural and responsibility-based testing.

The problem of these test design strategies is that neither of them can reveal all the errors. With black box testing, the problem arises when trying to achieve a coverage goal. It may be impossible to achieve sufficient coverage by looking at the component as a black box, and the implemented features that have not been specified may therefore never be revealed [3, p. 9].

White box approach eliminates the testing problem induced by black box approach, but the problem with the white box test is that it only shows that the code does what it does, and all the specified features that have not been implemented will never be recognised [3, p. 9]. Thus, the best combination of black box and white box techniques would be to start with the black box tests and use an appropriate coverage analysis tool to get feedback about the test. Subsequently, if the coverage is insufficient, more tests should be developed [10].

2.2 Testing of object-oriented systems

When the testing of traditional software and the testing of object-oriented software are compared, some differences are recognised. The main differences are techniques in software testing and difference of emphasis.

White box testing, which is based on internal knowledge about the code being tested, is the one that makes the testing different. Nevertheless, the underlying principle of white box testing to use implementation as a base of testing is the same for both testing. Black box testing, in which test cases are derived from requirements, is very similar to object-orientation systems compared to any traditional systems. For instance, numbers of black box techniques can be applied to the testing of object-oriented systems. [10]

There is also a difference of emphasis in traditional languages and object-oriented languages. The most detailed level of object-oriented systems tends to be very simple and small, which makes testing of these low-level components easier compared to traditional equivalent of methods. However, object-oriented testing is a challenge at the integration level. The lack of complexity at one level does not reduce the overall complexity of the system. In effect, the complexity of methods is pushed to interfaces between classes: there are many objects that interact in many ways. Thus, the interface errors are more likely in object-oriented languages than in the traditional software.

The object-orientation provides new features like inheritance, polymorphism and its enabling mechanism dynamic binding. Inheritance is a mechanism that allows one class (the subclass) to incorporate the declarations of all or part of another class (the superclass) [8, p. 1092]. Polymorphism is the ability to bind a reference to more than one object [8, p. 77]. Clearly, these essential features of object-orientation pose a challenge to testing. In addition, other features of object-orientation, such as encapsulation which enables information hiding, and complicated interaction between message sequence and state, lead to some problems in testing. A detailed discussion of the effect of inheritance, encapsulation and polymorphism on testing is provided in later sections.

2.2.1 Levels of testing in object-oriented software

Just like in traditional environments, there is a hierarchy of testing levels in the object-oriented world as well. In addition to the hierarchy of testing levels, a rough correspondence can be found between the traditional testing levels and the testing levels of object-oriented systems.

Testing levels in the object-oriented world can be categorised into five levels. These levels are class (small cluster), inheritance, integration/interaction, cluster, and system testing. Individual classes and small clusters are the focus of the class scope testing. It corresponds with the classical definition of unit testing. Inheritance testing is a new level of object-oriented testing, so there is no analogy for inheritance in traditional languages. [10]

The object-oriented world shares integration and interaction testing with the procedural world, although it tends to be more complicated because there exist many objects there, and the parameters of methods may themselves be objects with their own state dependencies. Integration testing is not a separate "phase" in the object-oriented development, but rather occurs throughout a project because of the nature of the object-oriented development process (incremental development) [8, p. 326]. Clearly, each

development increment requires integration testing. The purpose of the integration testing is to stabilise the component, so that the component testing can be done.

Cluster testing in the object-oriented world corresponds to subsystem testing in traditional languages. And finally, system testing is performed in the same way as in traditional languages.

2.2.2 Encapsulation

Encapsulation is a technique that enables information hiding. The physical details of a data structure, device interface, or other software component are not visible to other modules [8, p. 1085]. The only way that the components can interact with each other is through the interfaces. The definitions of interfaces and information hiding encourage modularity of the programs and allow the software to be allocated to independently developed units. But on the negative side, they cause testing problems for object-oriented systems.

From the testing point of view, encapsulation has it pros and cons. It can prevent a few bugs, which are quite common in traditional languages such as global data, and because encapsulation improves the modularity, testing can be limited when changes are applied. However, encapsulation can also present an obstacle to testing. Because of encapsulation, object-oriented languages make it difficult to directly set or get the concrete states, which are required by testing. To overcome this testing obstacle several approaches are proposed [8, p. 72, 11]:

- Increase the visibility of the features by modifying the Class Under Test (CUT) and by defining additional methods. The problem of this technique is that it indirectly violates encapsulation and can change the initial behaviour of the CUT.

- Another approach is to use inheritance. The CUT is inherited by another class, in which additional operations are defined. Then this new subclass is used to test the original class. But this approach also has its problem: if the inherited features are not visible to the subclass, all the superclass features cannot be tested and the using of this method is futile.

- To use language-specific features is equally one way to test encapsulated features. For example, C++ provides the concept of a friend class. By defining a friend, the class under test allows one class (the friend class) to access and modify the features of the CUT.

- Assertions are similarly one way to overcome obstacles to monitoring the states of objects. By means of assertions we can check anything that must be true in a certain point of code.

2.2.3 Inheritance

Inheritance is a mechanism in object-oriented languages that allows one class (the subclass) to incorporate the declarations of all or part of another class (the superclass). The subclass can inherit the features of a superclass as such (extension), modify and remove them (overriding), or add new features (specialisation). The advantage of inheritance is that it makes it possible for the subclass to re-use the features of the superclass, but on the other hand, it makes testing complex.

Because inheritance is a form of re-use and it weakens encapsulation, it can raise problems in testing and cause some bugs [8, p. 72]. Also, the fact that the definitions of the subclasses are distributed (deep and wide inheritance hierarchies), leads to bugs and reduces testability.

Inheritance also poses a problem when deciding how much testing should be done for subclasses. First of all, it seems wasteful to test the inherited characteristics twice because of the re-use. Basically, inherited features cannot be trusted in any circumstances and they require re-testing in the context of the subclass, although the superclass has shown its reliability in some context [11]. In addition, new methods of the subclass and the interaction among these methods should always be tested.

Even though inherited features require re-testing, the test cases of the superclass can, to some extend, be re-used in the context of the subclass. These inherited test cases provide a good starting point for testing and can be re-run in the context of the subclass with some limitations. However, this is only a partial solution for testing the subclass and needs an implementation of additional test cases unique to the subclass. Extra cases are needed to test new features of the subclass and to ensure that they do not disturb the correct behaviour of the original features. Table 1 outlines how much testing should be done for this scope [8, p. 512].

Table 1. Degree of testing for flattened class scope.

	Extension	**Overriding**	**Specialisation**
Re-test method in context of subclass	Minimal	Full	Full
Re-use superclass method test cases	Yes	Probably	-
Implement new subclass method test cases	Maybe	Yes	Yes

A special case of superclass in inheritance hierarchy is an abstract class. Abstract classes are classes that cannot be instantiated, and due to this there is clearly a difficulty in testing. The simplest way to overcome this obstacle is to develop an instantiation to test an abstract class. This can be achieved by defining a concrete subclass, in which pure virtual functions are provided with dummy implementation, just for testing purposes [10].

Instantiation can be also accomplished by making modifications to class under test. This is illustrated in Figure 3.

```
class A {

public:

//...

#ifndef TEST
    virtual void method1()=0 ;
#endif

//...
} ;
```

Figure 3. The abstract class.

2.2.4 Polymorphism

Polymorphism is the third essential part of the object-oriented paradigm. Polymorphism is the ability to bind a reference to more than one object. Basically, it means 'many forms'. The reason why polymorphism and dynamic binding are so important is because they provide flexibility to object-oriented systems and allow the creation of extensible programs that can be grown during the project when new features are desired. But just like inheritance and encapsulation, it provides unique testing problems and bugs that cannot be found from traditional languages [8, p. 68].

One can think that the equivalent statement for the polymorphism in the procedural languages is CASE, but the difference is that the CASE statement explicitly enumerates the cases, whereas the choices for binding in polymorphism are determinated at run-time. Since the actual actions performed depend on run-time conditions, which is a much more complex way than that determinated by traditional control flow constructs, things come more complex in testing. It may be difficult, but not impossible, to identify and exercise all actions and bindings.

Polymorphism and dynamic binding can cause both client and server to fail. Client (e.g. message with polymorphic argument(s)) can fail in at least three ways: 1) client fails to meet the preconditions of server object, 2) unpredictable binding occurs, and 3) a client has been tested, but changes and extensions are applied to a server class [8, p. 439–440]. To expose all these interface faults, test suite for client requires that each binding to a polymorphic server should be exercised at least once.

The server poses a problem if it fails to meet the Liskov Substitution Principle (LSP). LSP states that methods that use pointers or references to a base class must be able to use objects of derived classes without knowing it and causing a failure [12]. Hence, one should verify that derived class's contracts are consistent with the contracts of all of its superclasses [8, p. 514].

2.2.5 Test design strategies

Traditionally, black box (functional) testing is used in higher level of testing, and white box (structural) testing is used in lower level testing. In the object-oriented world, the balance shifts towards black box testing since it is hard to visualise code structure at levels higher than individual methods. [10]

As mentioned earlier, the traditional white box approach is less applicable to object-oriented software, because the methods are small and simple. Since most methods are

very simple programs, the coverage of these methods can be easily obtained, and many errors are not likely to be uncovered. Although it is easier to test methods in object-oriented systems than in the traditional environment, the complexity has nevertheless not disappeared, it is only pushed to interfaces between objects. So, it is not enough to monitor which statements have been executed and which conditions and branching decisions have been exercised, in addition we need to record:

- Inter-object message types and the characteristics of the parameters passed with each message, and

- code coverage in every method in every class which has inherited it.

Tom McCabe sets one coverage metric for implementation-based testing at the class level; he defines a safe class as follows:

A safe class is one where all methods have been tested to total branch coverage, integration testing has been done with all objects on which the methods rely, every method has been called at least once from every context of every call site (that is from every instantiation in every sub-class) [10].

In contrast to white box testing, black box testing is much the same for object-orientation systems as for any other development method. There are a number of black box techniques that can be applied to object-oriented systems. Equivalence partitioning and boundary value analysis can be applied at any level. In equivalence class testing partitions are defined so that if any single test value from partition passes or does not pass, then all other values in the partition are expected to pass or not pass [8, p. 402]. Boundary value analysis is used for strengthening the equivalence class testing. It concentrates on the boundaries of equivalence classes, as they are often proved troublesome.

State-based, in which test cases are derived from the statechart, is also particularly appropriate to object-oriented systems, because objects contain state-information and behaviour may be modelled at any scope by state machines. In addition, new black box techniques are developed for the object-orientation. [10]

2.3 Utilisation of software modelling techniques in testing

The Unified Modelling Language (UML) is language for specifying, visualising, constructing and documenting the artefacts of software systems and it has become de facto standard for object-oriented modelling [13]. From testing perspective, the UML is

interesting since as a modelling language it provides a source for test design. Sequence, class, use case and statechart diagrams can be used to design tests. In addition activity, collaboration, component and deployment diagrams are useful for testing purposes. But on the other hand, these diagrams can be inadequate for testing and need extensions to increase testability.

2.3.1 Use case diagram

A use case diagram shows the relationship among actors and use cases [13]. What are perceived, are the system tasks important from the user's point of view. Use cases provide some of the information for system testing representing many kinds of system requirements, for example functional requirements, allocation of functionality to classes, user interfaces, and user documentation [8, p. 276]. But then again, use case diagrams need extensions and definitions of additional information to increase testability.

Binder points out four limitations of use cases necessary for system test design. First, the domains of each input and output variables are not included in the UML. Second, input/output relationships among use case variables are likewise not part of the UML. In addition, relative frequencies of each use case and sequential dependencies among use cases are absent in the UML. [8, p. 280–281]

To make use case diagrams more beneficial for testing purposes, we have to pay attention to those four limitations and provide extra information necessary for testing by defining extended use cases. To construct extended use case, we define variables that participate in determining the response of use cases and their domain constraints. Once the operational variables (i.e. abstract states of the system under test, explicit inputs/outputs, environmental conditions) are specified, logical relationships among variables are modelled leading to an operational relation for each use case. Finally, the relative frequency of each use case is defined.

2.3.2 Class diagram

Class diagrams are used to construct the static structure of the model. They represent various static relationships among classifier elements, e.g. classes. Despite the name 'class diagram', the model may also contain interfaces, packages, relationships, and even instances, such as objects and links [13].

Class diagrams show different types of relationships among classifiers. These relationships are for example generalisation (e.g. a superclass generalises a subclass), association, and aggregation. Where association represents a loose binding between classifiers, aggregation relationships identify classifiers that are essential parts of another classifier. The relationship can be so tightly coupled that the contained classifier will be created and destroyed along with the container.

To develop tests based on the information provided by the class diagram, the concentration is mainly on associations among classes. To test associations, it is tried to verify the implementation of the associations between classes, because the implementation may contain several kinds of faults. Faults related to association can be for instance missing link, wrong link, and incorrect multiplicity: the implementation rejects a legal combination or accepts an illegal combination. [8, p. 286]

2.3.3 Sequence diagram

Sequence diagrams show interaction arranged in time sequence [13]. In sequence diagrams, time progresses from top to bottom (vertical dimension) and the horizontal dimension represents different objects. Sequence diagrams are typically, but not always, used to design the collaboration of objects to implement a use case.

Sequence diagrams provide information to design tests for the subsystem (system) scope. Although the sequence diagram provides information to design tests, it is not a highly testable model as such and needs to be first transformed into a control flow graph (see 3.2.1.). When the flow graph is completed, we can develop test cases. Each of these test cases is basically a possible entry-exit path in the control flow graph, and execution of these paths will reveal bugs in the implementation. However, there are two significant details that are usually absent in sequence diagrams: bindings to polymorphic servers, and exception-handling paths. These paths should equally be included in the test suite. Furthermore, sequence diagram does not always depict all the scenarios of the use case. Consequently, this makes sequence diagrams less useful for test design as all the flows of executions cannot be tested. [8, p. 290, p. 585–587]

2.3.4 Statechart diagram

A statechart diagram is a graphical presentation of a state machine. Statechart diagrams describe the behaviour of the model elements such as classes, subsystems, use cases, actors, operations, or methods, and are highly applicable to object-oriented analysis and

design since the behaviour of objects can be modelled by state machines at any scope [13].

Statechart diagrams also provide necessary information needed in testing. But sometimes this information is not so clearly available and needs to be searched out by testers: all possible events, for instance, are not explicitly defined at each state. On the other hand, the UML definition of statechart diagrams allows drawing nontestable and ambiguous statecharts. To make statechart diagrams more suitable for testing purposes, extensions are needed. The FREE (Flattened Regular Expression) state model is defined to meet these testing requirements [8, p. 204].

Although the FREE state model has definitions and restrictions not found in the UML, the FREE state model can be constructed using the UML [8, p. 204]. If the UML state model follows the FREE model definitions and restrictions, it is already testable, if not, testers have to close this gap by transforming the UML model into the FREE model. However, the FREE model is not absolutely necessary, but any behaviour model will do if it provides the same information.

The FREE state model can be developed as follows [8, p. 242]:

- Validate the model using the checklists,
- expand the statechart, and
- develop the response matrix.

The validation of the model is part of reviews and inspections and it focuses on the structure, state names, guarded transitions, subclass behaviour, and robustness of the state model. Using these methods is strongly recommended, since they will discover faults in the early stages of software development process and will also ease testing.

Expanding the statechart means that one has to flatten the statechart. For inheritance, flattening is accomplished by combining the statechart diagrams of the superclasses with the statechart diagrams of the subclasses. Let us suppose, we have a class hierarchy where a superclass and a subclass are modelled as a state machine. To generate a complete test suite for the subclass, we have to expand the statechart diagram of the subclass by combining it with the statechart diagram of the superclass. As a result, this model shows explicitly all the possible states and transitions for the subclass. In addition, expanding the statechart involves the orthogonal states (i.e. concurrent (and) states) and nested states to be flattened to make the statechart diagrams more testable. These flattened views explicitly show all the possible states and transitions, but are still

not adequate for testing purposes, since the responses must be specified in a more detailed way.

Where an expanded statechart diagram visualises transitions and states, a response matrix shows responses generated by a state machine. All the implicit responses have to be specified in the response matrix so that all event responses may be tested. Developing a response matrix concerns issues of how to define implicit responses from the incompletely specified state model. Usually there are two interpretations: "do not allow any events that are not explicitly represented" or "ignore events that are not explicitly represented" [8, p. 225].

3. Test design

Test cases are designed in parallel with the corresponding software development phase. The test case design is started from the top of the system as the system scope tests are developed first and accomplished, when tests have been planned for the individual units. Unlike the test case design, test cases are executed from bottom to top. First, the small components are tested and it is verified that they meet their intended behaviour. Next, the larger components are assembled from the smaller components and these are tested. This incremental process, in which smaller components are used to build larger components, is continued until the system scope is reached. When the system scope is attained, the whole system is finally tested.

In this chapter, some methods to design tests for various levels of the software testing, are given. First, the method scope testing techniques are provided. This concerns issues of how to select test values and how to cause a particular path to be taken (i.e. achieve coverage). After the method scope tests are designed (or at least planned), the message sequences are tested within a class. The purpose of the class scope tests is to verify the interactions of the methods, which cannot be done by testing the individual methods in isolation.

Class testing may interleave class and integration testing, as it is sometimes futile to the test classes apart from their servers. Thus, the integration process among classes should be considered, when class scope testing is planned. The third part of this chapter gives some integration testing strategies for levels higher than individual classes. And finally, system scope testing technique is presented.

3.1 Selecting test values

Most of the testing is, basically, choosing the inputs and evaluating the outputs. The definitions of inputs should lead to efficient testing and yet, keep the test suites small. The traditional techniques for choosing the inputs have been equivalence class partitioning and boundary value analysis. These two methods are also effective for object-oriented testing, although some extensions are furthermore needed. The Category-Partition strategy combines the elements of the equivalence class partitioning and boundary value analysis, and it could be used on any method or testable function at any scope. The One-by-One Selection Criteria provides a straightforward and effective way to detect domain errors. The Category-Partition and the One-by-One Selection Criteria techniques are discussed in later sections.

Equivalence class testing is a partition (i.e. any subset of all possible inputs) identification strategy. An equivalence class is a partition defined so that if any single test value from partition passes or does not pass, then all other values in the partition are expected to pass or not pass. Therefore, identifying the equivalence classes is the key to successful testing.

Suppose we have one input variable, a month, which has a range defined as $1 \leq \text{month} \leq 12$. This gives us three different partitions, one valid: $1 \leq \text{month} \leq 12$, and two invalid: $\text{month} < 1$ and $\text{month} > 12$. Once the partitions have been identified, test cases are derived by using one element from each equivalence class [3, p. 75]. Test cases for the month are shown in Table 2.

Table 2. Test cases for input variable month.

Test case ID	Month
TC1	5
TC2	15
TC3	-5

But if we choose the classes more carefully (by thinking the number of days in a month), we will be dealing with different classes: a month has 30 days, a month has 31 days, and a month is February. As a result, the idea of equivalence class testing is generally effective, but does not provide firm guidelines for choosing classes.

When an equivalence class is limited within a range, **boundary value analysis** can be used to strengthen the equivalence class partitioning. Boundary conditions are often proved troublesome, and the values that lie at the boundaries of the equivalence classes are chosen. Input variable values at their minimum and maximum, just above minimum and just below maximum are selected. Also, a nominal value should be chosen [3, p. 58].

To generate test cases using the boundary value analysis, all but one value are held at nominal values, and the remaining variable is assumed the minimum, just above minimum, just below maximum, maximum, and nominal. This should be repeated for all input variables, and so, one ends up with $4n + 1$ test cases (n is number of input variables). If a more exhaustive test suite is wanted, every possible combination of input variables is used (the number of test cases is 5^n). [3, p. 59–60]

Boundary value analysis can be also extended: one simply adds test cases that are slightly below minimum and above maximum. This testing is called **robustness testing**. Figure 4 illustrates a normal use of the boundary value analysis. An extended use of boundary value analysis is depicted in Figure 5.

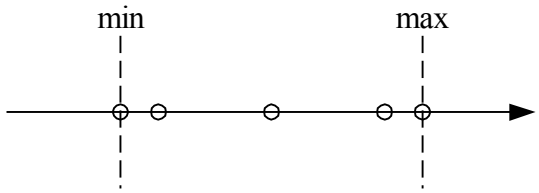


Figure 4. Test cases using boundary value analysis.

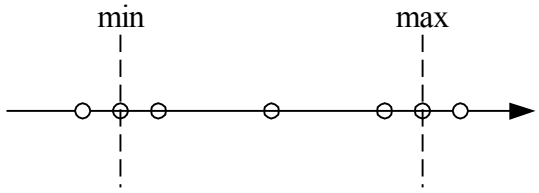


Figure 5. Test cases using extended boundary value analysis.

When these techniques are used in the object-oriented world, some additional information is needed. One needs to consider explicit inputs (i.e. message parameters), class variables, and the state of the object when defining partitions. The explicit inputs can be applied in a similar way in both traditional and object-oriented world. However, this is true only with the primitive data types: integer, floating-point, Boolean, and string. In the object-oriented world, message parameters can be more complex data types, i.e. objects, with their own state dependencies, than in the traditional environment. If global and system environmental variables can affect the output or they can be changed, they should be also examined.

3.1.1 Category-Partition

Testing methods in isolation is impractical, because inter-method (intra class) interaction is not simultaneously tested. However, classes are tested by sending messages to them one at a time, and a class test suite therefore consists of method tests. To test methods, one needs to consider classes as a whole. It is necessary to consider all the elements that determinate the function's response. In object-orientation (at the

method scope), they are class, global and system environmental variables, message parameters, and state of the object [8, p. 421].

The Category-Partition technique was developed for generating black box tests for parameterised functions [14], and it has been adapted to the object-oriented system in [8, p. 419]. The Category-Partition strategy is a straightforward technique to construct test cases for methods in the object-oriented systems. The first step is to identify input and output variables of Method Under Test (MUT). Class, global and system environmental variables, message parameters and state of the object should be considered in analysing the variables. Second, these elements are classified into *categories*, which are similar to equivalence classes, and then categories are *partitioned* into specific test values (choices). Choices can be identified by using the One-by-One Selection Criteria, boundary value analysis, or considering special cases. Once categories are partitioned into choices, constraints are determinated among choices (choices may be mutually exclusive or inclusive). And finally, test cases are generated by producing a cross product of all choices. [8, p. 422–423, Ostrand]

The Category-Partition technique is similar to equivalence class testing with boundary value analysis. It therefore suffers from same inadequacies of heuristic approach. Identification of categories (equivalence classes) and choices (e.g. boundary values) is an intuitive method: different testing persons may end up with different categories and choices. The test suite can also become unpractically large, since the size of the test suite is cross product of all choices minus some constraints.

3.1.2 The One-by-One Selection Criteria

The basic idea of the One-by-One Selection Criteria (The Simplified Domain-Testing Strategy [15]) is to detect a domain error by determining whether a border shift has occurred. To detect a border shift, one needs to sample the potential displaced areas. This can be accomplished by using one on point and one to two off points for each domain boundary. [15]

The selection rules for choosing on and off points are simple. For inequality borders (>=, <=, >, <), one on and one off point are required. The requirement for choosing an off point is that it should be as close to the on point as possible; for integers, this dimension is one [15]. Table 3 illustrates the choosing of the on and off points for inequality borders.

Table 3. On and off points for inequality borders.

Boundary condition	On point	Off point
x < 10	x = 10	x = 9
y <= 10	y = 10	y = 11

For equality or nonequality borders (==, !=), one on point and two off points are needed. Again, the points should be as near each other as possible [15]. Table 4 depicts the on and off points for an equality border.

Table 4. On and off points for an equality border.

Boundary condition	On point	Off points
z == 10	z = 10	z = 9, z = 11

One on point and one off point is needed for nonscalar types, i.e. for Booleans, enumerations, and strings. The nonscalar types either conform to the condition or not. Thus, on point is a value that makes the condition true and off point is the one that makes the condition false. [8, p. 411–412]

If a complex data type, that is object, participates in an input domain, one needs to model its boundaries by defining abstract state invariants. An abstract state invariant is a Boolean statement that determinates the state of the object. For example, the variable iBalance determinates the state of Account class. If iBalance is below zero, then Account is in the Overdrawn state, else (iBalance >= 0) it is in the Open state. Hence, the abstract state invariants for Account class are: iBalance < 0 for Overdrawn and iBalance >= 0 for Open. The selection rule for on and off points is that the condition for the invariant is made true once and false once, on and off points respectively. Table 5 illustrates this as follows:

Table 5. On and off points for Open state.

Boundary condition	On point	Off point
Account.isOpen	iBalance = 0	iBalance = -1

The nominal values are chosen after the on and off points to complete the test suite. These values can be developed by guessing, by thinking special cases, or by using a random algorithm. The restriction for choosing nominal values is that the same values should be avoided, since they will decrease the possibility of revealing unexpected bugs. On points should not be used either.

To generate test cases using the One-by-One Selection Criteria, we apply the same approach as in the boundary value analysis: the size of the test suite is the sum of the choices plus one. Test cases for conditions x < 10, z == 10 and Account.isOpen (tables 3–5) are shown in Table 6.

Table 6. Test case matrix.

			Test cases							
Variable	Condition	Type	1	2	3	4	5	6	7	8
x	< 10	On	10							
		Off		9						
		Nom			-99	-15	0	5	-115	4
y	== 10	On			10					
		Off				9	11			
		Nom	10	10				10	10	10
Account	isOpen	On						0		
		Off							-1	
		Nom	900	541	878	101	555			55
Accept value			no	yes	yes	no	no	yes	no	yes

The One-by-One Selection Criteria is an algorithmic selection method of test inputs, and therefore it is more straightforward than any intuitive technique. However, the identification and developing of state invariants can be time-consuming if the Implementation Under Test (IUT) uses many objects. But when invariants are defined, test cases for IUT can be developed very rapidly using the One-by-One Selection Criteria.

3.2 Code coverage

One fundamental question of software testing is to determine when to stop testing. Code coverage metrics provide one criterion for this. As a metric, code coverage is the percentage figure of the parts of an implementation exercised by our tests. It can be, for example, measured how many statements and branches have been taken.

By examining the source code, the paths that have to be travelled to achieve a sufficient code coverage are chosen. Typically, the paths that have not been executed by our black box tests are chosen to be travelled. However, since the methods in the object-oriented software tend to be small, the coverage of these methods can be easily obtained by black box tests, and little or no work has to be done to achieve the test stopping criteria.

The same coverage criteria that are used for traditional environments can be applied to object-oriented methods. In effect, testing is much the same at the most detailed level, the method level. On the other hand, the complexity of the object-oriented systems lies in the interfaces rather than in the methods, and there exist only proposals for coverage metrics for levels higher than individual methods [10].

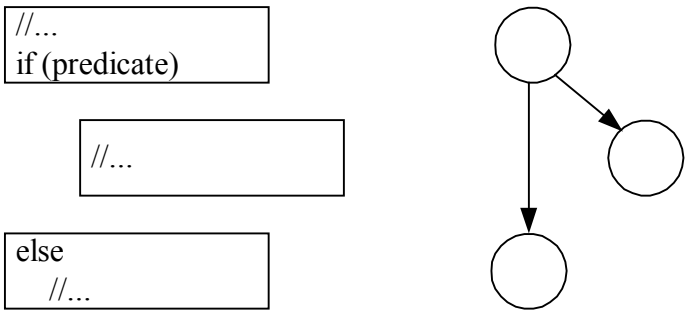
3.2.1 Method scope coverage

The traditional white box techniques to achieve the method scope coverage still apply to object-oriented systems since the object-oriented methods are procedural code. All the code coverage metrics use some form of control flow graph at the method scope. Thus, one needs to understand the white box-based testing to use code coverage information.

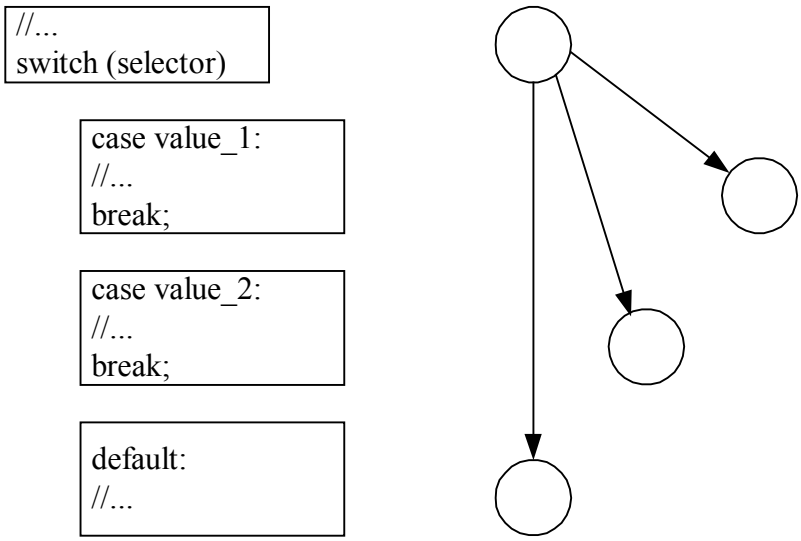
Statement, branch (decision), condition, multiple condition, decision/condition and path coverage are the coverage metrics used at the method level. These coverage metrics have to be produced by coverage analyser tools, since a manual instrumentation is an error-prone and highly time-consuming process. **Statement coverage** is achieved when each statement of the method has been executed. **Branch coverage** is met when each predicate (i.e. Boolean condition or conditions) has been evaluated as false and true. **Condition coverage** requires that each condition has to be evaluated as true and false at least once. **Decision/condition coverage** combines branch and condition coverage. To meet **multiple condition coverage,** each condition in predicate has to be executed as false and true. **Path coverage** is achieved when each entry-exit path of a method has been executed.

Sometimes it is necessary to construct a **control flow graph** and analyse the graph to determine how to achieve the coverage goal as our black box tests have failed to achieve sufficient test stopping criteria. The control flow shows which program segments (i.e. set of statements signifying that if one statement is executed then all the other statements must be executed) may be followed by others. The control flow graph is constructed by using nodes and edges. Nodes are derived from code segments, and edges are the connections between nodes. Control flow graphs for common control structures are shown in Figure 6 [3, p. 118].

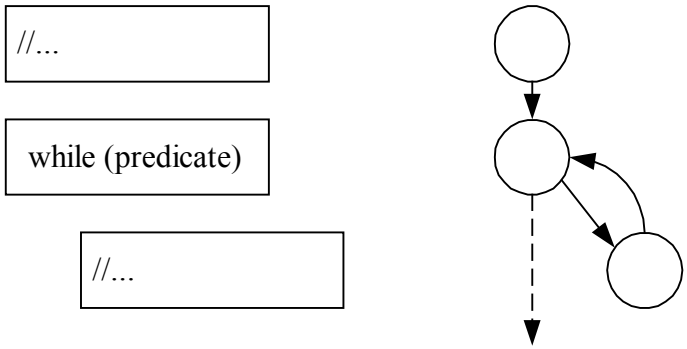
IF-ELSE



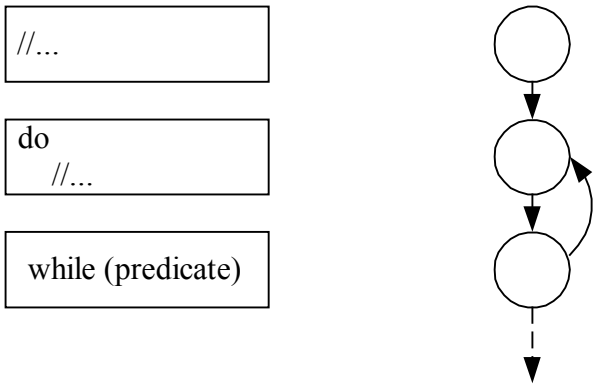
SWITCH-CASE



WHILE



DO-WHILE



FOR

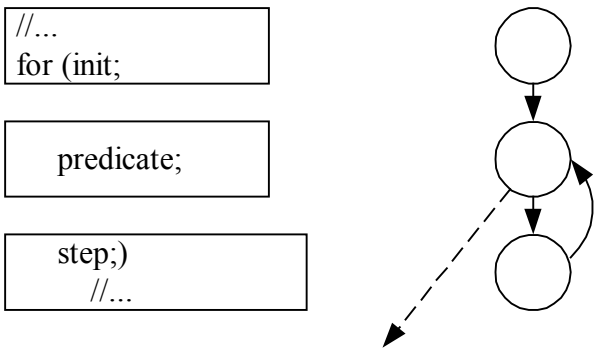


Figure 6. Control flow constructs.

3.2.2 Code coverage tools

In contrast to the manual instrumentation, which is an error-prone and highly time-consuming process, code coverage tools provide metrics that can be considered accurate. The coverage tools work by adding probes into the source code (or object code) through a process called instrumentation. Once the source code is instrumented, the probes capture the visit during the execution and record it in a log for later analysis. In addition, coverage analysis tools analyse and summarise the log. They count different probe messages and report the different coverage metrics.

The instrumentation needed to measure the coverage can itself cause failures and hide the faults by modifying the control structure of the system under test. It can also cause the program running low since the size of the code is increased due to the instrumentation [16, p. 201]. To avoid these problems, the test suite should be re-run on the uninstrumented implementation.

Code coverage analysis plays an important role in the software testing since it can find gaps and redundancies in black box test suites [3, p. 9]. But achieving any coverage goals never guarantees the absence of the bugs. For example, branch coverage requires that each branch is executed once. However, there may be hundreds of data combinations that can cause the same branch to be taken, and testing one set of data is not enough. On the other hand, a thorough testing requires that all paths should be tested for all possible inputs and states [10]. Hence, an exhaustive testing is an impractical task.

3.3 Class scope testing

Previous discussions have concerned method level and inheritance level testing, separately, and these separately from the class testing. However, the class (or small cluster) corresponds with the classical definition of a unit and therefore, class scope testing should bind these two elements together with the inter-method (intra class) testing.

Once it is proved that a class is minimally operable, which can be accomplished by class scope integration, one can proceed with the class testing. At the class level, one can gain benefit from UML statechart diagrams as they show in which order messages can be sent.

3.3.1 Integration testing within class

As stated earlier, the purpose of integration testing is to stabilise the component so that the component testing can proceed smoothly. At the class level, the component to be stabilised is a class, and the integrated parts are methods of the class. The class scope integration plan should show how to develop and test methods and demonstrate that class is ready for a more thorough testing (i.e. class testing). The traditional integration techniques, Big Bang and Bottom-up, can be adapted for these purposes [8, p. 643].

The traditional definitions of Big Bang and Bottom-up emphasise how the program structure is built using modules. The Big Bang technique states that all the modules are combined simultaneously and the entire program is tested as whole [2, p. 515]. This method is usually found ineffective, since the error tracking can be an impossible or at least difficult task. However, the method can be used at the class level in the object-oriented system if the class is small and simple, and a few intra class dependencies exist as well [8, p. 355].

The Bottom-up takes a more sophisticated approach to the integration than the Big Bang by using incremental integration. The Bottom-up integration process starts from the lowest level of the decomposition tree, then the next level is tested, and finally, the highest level of the tree is tested [2, p. 517]. Within class it means that the methods are coded and minimally tested, in the following order: 1) constructor method, 2) accessor method, 3) Boolean method, 4) modifier method, 5) iterator method, and 6) destructor method. Each step also requires that private, protected and public methods are tested, in this order, since the public methods usually depend on the private and the protected methods [8, p. 356]. The Bottom-up integration strategy for class scope is not necessary, but developing and testing in small increments eases the debugging and the error tracking.

3.3.2 Class testing

After it has been shown that a class is ready for more extensive testing, class testing can begin. The class testing focuses on exercising methods in various sequences, since the intra class visibility can cause similar bugs compared to the global data in traditional languages [8, p. 444]. If testing is limited to the method scope, these bugs will never be revealed. However, testing the method interactions does not always focus explicitly on the correct input/output, and classes typically implement features that are not shown by the behavioural model. Hence, it is necessary to interlace the test suites of the methods with the testing of the method interactions.

The statechart diagrams can be used to represent the behaviour of the classes, and the valid message sequences can be derived from these diagrams. By testing all the transitions explicitly shown in the statechart diagram, a basic level of conformance is achieved. A better approach is to achieve all-round-trip coverage, since all-transition coverage does not require particular sequence, and any sequence that exercise each transition once will be sufficient.

All-round-trip coverage is obtained when every sequence of specified transitions beginning and ending in the same state is exercised at least once. Thus, it will achieve all-transition coverage. If a higher level of confidence is needed, then all the events in all the states should be tested. With these test cases it is tested that there are not any illegal transitions present, and some of the error handling code is also tested [8, p. 253, 10].

State-based testing, in which test cases are derived from the statechart, would become simpler if all the classes place constraints in past messages (e.g. application control classes), current content of the object (e.g. container classes), or both (e.g. problem domain classes). However, there are classes that accept any message in any state. These classes typically implement basic data types [8, p. 444].

To test classes which do not constrain messages, a different approach is needed. The basic idea of testing these classes is that one sets the Object Under Test (OUT) to a test case by using modifier methods or constructor and then checks that the OUT is placed in the correct state with a trusted inspector. And then, all accessor messages are sent and it is verified that they report correct values and that the class state is unchanged. The test cases can be developed by using the One-by-One Selection Criteria.

Kim and Wu present a similar class scope testing technique based on data bindings. In their approach, data bindings between the methods of a class are used to derive the message sequences. As each data member of a class can be considered as a shared variable among member functions, the testing order of the member functions of a class can be defined by define-use relationships of data members. As a result, each define-use pair forms a message sequence.

3.4 Integration testing

Integration testing concerns issues of how the program structure is built using components, and which interfaces of these components should be exercised [8 p. 629]. Integration testing strategies can be applied within every scope in the object-oriented world. At the class scope, parts to be integrated are methods, instance variables, and

message parameters; at the cluster scope, a cluster is a component of classes, and at the system level, subsystems are integrated. Because integration testing does not directly test the responsibilities of the component under test, the actual test cases should be developed using appropriate testing techniques. For example, the test cases can be derived from sequence diagrams.

In traditional languages, the architectural design outputs drive the integration testing. The purpose of the architectural design is to develop a modular program structure and represent the control relationships between modules [2, p. 389]. By following this physical structure of the system, modules can be assembled and tested to create larger components, this thereby finally leading to the testing of the whole system. In the object-oriented languages the dependency analysis becomes more complicated and difficult, since, in addition to module call paths, there exist many kinds of dependencies between components. At the cluster and class scope dependencies result, for instance, from the following: inheritance, aggregation, message passing, and objects used as message parameters.

The **dependency analysis** should bind the different types of component dependencies together. The dependency tree does not show an inheritance hierarchy or message passing paths. Instead, it shows client/server relationships [8, p. 636]. The dependency tree can be developed by using a class diagram. Alternatively, the UML component diagram could be used for developing the dependency tree if the component diagram is sufficiently detailed.

The dependency tree forms the basis of integration testing in the object-oriented languages just like the decomposition tree does it for traditional languages. It shows explicit dependencies between components and therefore the assembling order of the system can be derived from the dependency tree. Thus, it can be used to support Bottom-up, Top-down and Collaboration integration.

3.4.1 Bottom-up integration

All the traditional integration orders (Bottom-up, Top-down, Big Bang, and Sandwich) presume that the units are separately tested before the integration testing can begin, and thus the goal of the integration testing is to test the interfaces among units [3, p. 178]. In the object-orientation, attempting to test classes in isolation is sometimes futile, since the servers of the class under test have to be replaced with controllable stubs. This turns out to be often too difficult and complex task [8, p. 644]. The Bottom-up integration takes a different approach to testing. It interleaves the integration testing and the component testing.

The Bottom-up integration process starts from the bottom of the dependency tree by testing the leaf-level components. Then, the next level of the dependency tree is coded and tested with the appropriate test strategy. At this stage, the previous level components are interleaved with current components, and so the interfaces between the client and servers should be also exercised by the test cases of the client components. And finally, a root-level component is used to exercise the entire system. [8, p. 653–658]

The benefit of this approach is that it reduces stubbing compared to the traditional approach, since the upper level components are tested with real components and not in isolation as is done in traditional unit testing. On the negative side of this approach, the driver does not directly exercise the inter-component interfaces, but it exercises the responsibilities of the component under test. This may be inadequate to achieve all the component interactions. In addition, stubs are sometimes needed to throw the exceptions and return values as it gets more difficult to generate desired conditions from the upper levels of the dependency tree. Also, the driver development and maintenance is a significant flaw in the Bottom-up integration.

3.4.2 Top-down integration

Just like Bottom-up integration, Top-down integration interleaves the component testing and integration testing. It tries to achieve stability by adding components in the control hierarchy order, beginning from the top-level of the dependency tree. Where the Bottom-up integration testing suffers from the large amount of drivers, only a single driver is needed for Top-down integration. Since the next lower level of the component under test has to be stubbed, the stub development and maintenance are the most significant costs of the Top-down integration.

The Top-down developing and testing procedure starts from the highest level of the dependency tree, and by stubbing, the lower level services are provided to the highest level. Then, the stubs are removed and the lower level is coded and tested with appropriate stubs. As the lower level components are tested, the same driver can be used to exercise the lower level components and upper level components again, providing regression testing to ensure that new errors have not been introduced. [8, p. 663–668]

As this approach to Top-down integration interleaves component testing and integration testing, it may be difficult to exercise the lower level component sufficiently. Every integration increases the distance between the test driver and the components being integrated, and attaining the high coverage for components under test becomes difficult. The traditional approach to Top-down avoids this major disadvantage of the pure Top-

down integration: the components are tested in isolation with an appropriate test strategy, and the integration testing concentrates purely on exercising the interfaces between components.

The advantage of the Top-down integration is that the cost of the driver development is reduced as test cases can typically be re-used to drive the lower level tests. And in addition, at every stage in the process, one has a working system.

3.4.3 Collaboration integration

Collaboration integration organises integration to support a particular collaboration (e.g. system function) and presumes that the participants of a collaboration are minimally operable (i.e. they have passed their component scope tests) [8, p. 670]. In effect, Collaboration integration is similar to Big Bang integration: the participants of a collaboration are not exercised separately.

The Collaboration integration test process is started by developing the dependency tree. As the dependency tree is developed, the collaborations are mapped to the dependency tree until all components and interfaces are covered. Then, the interfaces are tested by exercising one collaboration at a time. The test suite can be developed for the scope of the integration, for example by using a sequence diagram. [8, p. 671–675]

Collaboration integration is similar to Big Bang integration, and so the participants in collaboration are put together at once. This approach to integration testing suffers from the inadequacies of non-incremental integration testing. As every component in an integration is equally suspect when a failure has occurred, debugging can be difficult. With the incremental integration testing, most recently added components are likely to be buggy. Furthermore, it may be difficult to exercise the lower level component sufficiently with Collaboration integration. Every layer in the dependency tree increases the distance between the test driver and the components being integrated, and attaining the high coverage for lower level components becomes difficult.

The exit criterion for integration testing is that all interfaces and components are exercised. Sometimes it may not be necessary to exercise all the collaborations to obtain the interface coverage if a few collaborations exercise all the interfaces. Since interface coverage may be achieved with a few test runs without exercising all the collaborations, it may result that some interface bugs are missed. However, Collaboration integration test cases will probably be re-used and expanded to test these uncovered collaborations, for instance in the system scope testing.

Jorgensen and Erickson propose a similar approach for the object-oriented integration testing [18]. The integration is accomplished by identifying Method/Message paths (MM-path) for each Atomic System Function (ASF). MM-path is a sequence of method executions linked by messages, and an atomic system function is an input port event, followed a by set of MM-paths, and terminated by an output port event. The ASF is visible at the system boundary, and thus the ports constitute the boundaries of the system. Below, Figure 7 illustrates ports, ASFs and MM-paths.

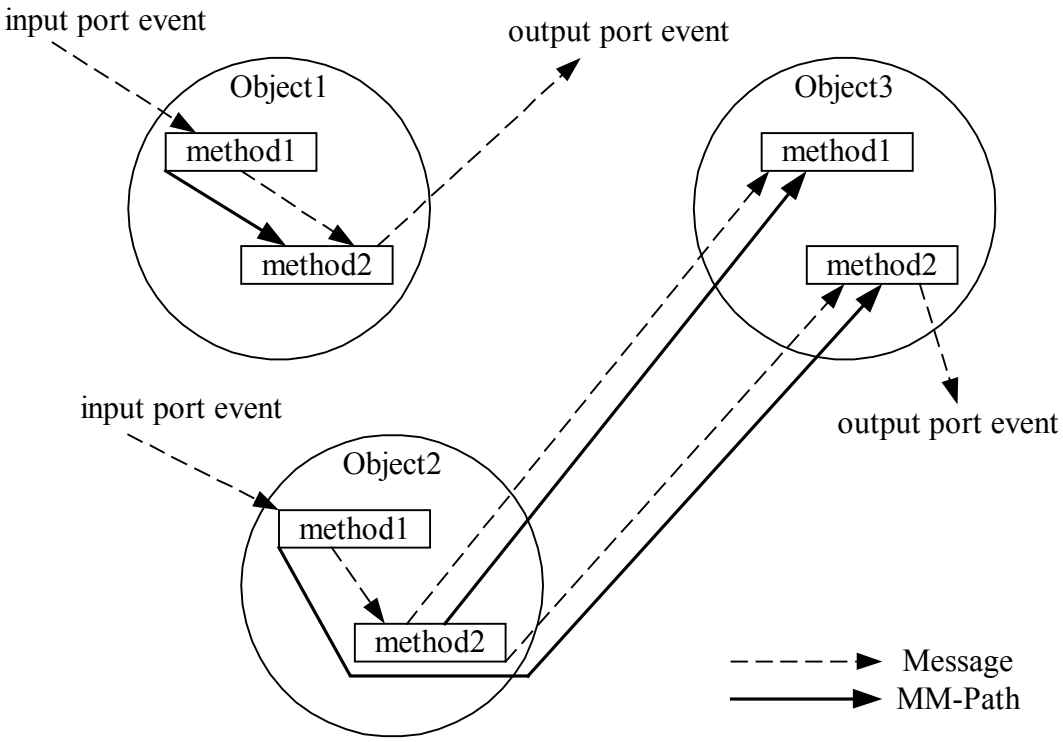


Figure 7. Object network for integration testing.

3.5 System scope testing

As previously proposed, sequence diagrams can be used to test the collaborations of components at the integration level, demonstrating end-to-end functionality. Although all collaborations shown in the sequence diagrams could be tested during the integration testing, there may still exist gaps in the test suite. Sequence diagrams are obligated to show a possible scenario of use cases, where proper implementation of a use case may require several scenarios. Thus, a more thorough testing is needed, and system scope testing becomes necessary.

On the other hand, system testing means more than demonstrating that the system implements all the required functional capabilities. Additional testing is needed after the functional test cases are derived from the system requirements. These test cases address the non-functional capabilities of the system, such as configuration and compatibility, stress, performance, concurrency, recovery, and human-computer interaction testing [8, p. 742].

3.5.1 Use case testing

Use case diagrams can be used to drive the system scope testing. Use cases express most of the system requirements in object-oriented development, and due to this, system scope test cases can be constructed from these requirements. Nevertheless, use case diagrams are seldom test-ready, and more testable use cases are needed. The extended use cases provide this extra information for system scope testing.

To develop extended use cases one needs to define:

- Operational variables,
- domain constraints for each operational variable,
- operational relation for each use case, and
- the relative frequency of each use case.

Operational variables are variables that participate in determining the response of a use case. In short, they can be explicit inputs/outputs, state of the system, or environmental conditions. Different environmental conditions can be identified by considering the interaction between the system and actor, and those that are likely to produce a different system behaviour are selected. [8, p. 724–731]

After the domain for each operational variable is defined, logical relationships among operational variables are developed, this thereby leading to an operational relation. An operational relation is a decision table, in which each row is a variant and each column is an operational variable. The test cases are generated from the operational relation: one true and one false test case is required for each variant.

Binder uses the relative frequency of each use case to determinate that reliability is maximised: those use cases that are used most often are tested more thoroughly than the others. However, two different approaches can be also identified. First, the use cases

that pose the highest risk to the project are tested most heavily. Alternatively, the use cases that are critical to the operation of the system are tested more thoroughly.

The benefit of developing test cases based on extended use cases is that the development of the extended use cases will find errors and omissions in the design, and thus, developing the extended use cases from the outset is preferable. Another advantage of deriving test suites from the use cases is that use cases reflect the user point of view, which is often effective in revealing omissions.

4. Test automation

So far, only one activity of the test process (i.e. test design) is discussed. In addition to test design, the test process encompasses test case implementation, execution and comparison between actual outcomes and expected results. These remaining activities of the test process are described in the first part of this chapter.

Object-orientation makes testing difficult, emphasises regression testing and creates unique errors that cannot be found from traditional languages. Hence, a test driver implementation should take these special needs of the testing of object-oriented software into account. A test driver implementation for object-oriented software is presented in the second part of this chapter.

4.1 Test process

The test process includes a number of distinct activities each of which addresses a different development phase of test cases. The test process encompasses test design, test case implementation, execution, and comparison between actual outcomes and expected results. These activities can be considered sequential. Test design is accomplished before test case implementation, and a test case must be implemented before it can be run, and run before its actual outcomes are compared to expected results [19 p. 14]. The activities of the test process are illustrated in Figure 8.

The first activity, **test design,** involves designing test cases based on the responsibilities of component under test. Test cases based on code analysis and suspicions are also added to a test suite. And finally, expected results for each test case are developed. The expected outcomes include explicit outputs or things that are created or changed. Also, things that should not be changed or things that should be deleted should be included in the set of the expected outcomes. The test case design should be carried out parallel with the associated development activity before the software to be tested has been built. For example, extended use cases will be developed as soon as nontestable use cases are available.

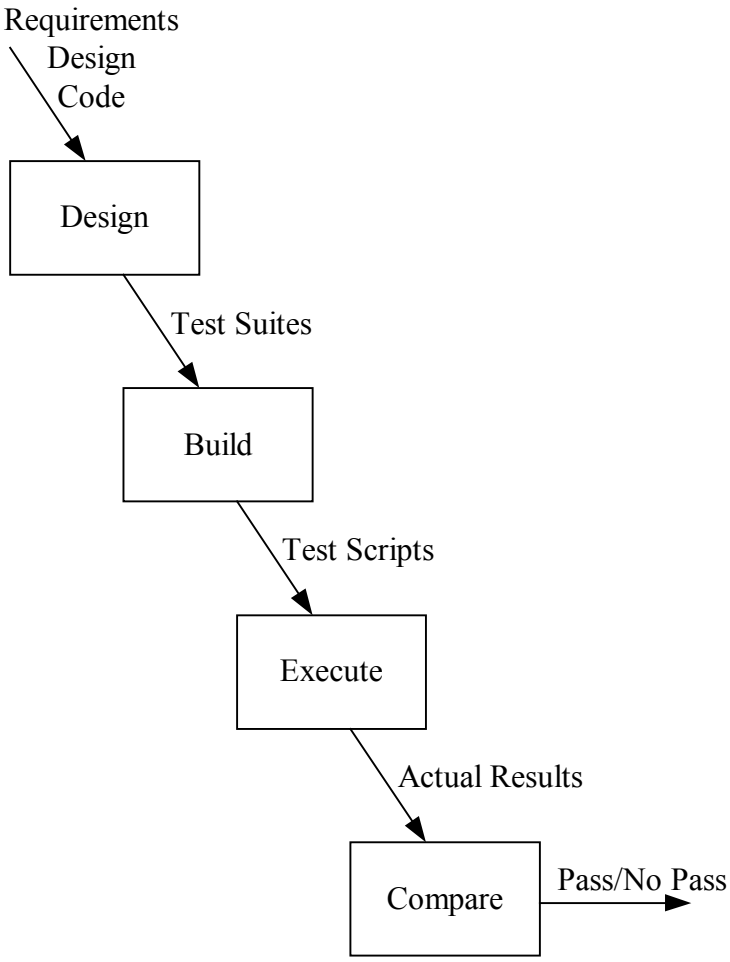


Figure 8. Test process activities.

After the design is completed, **test scripts** are **implemented**. A test script is a program typically written in a procedural language that executes the test suite. If manual testing is indicated, a test script is merely a test procedure, and no other software, besides to the software under test, is needed. A test script may implement one or many test cases, set-up environment, bring the software under test to the required pre-test state, compare the resulting outputs, and finally do some clear-up procedures. To save some time later, the test case building activity can be prepared in advance before the software to be tested has been built.

In the third activity, **test cases** are **executed** using the test scripts. For manual testing, a tester will enter the inputs, observe outcomes and make notes about any problems as they occur. For automated testing, a testing personnel starts the test tool and tells it which test cases to execute. Test execution can be done only when the software to be tested is ready.

At the final stage of the test case development life cycle, **outcomes of each test case** are **compared** to expected results. As stated earlier, the comparison of the test outcomes

can be done only after the test case is executed. This is not always the case. Typically, the comparison of some outputs, such as messages sent to a screen, is done in parallel with the execution. On the other hand, some comparison is done after the test case is executed. As a result, comparison automation needs to use a combination of these two approaches.

If actual and expected results are equal, the software under test has passed a test case; if they are different, the software has not passed a test case, and this thereby results in debugging of the software results. Sometimes it is not the software under test that has failed, but rather, test suite is executed in a wrong sequence, expected outcome was incorrect, test environment was set-up incorrectly, or test was incorrectly specified. Consequently, the outcomes should be verified, since the comparison tool cannot say whether the output is correct or not. The tool can only compare the outputs and flag the differences. It is the testing personnel who have to verify that the results being compared are correct. [19, p. 17]

4.2 Test driver implementation

Any aspect of the test process can be automated, but only a few of those are truly amenable to automation, and since test automation contributes an extra overload to testing, a careful judgement is needed. Typically, those parts of the test process that are repeated most often are automated. Also, the nature of the test process activity makes one activity more useful than the other for test automation.

Test design is intellectual work at least in selecting the inputs and outputs. A tool does not have imagination to select the proper values, and everything must be spelled out in great detail. In the other end of the test process, testing is much more straightforward and less intellectual than test design. This makes test case execution and comparison more amenable to test automation.

Typically, test case design is done once, but the execution and comparison often. For instance, object-orientation emphasises re-testing because of inheritance, incremental development life cycle, and re-use [20]. When a test case fails and the error is fixed, we may then also want to re-run the test case to ensure proper behaviour. Test cases must also re-run for an instrumented and uninstrumented code, or in embedded systems, where software is developed in a host and then transferred into a target, regression testing is needed. Thus, we automate test case execution and comparison. Figure 9 illustrates an automated test environment.

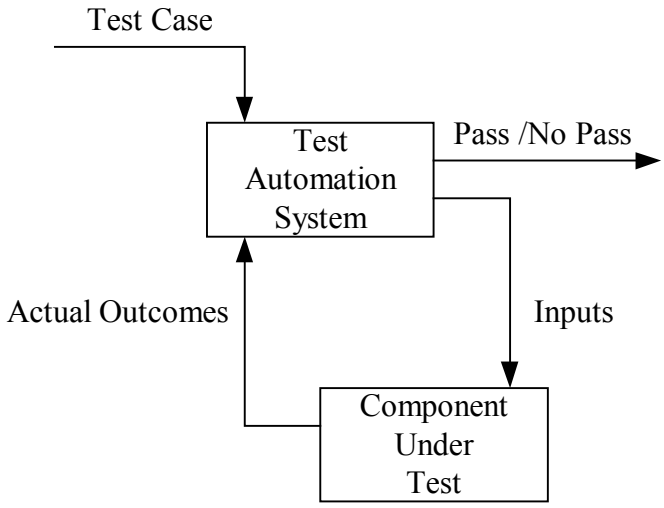


Figure 9. The automated test environment.

It is already shown that object-orientation makes testing difficult, emphasis regression testing and causes unique errors that cannot be found from traditional languages. Moreover, testing of a component typically requires a driver that can simulate the messages sent to CUT as there is no user interface to provide test inputs for the CUT. The object-oriented test automation should provide answers to these testing problems. Hence, the test automation system has to make it possible to:

- Observe the states of the class under test, and
- re-use the test cases of the superclasses.

To re-use the test cases of the superclasses, Firesmith proposes that the original inheritance structure should be duplicated: as each new class to be tested is derived, a corresponding test driver class is derived. The advantage of this approach is that derived classes of test drivers use inheritance to obtain, extend and modify the test cases of their base classes [20]. However, it does not address the issue of how the states of the class to be tested are observed, nor does it demonstrate how the test cases are implemented. On the other hand, it provides a good basis for our test driver implementation.

The test driver implementation takes the difficulties of the testing of object-oriented software into account and combines the solutions that were presented in the earlier sections (see sections 2.2.2., 2.2.3. and 2.2.4.). Hence, it overcomes the obstacles that are caused by encapsulation and information hiding by providing a view to the private and protected features of the class under test. The test driver implementation equally gives a concrete solution of how to re-use the test cases of the superclasses in order to diminish the effort of re-testing. As it provides an ability to re-use the test cases of the superclass, it can be verified that the derived class's contracts are consistent with the

contracts of all of its superclasses (i.e. subclass is a type of superclass), and thus, it is checked that the server classes are LSP compatible. Furthermore, extended features can be re-tested in the context of the subclass by using the test driver implementation. The test driver implementation is depicted in Figure 10.

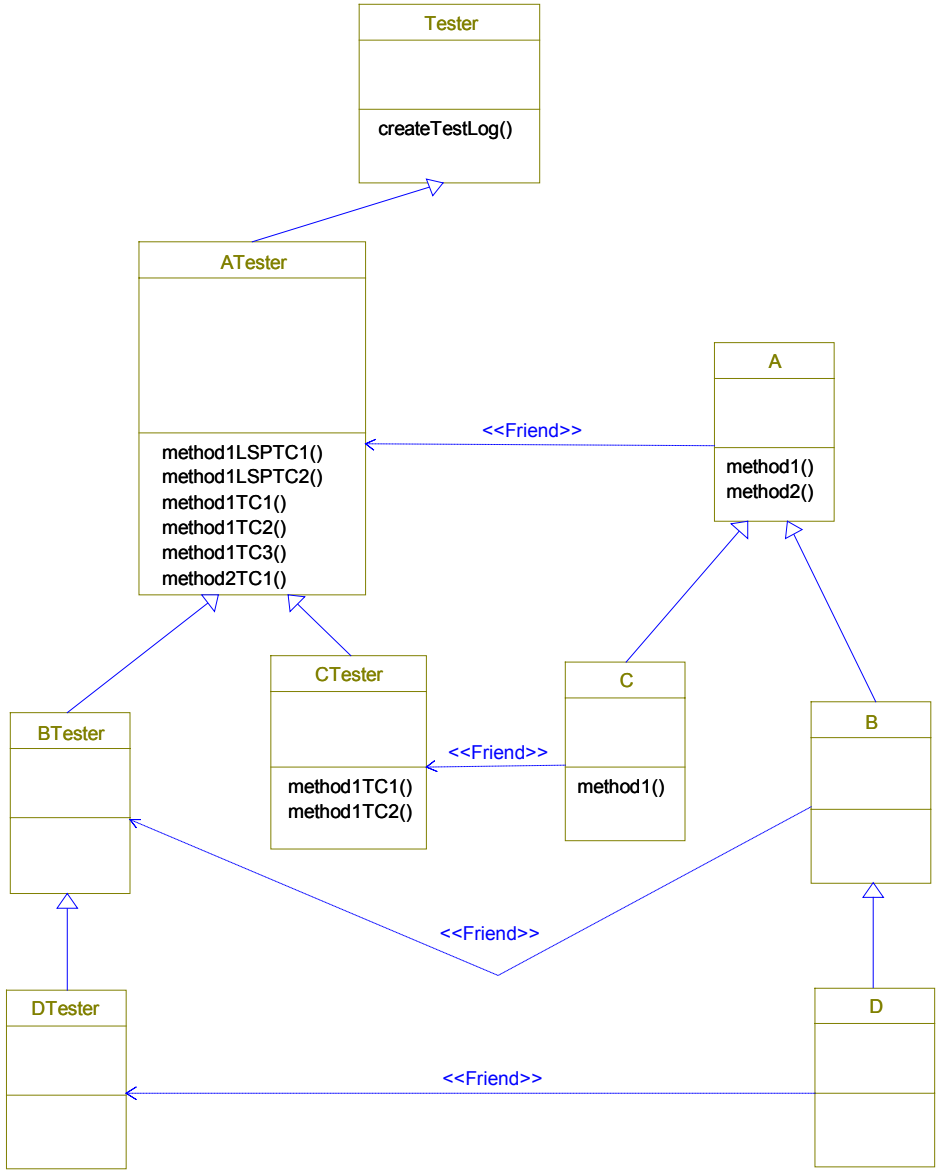


Figure 10. Duplicated test driver hierarchy.

As it can be seen from the figure, test cases are implemented as methods. For instance, there are five test cases for A::method1. These test cases are implemented in ATester. ATester::method1LSPTC1 and ATester::method2LSPTC2 are used to verify LSP compliance and can be re-used in CTester to test that C is LSP compatible with A. However, ATester::methodTC1, ATester::methodTC2 and ATester::methodTC3 cannot be re-used, since the method1 is overridden in C. Thus, new test cases are needed

(CTester::methodTC1 and CTester::methodTC2). On the other hand, A::method2 is re-used as such in C and so the test case from ATester can be re-used in the context of CTester. The purpose of the superclass Tester is to make it possible to record the result from each test case into a file for later analysis.

To observe the states of a class under test, we take advantage of the concept of a friend class in C++. Using a friend class we have an access to all variables of the CUT avoiding the obstacles of encapsulation. We have an ability to examine CUT's private data (i.e. get the states of the CUT) to verify its correct implementation. In addition, we have an ability to initialise its private data (i.e. set the states of the CUT) to a force particular path to be taken. The friend class also gives access to protected and private methods to ensure that they are fully tested.

Another approach to overcome the obstacles of encapsulation is to make modifications to the class under test (Figure 11). On the negative side, this approach requires more work than using a friend class and indirectly violates encapsulation. By making modifications to the class under test, one grants access to encapsulated features for all the classes in the system. Instead, using a friend class, the driver is the one who has access to the features of CUT.

```
class A {

#ifdef TEST
public:
#endif
          int x,y ;

#ifdef TEST
public:
#else
protected:
#endif
          method1() ;

#ifdef TEST
public:
#else
private:
#endif
          method2() ;

//...
} ;
```

Figure 11. The modified class.

5. A case study

In this chapter, the proposed techniques are put into practice, and for this purpose, a demonstration system is built. The system is designed and implemented in the Rhapsody environment, since it supports the full life cycle of software development from requirements to testing. Rhapsody supports the software design by providing various UML diagrams. It equally supports implementation by generating code from the UML diagrams and testing by showing the animated diagrams.

5.1 Rhapsody

Rhapsody (Figure 12) is a visual development environment for real-time embedded systems. It enables software engineers to model the system with the Unified Modelling Language. Use case diagrams, class diagrams (object model diagrams in terms of Rhapsody), sequence diagrams, statechart diagrams and activity diagrams are supported by Rhapsody. However, it does not support implementation diagrams such as component and deployment diagrams. Furthermore, Rhapsody makes it possible to verify the intended behaviour of the system much earlier in the development life cycle by generating code from design and testing the system as it is built.

Use case diagrams in Rhapsody are fully compatible with the UML. They capture the high-level functionality of the system and give the designer the ability to depict the standard relationships between actors and use cases or among use cases. Extension points, generalisations, includes and associations can be shown in Rhapsody's use case diagrams. At the implementation phase, there is not much use of use case diagrams, as Rhapsody does not generate code from the models. Nor can they be used in the test case execution phase.

Typically, the use case diagram itself does not show all the information to implement the use case, but the description of the use case must be put into another document. This extra information describes for instance the name of the use case, actor(s), pre-conditions, post-conditions, possible errors, flow of the execution, and purpose of the use case. In Rhapsody, the description of the use case can be merged into the use case diagram thus avoiding unnecessary documentation.

Sequence diagrams are used to give an example of a use case. The scenario of the use case can be either a "black box" sequence, in which messages between the system and its environment are illustrated, or a "white box" sequence, which shows the internal interactions within the system. The sequence diagrams in Rhapsody show one possible scenario of the use case. Only one entry-exit path exists in the diagram, as there are no

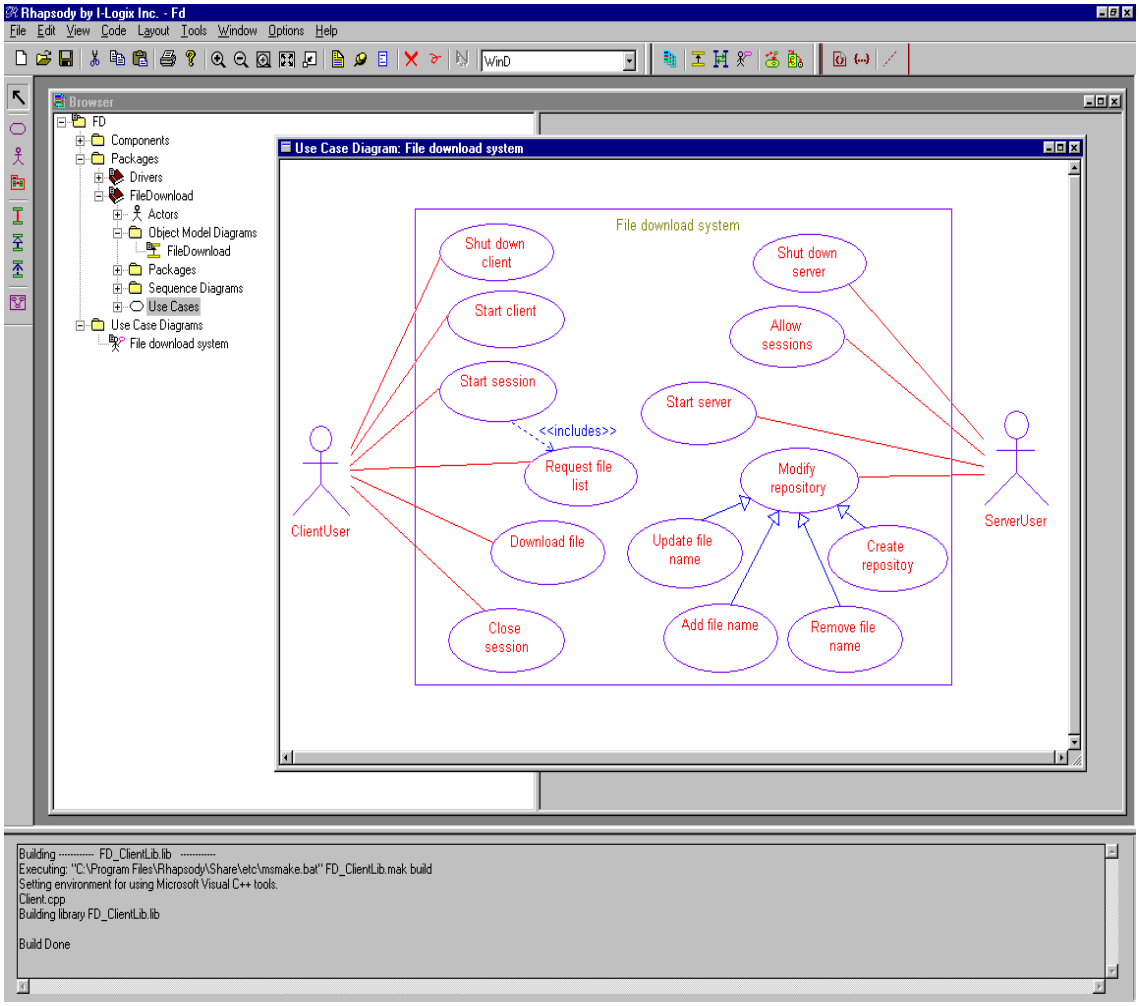


Figure 12. A view of Rhapsody.

conditional or iterated messages. This eases the test case design since one does not have to identify different flow paths from the model, and as a result, one test case is derived from each sequence diagram. This, however, is not an effective way to test the collaborations of the classes. The complete implementation of a use case may require several scenarios, and testing one of them is not enough. Thus, one needs to develop a complete model where each scenario is depicted in a separate diagram and then overlay these diagrams to develop the test suite for the system/subsystem scope.

At the design phase, when an instance line or message is drawn in a sequence diagram, Rhapsody checks whether the message or class already exists, and if not, they are created according to the information given by a user. At the testing phase, the actual

trace of the execution can be captured and compared to the specification. In addition to the interactions of the objects, the values of the arguments of the methods are shown in the animated sequence diagrams. Thus, animated sequence diagrams provide a possibility to check inter-object messages and values of the arguments passed with each message. This may be useful at the integration phase.

Class diagrams are used to show the static structure of the system. In Rhapsody, they depict classes, packages and relations that exist among classes such as dependencies, generalisations, associations, aggregations, and compositions. Also, active objects (i.e. objects that run in their own thread) can be illustrated in the class diagrams. Rhapsody generates low-level code including relations, multiplicities of the objects, and threads from the model, and by generating code from the diagram, it reduces the effort to implement the system structure. Because it provides operations to handle multiplicities and the relations of the classes, Rhapsody speeds up the implementation of the system. On the negative side, if you do not know how to use, or mistakenly misuse generated operations using them is subtle to new errors and failures.

Statechart diagrams provide the most detailed view of the behaviour of classes. They define the behaviour of classes by specifying how they react to events and operations. Combined with the code generation in Rhapsody, statechart diagrams provide a high-level, iconic programming language, which hides the low-level implementation of the statechart. In animated mode, Rhapsody highlights the transition taken and the state entered, providing a view to the encapsulated features of the class under test.

In Rhapsody, classes with statecharts are called reactive classes. Reactive classes can accept asynchronous messages, i.e. messages that happen over time, and synchronous messages, i.e. messages that happen immediately. Asynchronous messages are called events and synchronous messages are called triggered operations. The usage of the triggered operations corresponds to the usage of instance functions in C++. These can return value (this is what events cannot do) and accept parameters, and the client waits until the operation is completed (synchronous communication). The difference is that the triggered operation is included in the statechart framework and the instance functions are not.

Classes can also have primitive operations (instance functions in C++). These are the operations whose bodies must be defined instead of letting Rhapsody generate code for them. Contrary to triggered operations and events, which are accepted only when shown explicitly in the statechart diagram, primitive operations are accepted in every state. Furthermore, primitive operations cannot trigger a state transition. Instead, timeouts and messages, such as triggered operations and events, are used to trigger a transition from one state to another in the statechart.

5.2 The example system

To put the proposed techniques into practice, an example system is built. A simple distributed system is constructed for this purpose. The system follows the so called client/server-system architecture providing clearly two distinct subsystems, the client and the server. The client sends messages, e.g. through the computer network, to the server and the server responds to them accordingly. In addition, the client and the server must use a protocol to communicate with each other and to provide a reliable connection. Figure 13 shows an example of the client/server architecture.

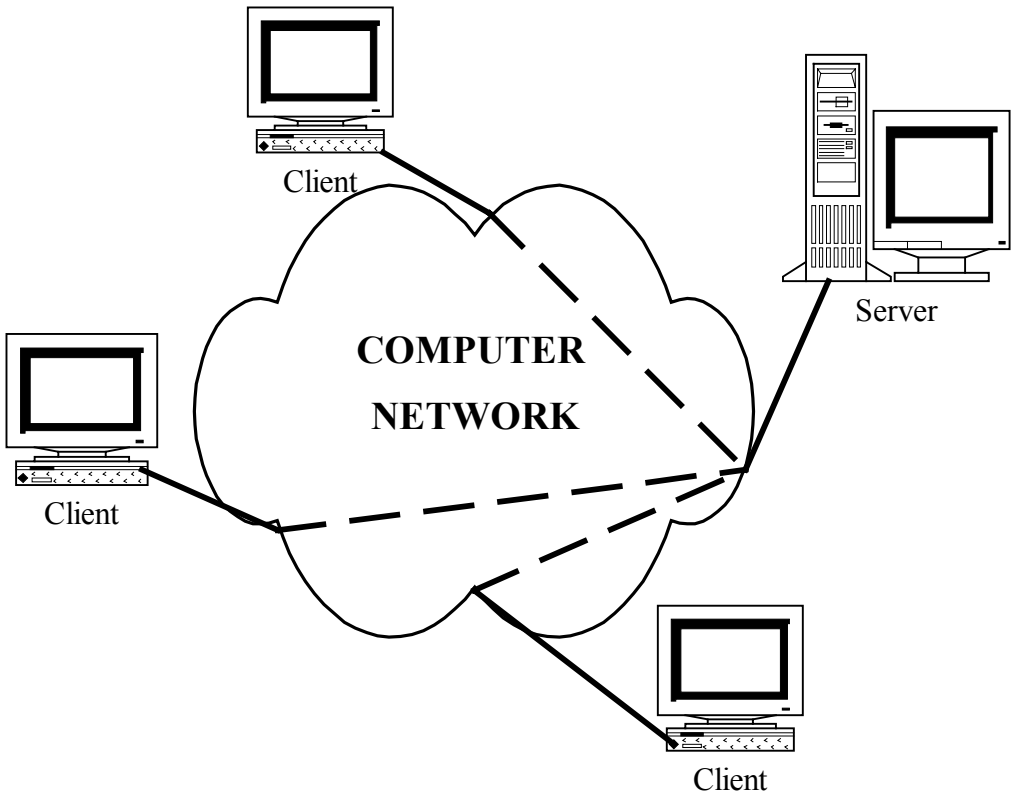


Figure 13. Client/Server architecture.

5.2.1 Requirements

The main function of the example system is to enable the client user to download files from the server's repository through a TCP/IP connection. The more specific problem statement for the file download system is following:

A server user creates a file name repository, where all the names of the files that can be downloaded by the client(s) are put into. The server user can add, remove and update these file names. After the repository is created, and at least one file name exists in the repository, the server user is able to enable the reception of the connections. Furthermore, the server must be able to handle five sessions simultaneously.

The connection to the server is initiated by the client user. After the connection is established, the client user may select the file and download it. Alternatively, she/he may shutdown the connection or may download the repository again.

From the above problem statement, use cases for the entire system are built (the use case diagram for the example system is illustrated in Appendix 1). Typically, after many revisions, the use cases are settled, and the development of the extended use cases is started. First, the operational variables are defined. For instance, the operational variables of the *download file* use case are: status of the client (connected/not connected), status of the server (respond/not respond), file to be downloaded (valid/invalid), and status of the file names (one selected/none selected). Next, operational relation is developed for the use case, and the test cases are generated from the operational relation: each variant requires one true and one false test case. Operational relation for the *download file* use case is shown in Table 7. Table 8 illustrates the correspondent test cases.

Table 7. Operational relation for the download file use case.

Variant	Status of the client	Status of the server	File to be downloaded	Status of the file names
1	Not connected	Do not care	Do not care	Do not care
2	Connected	Do not care	Do not care	None selected
3	Connected	Not respond	Do not care	One selected
4	Connected	Respond	Valid	One selected
5	Connected	Respond	Invalid	One selected

Table 8. Test cases for the download file use case.

Test cases	Operational Variables			
	Status of the client	**Status of the server**	**File to be downloaded**	**Status of the file names**
1	Not connected	Do not care	Do not care	Do not care
2	Connected	Respond	Valid	One selected
3	Connected	Do not care	Do not care	None selected
4	Not connected	Do not care	Do not care	None selected
5	Connected	Not respond	Do not care	One selected
6	Connected	Respond	Valid	One selected
7	Connected	Respond	Valid	One selected
8	Connected	Not respond	Valid	One selected
9	Connected	Respond	Invalid	One selected
10	Connected	Not respond	Invalid	One selected

A similar way to enumerate test cases is to use Category-Partition: when operational variables and their test values (choices) are defined, test cases are generated by a producing cross product of all choices minus some constraints. To detect domain errors, the One-by-One Selection Criteria should be used. Test cases based on suspicions should be also included in the test suite.

A better approach to define extended use cases would be to consider them as the use cases are developed, since they not only give you a better understanding about the problem but also make the use cases more complete and stable. If the system is implemented from the original use cases after which the extended use case tests are applied to the system, these may reveal bugs in the design. Alternatively, these design omissions are revealed at the implementation phase, and programmers may improvise these ambiguities and omissions as they are found.

On the other hand, an experienced system designer would probably include this information in the use case description and thereby the test cases are derived from the description. These test cases include, for instance, the expected flow of events, all

possible error conditions and possible violations in the pre-conditions. However, it is a good practice to speak in terms of extended use cases as they provide a systematic way to define test suites for the use cases.

5.2.2 Scenarios and the structure of the system

After the use case diagram is constructed, scenarios of use cases are created by using sequence diagrams. This is done in parallel with the development of the static structure of the system (class diagram). Rhapsody provides means for the parallel development of sequence diagrams and class diagram, because it checks whether the message or class already exists when an instance line or message is drawn in a sequence diagram, and if not, they are created according to the information given by a user.

Since Rhapsody does not support iterated or conditional messages, a sequence diagram presents only one scenario of a use case. Then it is up to the designer whether she/he wants to depict all the collaborations in separated diagrams. And the more sequence diagrams are illustrated, the better it is for test design purposes.

The *download file* use case of the demonstration system encompasses at least five usage scenarios. It is useless to illustrate all these scenarios for testing purposes, because all the collaborations are tested at the system level. Nevertheless, as Rhapsody supports animated sequence diagrams, in which a trace of execution is shown, it is useful to draw scenarios to test the interface faults. These interface faults include for instance missing methods, wrong method called, and wrong parameters used in a method call. A sequence diagram of the *request file list* use case, which was used prior to use case tests, is illustrated in Appendix 2. With this test case, it was verified that the use case is stable enough for extensive testing.

The class diagram of the system (shown in Appendix 3) can also be used in testing purposes. It depicts client/server dependencies at the subsystem and at the system level providing information useful for integration testing, since dependency trees can be developed from the models by inspection. Figures 14 and 15 illustrate the example system's dependency trees.

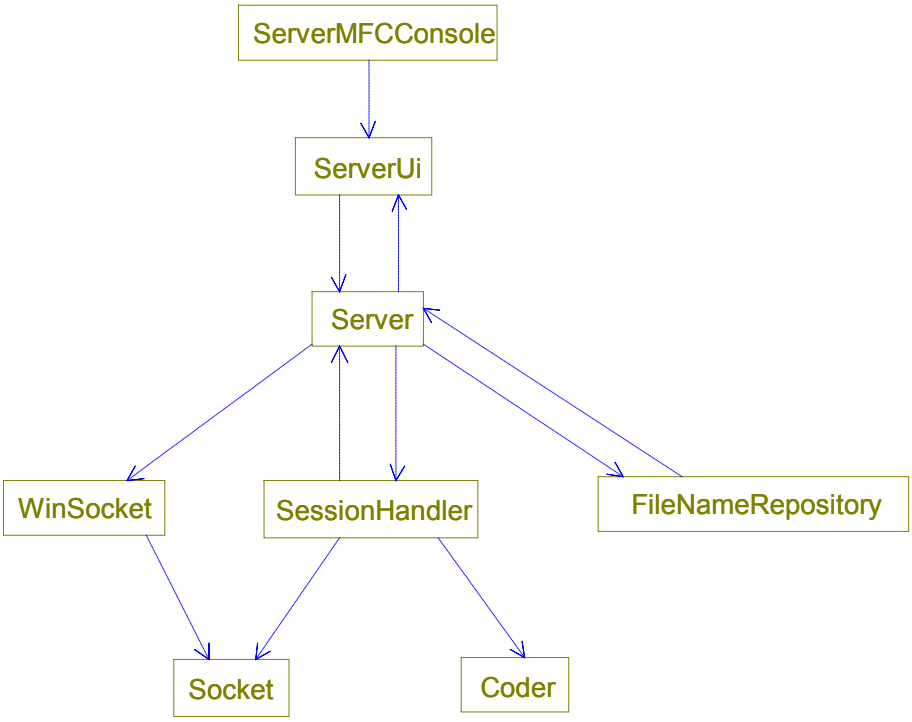


Figure 14. Dependency tree of the file server.

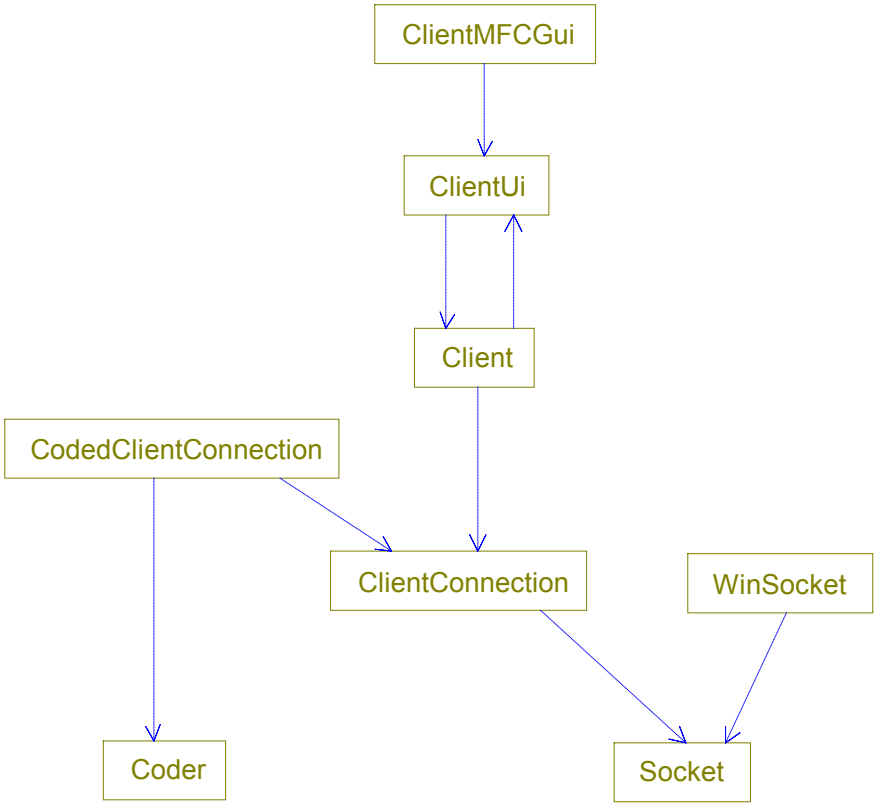


Figure 15. Dependency tree of the file client.

For the example system, the file server is tested in top-down order, and the file client is tested in bottom-up order. The whole system is tested using Bottom-up integration. Another approach could be that the subsystems, the file client and the file server, are developed and tested in isolation, after which integration testing is carried out using Collaboration integration. This turned out to be futile exercise, since the file server is developed and tested first, providing trusted component for the file client. However, if the subsystems are developed in parallel, the Collaboration integration may be useful.

Besides depicting the integration process, a few subsystem and system level test cases can be developed from the class diagram. These tests verify the associations of the classes shown in the class diagram. For instance, it should be tested that the server can handle five connections simultaneously.

5.2.3 Objects behaviour

At the final phase of the design, the behaviour of the classes is developed using statechart diagrams. The operations that are not part of the statechart and are not defined in early part of the design are also developed. This is often accomplished by functional decomposition of the public operations [21, p. 275].

Events, triggered operations and primitive operations clarify the test case development from a state model in Rhapsody. If the event or triggered operation is not defined in the particular state, it is simply ignored and discarded, and due to this, there are no implicit transitions present. Furthermore, there are no illegal transitions, since the code is generated from the diagram. Basically, one can trust this implementation and discard testing every message in every state. Testing all the transitions explicitly shown in the statechart diagram would be sufficient.

The statechart diagram in Rhapsody provides a high-level programming language, and specifying the object behaviour using a statechart diagram denotes rather implementing than designing the behaviour. Hence, deriving the test cases from the model signifies white box testing and the testing shows that the code does what it does; this is not necessarily what it is supposed to do. On the other hand, the action parts of transition are still implemented by the user. These action statements may include complicated control flow paths or they are simply a few statements of code with no conditional branching. In either case, the action statements should be tested. Testing all transitions would once again be sufficient, as it might result in that all control flow paths within the actions would be executed.

Rhapsody eases the test case design for inheritance hierarchies. In order to make statecharts testable, they have to be flattened. For inheritance, flattening is accomplished by combining the statechart diagrams of the superclasses with the statechart diagrams of the subclasses. In Rhapsody, the statechart diagram of the superclass is inherited by the subclass, i.e. the statechart of the subclass (Appendix 5) is initially a clone of the statechart of the superclass (Appendix 4).

For the example system, message sequences within the class ClientConnection (Appendix 4) were developed using the technique described in the section 3.3.2. to achieve all-transition coverage. In addition, test cases for the methods were developed using Category-Partition. As stated, the effort of testing message sequences within a class is diminished because of code generation. Consequently, some of the sequences that would have resulted from the flatting of the nested states could be left out, as the code generation of Rhapsody is not the aim of the testing. Testing every message at every state could be precluded as well.

Category-Partition is an effective means to find errors within methods [8, p. 419]. It could also be used to enumerate tests for use cases. The negative side is that the test case design is laborious. After identifying the variables and their values, constraints and combinations between values have to be identified. This has turned out to be a complex task as the number of choices increases. Thus, some sort of tool support for enumerating the test cases would be useful.

The One-by-One Selection Criteria is useful for methods that include a domain constraints for input parameters (e.g. have pre-conditions). For the example system, there is a method called Coder::encodeMessage whose specification requires that input parameters meet the following assertions:

```
assert (_dataLength <= MAX_DATA_LENGTH) ;
assert (_dataLength >= MIN_DATA_LENGTH) ;
assert (_msgID == VERIFY_CONNECTION ||
        _msgID == REQUEST_FILE_LIST ||
        _msgID == REQUEST_FILE ||
        _msgID == CLOSE_SESSION);
```

This presentation provides enough information to build a correspondent test case matrix (Appendix 6). If Category-Partition had been applied for this method, it would have given a much larger test suite for the method. And in addition, constraints and combinations between values would have to be identified, which is not an easy task. Hence, the One-by-One Selection Criteria offers an effective way to enumerate test cases when there are well-defined domains for input parameters.

5.2.4 Unit testing

The duplicated test driver hierarchy was used for implementing the test cases. The test driver hierarchy was found to be effective for finding inheritance-related faults. It revealed omissions of virtual declarations in the polymorphic superclass and omissions of methods that should have been defined in the subclass (missing override). This is due to the fact that the test cases of the superclass are used in the context of the subclass using dynamic binding. And since tests were automated, tests could be easily re-run in the context of the subclass (CodedClientConnection) revealing these inheritance-related errors.

Testing of the message sequences within the class ClientConnection did not find any state-related faults. However, two explanations can be given for this. Firstly, the code generation from the statechart diagram obviously causes fewer bugs, and secondly, the state space of the ClientConnection was simple. Thus, it can be argued that when the state spaces of classes become more complex, testing of message sequences within classes are more important, and effective as well.

The strength (and the flaw) of Category-Partition is that it is a heuristic approach. For the example system, there was a method which was tested using Category-Partition. It turned out that achieving any coverage metrics would not have revealed an error, but instead, selecting a test case using Category-Partition uncovered an error.

5.2.5 Integration testing

The file server subsystem was tested using the Top-down integration strategy. Since the next lower level of the class under test had to be stubbed, the stub development caused the most significant cost of the integration. The stub development is not reasonable for some classes, since developing a stub causes a load that is equally great as in constructing a real class. A reasonable approach, without a tool support for stub development, is to postpone testing classes in isolation until the server class is constructed. The negative side of this approach is that it makes error tracking complex, and if testing is postponed on many levels it may be difficult to cause desired conditions to happen.

The strength of the Top-down integration was that in every stage in the process, there was a working system (the server, in our case). But when the testing proceeded to lower levels, attaining the high coverage for components under test became difficult. Thus, some classes which lie at the bottom of the dependency tree, were forced to be tested in isolation using a separate driver.

The file client was tested in reversed order compared to the file server, i.e. in bottom to up order. This diminished the effort to implement the stubs, because the classes were tested with the real classes, not with the stubs as it was done in Top-down integration. However, as the integration increased the distance between the tested classes and the classes being integrated, it was difficult to force the already tested classes to cause desired conditions. Hence, some stubs were needed.

The driver implementation caused the most significant cost of the Bottom-up integration. And as the driver cannot be re-used, it is tempting to postpone testing. But once again, the debugging is laborious because you do not know whether it is the class that was recently developed that caused the fault or the class whose testing was postponed.

Another flaw in Bottom-up integration is that if the interfaces of the low-level components are not stable and complete, testing of these components thoroughly before the interface is made robust, is useless. For instance, when the upper level is developed and changes are applied to the interfaces below (because of bad design), testing of the low-level components would have been futile as new test cases had been developed due to the changes in the interfaces.

Circular dependency also caused a testing problem. It forced the classes to be integrated in groups, making effective unit testing harder. Alternatively, stubs could be used to enable isolation testing.

5.3 Conclusions

In order to catch the errors in an early state of the software development and achieve extensive testing, one has to focus on one class at a time. However, in most cases testing classes in isolation is impractical, as both driver and stub development cause an overload that is just not acceptable. The solution for class testing is to interleave class and integration testing and test classes within a cluster. On the other hand, a thorough testing of class level functionality for larger components is a complex task as well, since some stubs and drivers are still needed. Moreover, interleaving integration and class testing makes class testing even harder, as one does not only focus on class level faults, but integration faults between classes have to be addressed as well. As a result, there is no single way to perform class and integration testing, and thus, every object-oriented system requires a careful judgement of how the testing is performed.

Testing cannot be accomplished effectively without a proper software design. There have to be design documents to provide inputs for test design. For object-orientation,

the Unified Modelling Language provides effective means for this. In addition to finding faults in the implementation, test design is also effective in finding bugs in software design. The extended use cases are the most concrete example of this, as they do not only provide a systematic way to develop test cases, but they equally make test cases more complete and consistent.

Although the UML models give useful information for test design, one significant limitation is nevertheless recognised. In most cases, the UML models are nontestable, and the testing personnel have to turn nontestable models into testable ones. To make use case diagrams more testable, extended use cases must be developed. For sequence diagrams, different control flow paths must be recognised. And for statechart diagrams, flattened views have to be constructed. A better approach is that they are made testable during the design. But in the case of statechart diagrams, orthogonal states and substates provide a compact description of the behaviour of the classes. This is contrary to the flattened view, which may lead into spaghetti and blurred diagrams, when the number of states and transitions increase. Hence, this is the trade-off between design and testing that one has to accept.

Some object-oriented design tools, such as Rhapsody, contribute their own impact to test design and testing. For instance, test case development from a sequence diagram is an easy task, since one test case is derived from each diagram. But then again, depicting all scenarios requires a lot of work. This makes sequence diagrams less useful for test design, as it may be that all the collaborations are not depicted. On the other hand, sequence diagrams are useful in the test case execution phase in Rhapsody, since they can be used to capture the system trace, and furthermore, the actual trace can be compared to a specification.

Code generation from a statechart diagram eases the test case development. One tests all the transitions shown explicitly in a diagram to execute action statements and discard testing every message in every state, as one can rely on the fact that the state machine is implemented correctly by Rhapsody. With code generation, there also rises an interesting issue of testing statechart diagrams, since effective testing relies on black box testing, and in Rhapsody, deriving test cases from a statechart partly signifies white box testing because of code generation. And if a statechart is tested too thoroughly, one may end up testing the Rhapsody's code generation. This is not what is wanted. Alternatively, we should redirect the effort that we spare using code generation towards the validation of the model.

Table 9 outlines how the different UML models can be used at the various levels of testing. These models are chosen according to what are supported by Rhapsody and

what was studied, even though diagrams such as activity, collaboration, component and deployment diagrams are useful for testing purposes.

Table 9. Utilisation of the UML in testing.

Level of testing	Use case diagram	Sequence diagram	Statechart diagram	Class diagram
Class			Primary[1]	
Integration	Top-down Collaboration[2]	Top-down Collaboration[3]	Top-down Bottom-up[4]	Integration order and associations[5]
System	Primary[6]	Secondary[7]	Secondary[8]	

1. Message sequences can be derived from the model. If all-roundtrip coverage is required, the diagram should be flattened.
2. Although a use case diagram does not show interfaces of components being integrated, the test cases can be developed using extended use cases.
3. Useful if the test cases, derived from the model, exercise interfaces to be tested.
4. Class scope testing may interleave integration testing among classes and class level testing.
5. Dependency tree can be developed from the model. Furthermore, valid associations can be derived from the diagram.
6. Extended use cases.
7. Useful if all the scenarios of a use case to be tested are depicted.
8. Also system behaviour can be modelled as a state machine.

As it can be seen from Table 9 there is not much use of the UML diagrams at the lowest level of testing (i.e. method testing), but techniques, such as Category-Partition and One-by-One Selection Criteria with the object-oriented aspect, are used. And although the UML models are nontestable and require a definition of additional information in some cases, the UML gives a source of information to software testing that should not be overlooked.

6. Summary

The purpose of this paper was to find out how object-orientation affects testing – especially test design – as well as how the Unified Modelling Language can be used for test design purposes. Furthermore, detailed instructions to design tests were given, and practical solutions to avoid the obstacles of the testing of the object-oriented software were presented as well. Some of the testing techniques, which were studied, were applied to a demonstration system. The system was designed and implemented by using a CASE tool called Rhapsody, as it provides its own impact on test design and testing.

It was argued that testing is a complex, challenging and time-consuming part of a software project. And this is still true with the object-oriented world: object-orientation does not make faults disappear. On the contrary, it provides some unique errors and faults that cannot be found from traditional languages and occasionally makes testing harder. Inheritance, integration to more extent, polymorphism, dynamic binding, and the increased number of interfaces make the testing of object-oriented systems more difficult compared to the traditional systems.

Encapsulation also presents an obstacle to testing. It hides the information necessary to testing, since the information of the classes is only available through the interfaces. However, using a driver that has access to protected and private features of CUT neatly avoids this obstacle. In C++, this problem is avoided using a concept of friend class. The positive side of encapsulation is that it prevents bugs common to traditional languages, and by increasing the modularity, testing can be limited.

The lowest level of testing in the object-oriented systems tends to be a simple task, because the methods are small and simple, and the coverage of these methods can be easily obtained. Although this is an important strength of object-orientation, as it provides means to break down large and complex systems into smaller ones, the complexity has not disappeared: it is only pushed to interfaces.

The black box approach is the primary strategy to develop test cases for object-oriented software, and there are numbers of traditional techniques that can be used for object-oriented systems. In contrast to black box testing, the white box approach is less applicable for the object-oriented systems. Because of the fact that there are increased numbers of interfaces and because objects communicate with messages, messages and their contents should somehow be recorded. Rhapsody provides means for this. In the testing phase, a sequence diagram can be used to capture the system trace, and furthermore, the actual trace can be compared to a specification.

In this paper, three levels of software testing were introduced: class testing, integration testing, and system testing. Class testing was mapped to the traditional definition of unit testing. In object-orientation at the class level, one does not only have to test methods and interactions of these methods, but inherited features should equally be tested in the new context. Moreover, it should be tested that the subclass is substitutable for the superclass (i.e. a subclass is a type of superclass). However, the effort of re-testing can be diminished by using the test cases of the superclass. The duplicated driver hierarchy provides a capability for this.

Object-oriented integration testing concentrates on testing components' interfaces as it does in the traditional software. However, Top-down and Bottom-up interleaves integration testing and class testing, which makes class testing even harder, as one does not only focus on the class level faults, but also integration faults between classes have to be addressed. The testing of the object-oriented system is still much the same when non-functional capabilities of the system are concerned. On the other hand, use cases provide a new object-oriented view for testing functional capabilities of the system.

Effective testing cannot be accomplished without a proper software design. There must be complete, stable and robust design documents before the test design may begin. Preferably, the testing point of view should be taken into account in the design phase by making the UML models testable. For use cases this works well, but for the statechart diagrams it is not always reasonable to flatten the view, as the flattened view may lead to spaghetti diagrams. And for the sequence diagram, it is reasonable to depict different scenarios in the same model, as it provides a more compact and comprehensive view of the use case.

The UML diagrams can be used at the various levels of testing. At the class scope, statechart diagrams are useful as they show the intended behaviour of the classes, and the valid message sequences can consequently be derived from these diagrams. But in Rhapsody, deriving the test cases from the diagram partly denotes white box testing because of code generation. More importantly, it should be verified that the specification is correct.

At the integration level, the test suite can be developed using sequence diagrams. But then again, using Rhapsody as a modelling tool, the benefit for test design is diminished since sequence diagrams depict only one scenario of a use case. This is not an effective way to test the collaborations of classes that implement a use case. One can also gain benefit from the class diagrams at the integration level, since dependency trees can be developed from these diagrams by inspection. In addition, valid associations can be derived from the model.

Use case diagrams can be used to test the functional capabilities of the system. However, use cases are seldom test-ready and extensions are thereby needed. Extended use cases provide this extra information for system scope testing. At the lowest level of testing (i.e. method testing), there is not much use of the UML diagrams. Instead, techniques such as Category-Partition and One-by-One Selection Criteria with the object-oriented aspect are used.

7. References

[1] Koskimies, K. (1998) Pieni oliokirja. Espoo: Suomen ATK-kustannus Oy. 281 p.

[2] Pressman, R. (1997). Software engineering: a practitioner's approach. New York, NY: McGraw-Hill Companies. 885 p.

[3] Jorgensen, P. C. (1995) Software testing: a craftsman's approach. Boca Raton, FL: CRC Press. 254 p.

[4] Kit, E. (1995) Software testing in the real world: improving the process. Wokingham: Addison-Wesley. 252 p.

[5] Hetzel, W. C. (1988) The complete guide to software testing. Wellesley, MA: QED Information Sciences. 284 p.

[6] Beizer, B. (1990) Software testing techniques. New York, NY: Van Nostrand Reinhold. 550 p.

[7] Pekkola, T. Oliosuuntautuneen ohjelmiston regressiotestaus. Oulu: University of Oulu, Department of information processing science. 58 p. Master thesis.

[8] Binder, B. V. (1999) Testing object-oriented systems: models, patterns, and tools. Reading, MA: Addison Wesley Longman. 1191 p.

[9] Beizer, B. (1995) Black-box testing: techniques for functional testing of software and systems. New York, NY: John Wiley & Sons. 294 p.

[10] Ovum: Software testing tools. Ovum Ltd.

[11] Varvikko, A. (1995) Oliosuuntautuneen testauksen erityispiirteet. OULU: University of Oulu, Department of information processing science. 62 p. Master thesis.

[12] Available: (21.5.2000). Free On-Line Dictionary Of Computing. URL: http://wombat.doc.ic.ac.uk/foldoc/index.html

[13] Available: (20.4.2000). OMG Unified Modeling Language v. 1.3 Specification.
URL: http://www.rational.com/uml/resources/whitepapers/index.jtmpl.

[14] Ostrand, T. & Blacer, M. (1988). The category-partition method for specifying and generating functional tests. Communications of ACM, Vol. 31, p. 676–686.

[15] Jeng, B. & Weyuker, E. (1994). The simplified domain-testing strategy. ACM Transactions on software engineering and methodology, Vol. 3, p. 254–270.

[16] Kaner, C., Falk, J. & Nguyen, H. (1993). Testing computer software. New York, NY:Van Nostrand Reinhold. 480 p.

[17] Kim, H. & Wu, C. (1996). A class testing technique based on data bindings. Los Alamitos, CA: IEEE Computer Society Press. P. 104–109.

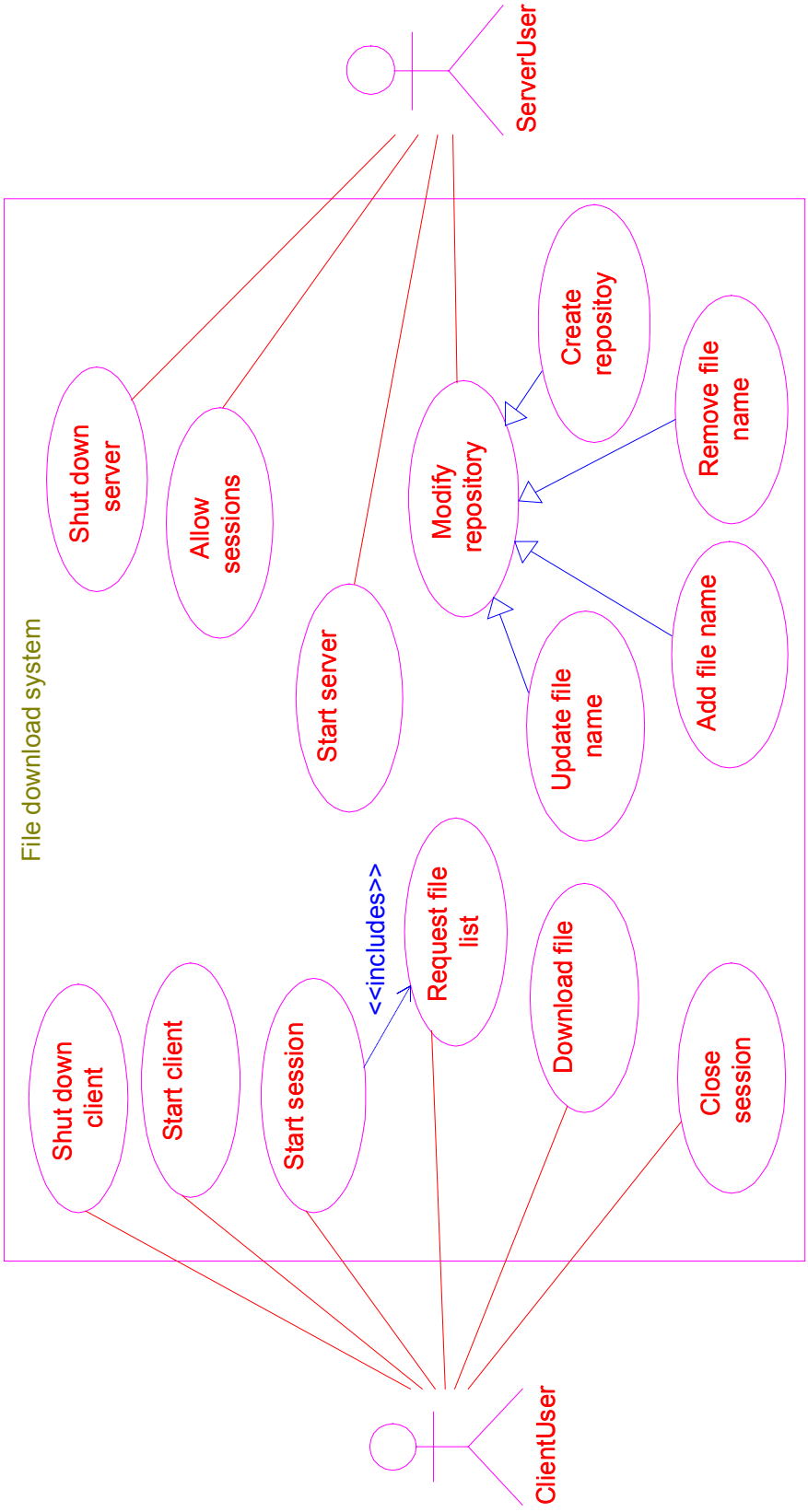
[18] Jorgensen, P. C. & Erickson, C. (1994). Object-oriented integration testing. Communications of ACM, Vol. 37, p. 30–38.

[19] Fewster, M. & Graham, D. (1999). Software test automation: effective use of test execution tools. Harlow: Addison-Wesley. 574 p.

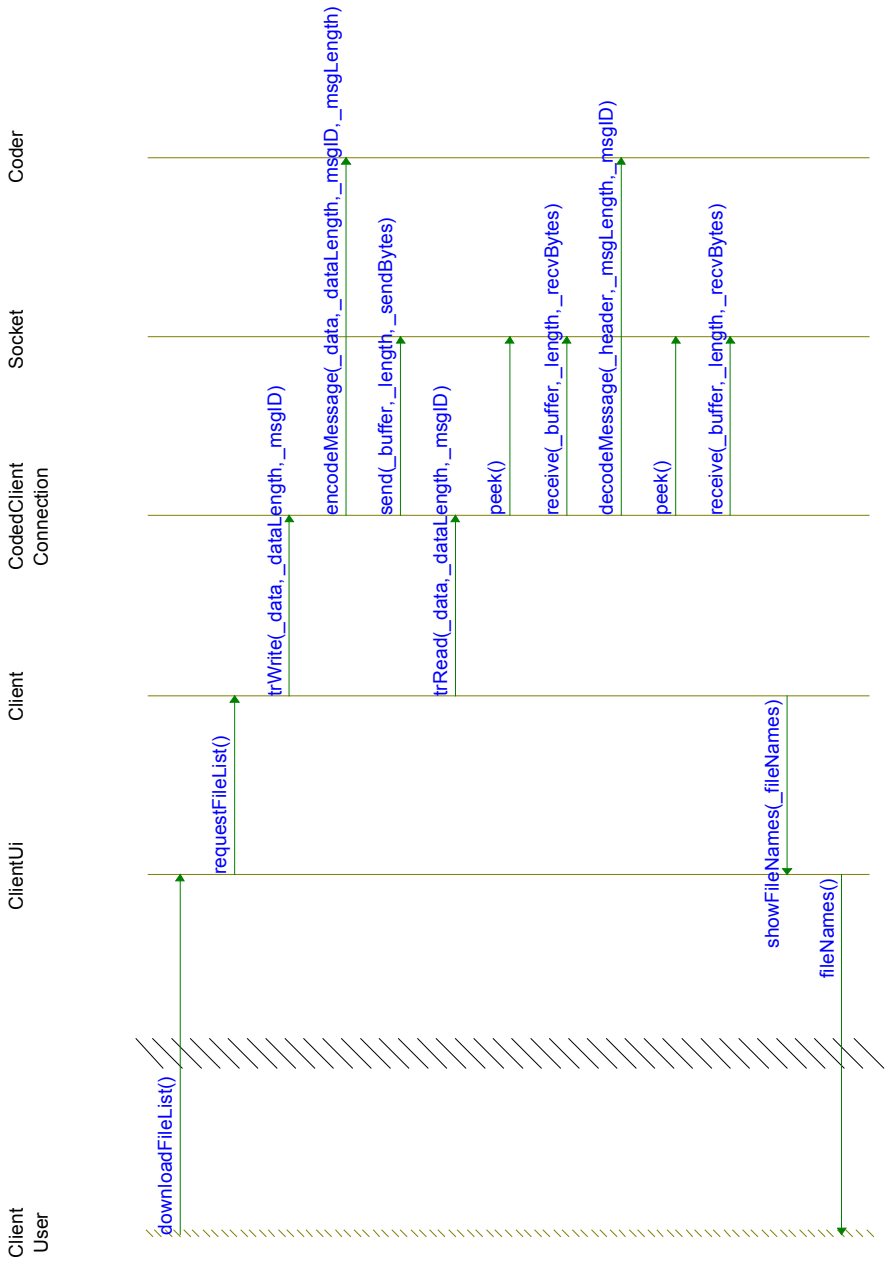
[20] Firesmith, D. G. (1995). Object-oriented regression testing. Report on Object Analysis and Design 5. P. 42–45.

[21] Douglass, B. P. (1999) Real-time UML: developing efficient objects for embedded systems. Reading, MA: Addison Wesley Longman. 328 p.

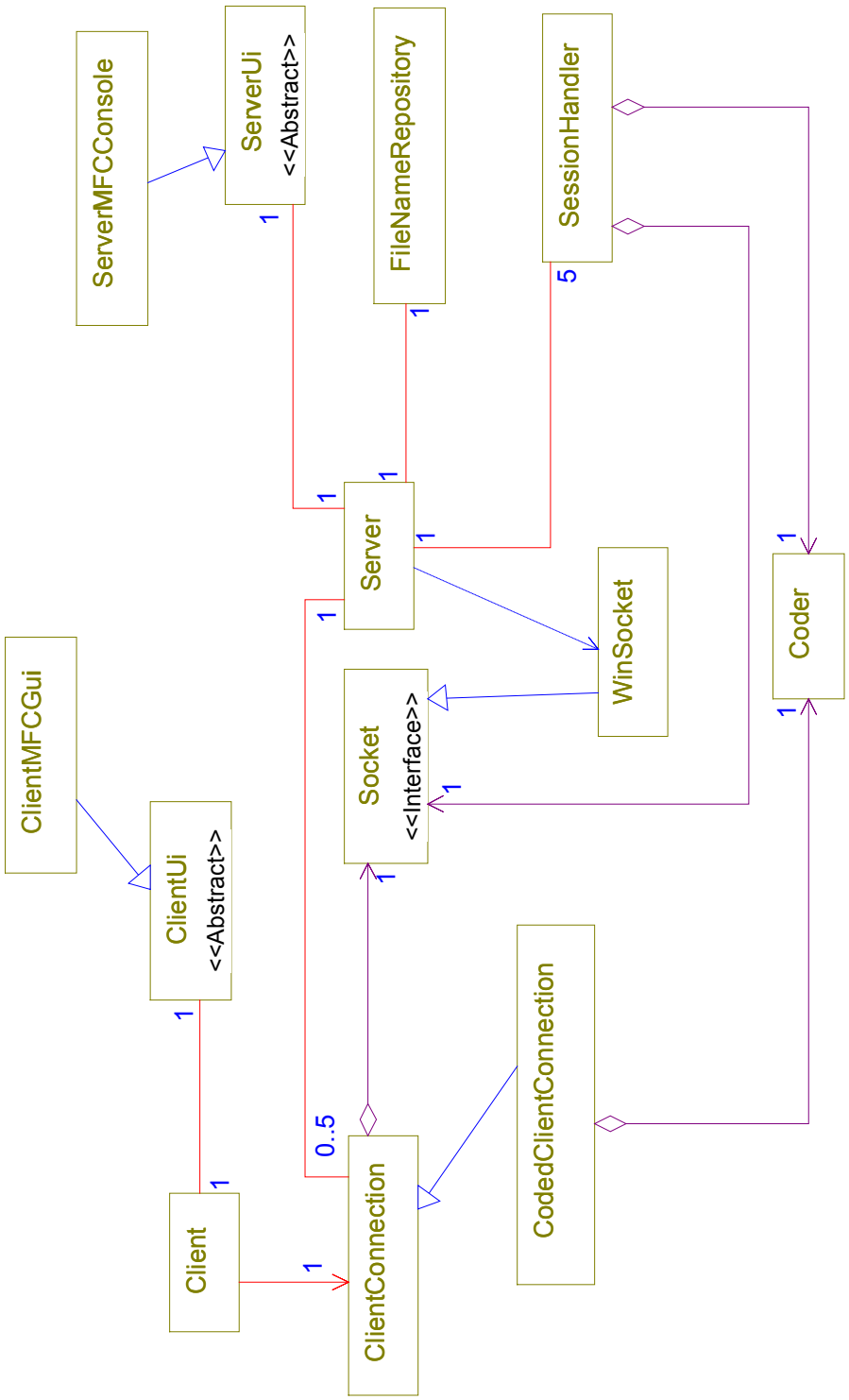
Appendix 1
Use case diagram of file download system



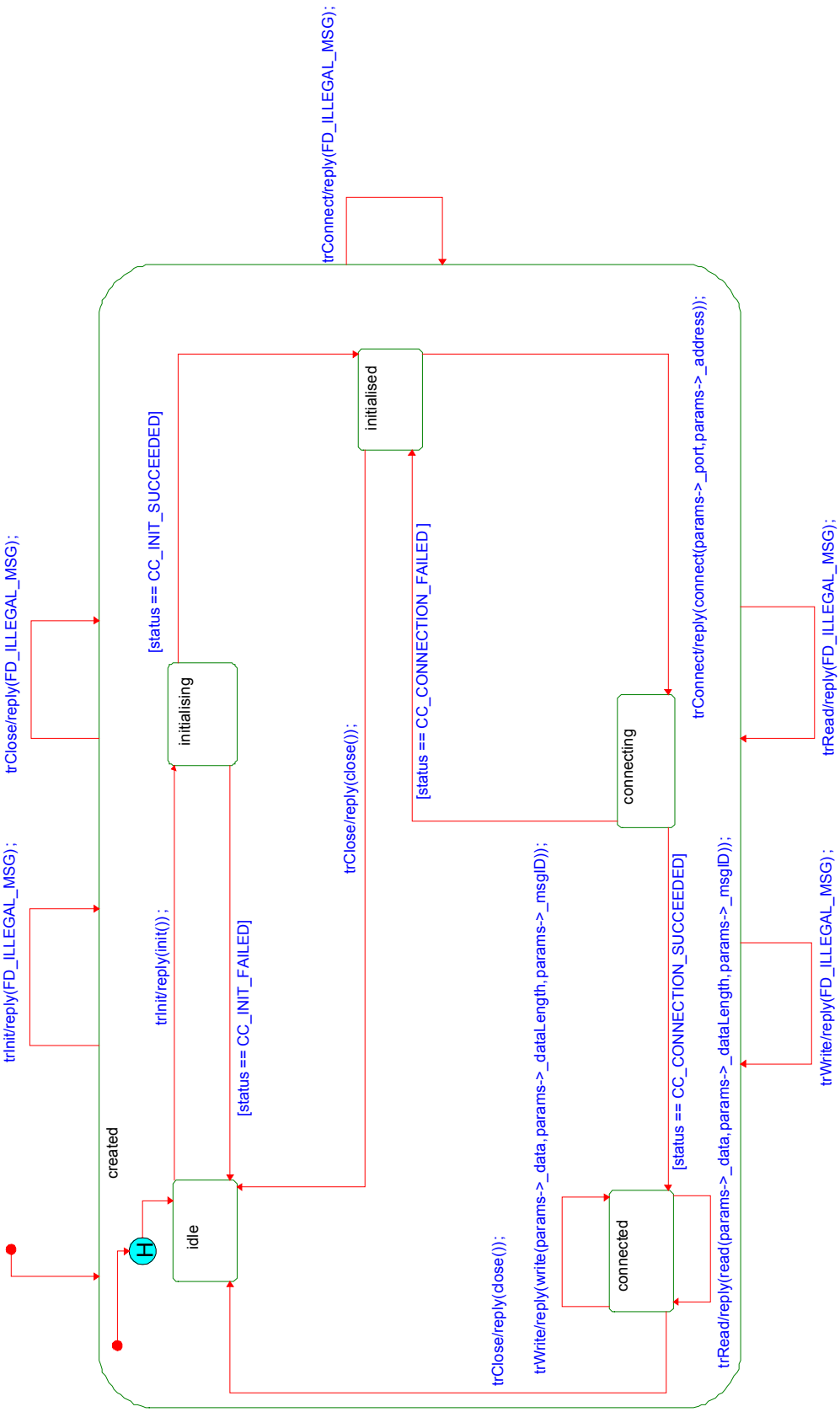
Appendix 2
Sequence diagram of *request file list* use case



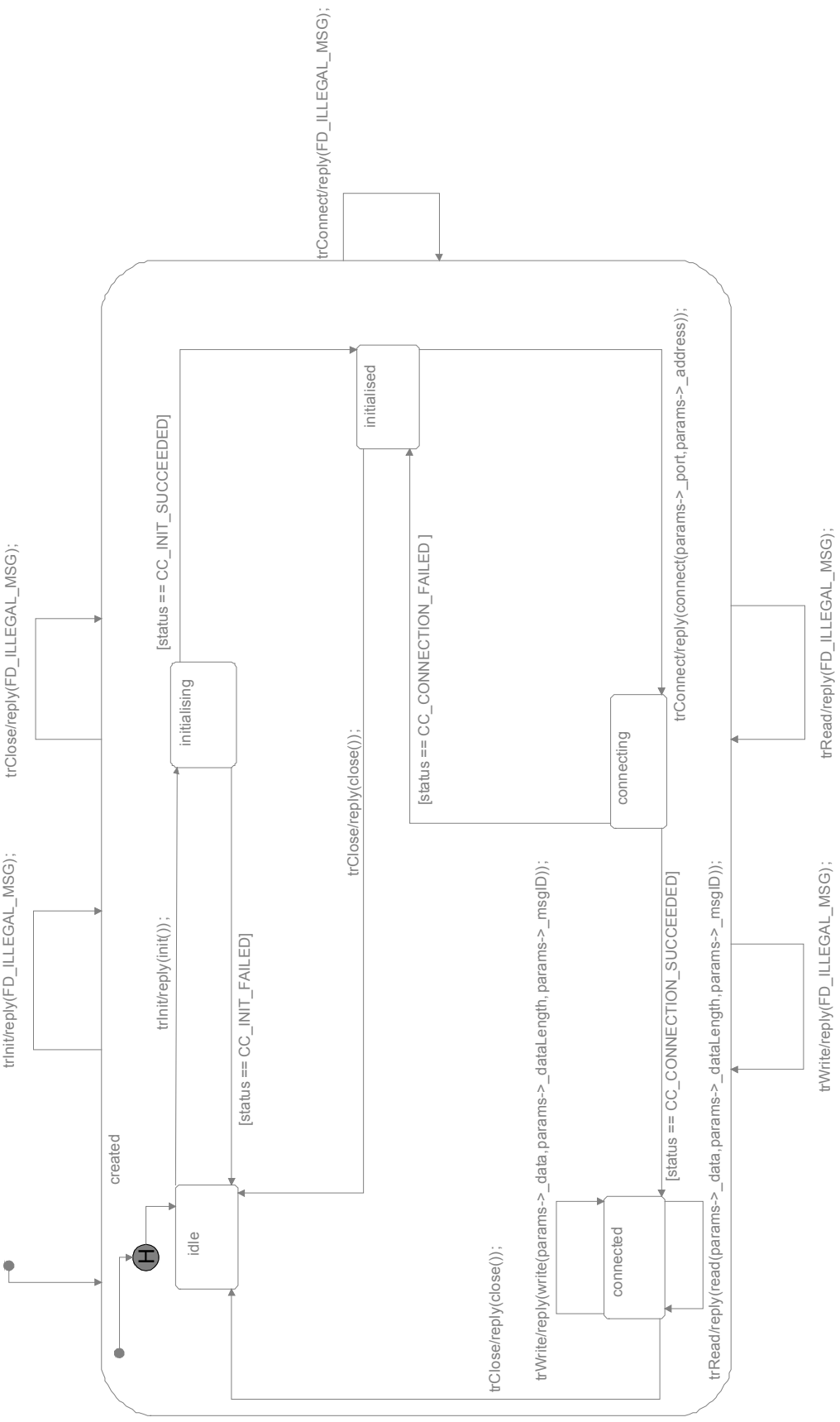
Appendix 3
Class diagram of file download system



Appendix 4
Statechart diagram of ClientConnection



Appendix 5
Statechart diagram of CodedClientConnection



Appendix 6
Test case matrix for Coder::encodeMessage

Input			Test cases							
Variable	Condition	Type	1	2	3	4	5	6	7	8
_dataLength	<=MAX_DATA_LENGTH	On	MAX_DATA_LENGTH							
		Off		MAX_DATA_LENGTH+1						
	=>MIN_DATA_LENGTH	On			MIN_DATA_LENGTH					
		Off				MIN_DATA_LENGTH -1				
		Nom					10	5000	0.5M	6M
_msgID	>=VERIFY_CONNECTION	On					VERIFY_CONNECTION			
		Off						VERIFY_CONNECTION-1		
	<=CLOSE_SESSION	On							CLOSE_SESSION	
		Off								CLOSE_SESSION+1
		Nom	REQUEST_FILE	VERIFY_CONNECTION	REQUEST_FILE_LIST	CLOSE_SESSION				
_data		Nom	all \0s	all Qs	all Ps	all Gs	all \ns	all Ds	all %s	all 1s
Accept value			Yes	No	Yes	No	Yes	No	Yes	No

Published by

Vuorimiehentie 5, P.O.Box 2000, FIN-02044 VTT, Finland
Phone internat. +358 9 4561
Fax +358 9 456 4374

Series title, number and report code of publication

VTT Research Notes 2092
VTT–TIED–2092

Author(s)
Kärki; Matti

Title

Testing of Object-Oriented Software
Utilisation of the UML in testing

Abstract

The modern software development requires more efficient production methods than ever before. It has been recognised that benefits can be obtained in software development by using object-orientation. Testing, however, has gained less attention, although it is still an important task in the software development to achieve such goals as finding errors and quality. The goal of this paper is to study how object-orientation affects testing as well as how the testing techniques that are adapted for object-orientation can be used for test design purposes. Utilisation of the Unified Modelling Language (UML) in testing is introduced, and some practical solutions to avoid the obstacles of the testing of object-oriented software are addressed as well. Moreover, these solutions are combined and a test automation system (test driver implementation), which makes it easier to test the object-oriented software, is presented.

Finally, the testing techniques that are studied, are applied to a demonstration system, which is designed and implemented by using a CASE tool called Rhapsody. As Rhapsody provides its own impact to testing and test design, it is shown how the various UML diagrams are used for test design purposes in the context of Rhapsody.

Although object-orientation provides benefits for software development, it can be argued that the testing of object-oriented systems is occasionally more difficult compared to the testing of traditional systems. However, by planning tests carefully and taking the special needs of the testing of object-oriented software into account, these obstacles can partially be avoided. Furthermore, since the UML provides a notation to express software designs, and as object-orientation emphasises functional testing, the UML gives information for test design that should not be overlooked.

Keywords
Unifield Modelling Language, software development, software test processes, software test automation

Activity unit
VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland

ISBN			Project number
951–38–5816–2 (soft back ed.) 951–38–5817–0 (URL:http://www.inf.vtt.fi/pdf)			
Date May 2001	Language English	Pages 69 p. + app. 6 p.	Price B
Name of project ASCENT-project		Commissioned by Kaski Tech, National Technology Agency (Tekes), Nokia Mobile Phones, Nokia Networks, VTT Electronics	
Series title and ISSN VTT Tiedotteita – Meddelanden – Research Notes 1235–0605 (soft back ed.) 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN-02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	