TEST DRIVEN DEVELOPMENT OF EMBEDDED SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUSTAFA İSPİR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

NOVEMBER 2004

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Canan ÖZGEN
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. İsmet ERKMEN
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Semih BİLGEN
Supervisor

Examining Committee Members

Prof. Dr. Hasan GÜRAN	(METU, EE)	
Prof. Dr. Semih BİLGEN	(METU, EE)	
Prof. Dr. Uğur HALICI	(METU, EE)	
Asst. Prof. Dr. Cüneyt BAZLAMAÇCI	(METU, EE)	
Dr. Altan KOÇYİĞİT	(METU, II)	

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name :

Signature :

ABSTRACT

TEST DRIVEN DEVELOPMENT OF EMBEDDED SYSTEMS

İSPİR, Mustafa

M.Sc., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Semih BİLGEN

November 2004, 111 pages

In this thesis, the Test Driven Development method (TDD) is studied for use in developing embedded software. The required framework is written for the development environment Rhapsody.

Integration of TDD into a classical development cycle, without necessitating a transition to agile methodologies of software development and required unit test framework to apply TDD to an object oriented embedded software development project with a specific development environment and specific project conditions are done in this thesis. A software tool for unit testing is developed specifically for this purpose, both to support the proposed approach and to illustrate its application.

The results show that RhapUnit supplies the required testing functionality for developing embedded software in Rhapsody with TDD. Also, development of RhapUnit is a successful example of the application of TDD.

Keywords: Test Driven Development, TDD, Embedded Software, Agile Methodology

ÖZ

SINAMAYA DAYALI YÖNTEMLE GÖMÜLÜ YAZILIM GELİŞTİRME

İSPİR, Mustafa

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Danışmanı: Prof. Dr. Semih BİLGEN

Kasım 2004, 111 sayfa

Bu tezde, sınamaya dayalı yazılım geliştirme yöntemi gömülü sistemlerde kullanılmak için incelenmiş ve Rhapsody geliştirme ortamında sınamaya dayalı yazılım geliştirme yöntemini kullanmak için gerekli altyapı yazılmıştır.

Bu tez kapsamında sınamaya dayalı yazılım geliştirme yönteminin çevik bir yazılım yöntemine geçişi mecbur kılmadan uygulanabilmesi için bir süreç oluşturulmuştur. Bu yöntemin uygulanabilmesi için gerekli olan birim testi altyapısı, somut bir uygulama örneği de olmak üzere hazırlanmıştır.

Elde edilen sonuçlar RhapUnit'in Rhapsody ortamında TDD yöntemiyle yazılım geliştirmek için gerekli yetenekleri sağladığını göstermektedir. Bunun yanısıra, RhapUnit'in geliştirilmesi başarılı bir TDD çalışması olmuştur.

Anahtar Kelimeler: Sınamaya Dayalı Yazılım Geliştirme, Çevik Süreç, Gömülü Yazılım

ACKNOWLEDGMENTS

I express very sincere appreciation to Prof. Dr. Semih Bilgen for his valuable supervision throughout the research. With his precious ideas that he has shared with me at every phase of the study, this thesis could be completed.

I wish to thank my chief engineer Özgü Özköse Erdoğan and my department manager Levent Alkışlar on behalf of ASELSAN Inc. for their encouragement and support.

I wish to thank my colleague Ali Özzeybek for introducing me with the Kent Beck's book on TDD and for discussing patiently on TDD during this work.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Requirements and technologies change rapidly and market pressure on rapid development leads to new methodologies in order to develop reliable, maintainable and useful software in a short period of time. These methodologies are called “agile”. One of the core practices of agile methodologies is Test Driven Development (TDD).

“Clean code that works” [1] is the main goal of TDD. TDD is a predictable way of developing software. The code you write is always in a working state, so you don’t give any chance to get a code that is full of bugs. Also, you learn all lessons that the code has to teach you at the early stages. Behind the quality effects, TDD also makes developers’ lives easier by protecting them from big surprises and by making developers’ code trustable.

In this thesis it is explained how TDD ensures these qualities. Firstly an interpretation of TDD is given in chapter 2. After that the integration of TDD with a classical methodology is described in chapter 3. The main concern of this integration is changing the way of coding according to TDD without affecting the contractual and interdepartmental issues. Also, the properties of required unit test framework are given in chapter 3. Also a unit test framework is written within the scope of this thesis. This framework is described in chapter 4. Development of this framework itself has been an application of TDD. RhapUnit is an automated unit test framework. It works both on development environment and on target environment. So that unit tests can be executed on both environment. Chapter 5 concludes the thesis.

Throughout this study, we use the term “classical method” to indicate any software development approach which designs and implements unit tests after, rather than before, software development. We use the term “unit test” to indicate gray-box testing of a method of a class.

CHAPTER 2

TEST DRIVEN DEVELOPMENT (TDD)

In this chapter, firstly an overview and meaning of TDD is given. After that, the main TDD cycle is introduced. Some patterns of TDD are given. After stating what is TDD and what are its best practices, to concretize TDD, the interpretation of an example is given. Then the challenges of TDD are explained.

2.1 TDD Overview

K. Beck is the one of the creators of eXtreme Programming (XP) and he also introduced TDD [1]. His fundamental premise is: "To achieve clean code that works, drive the development by automated tests" which is called Test-Driven Development. Mainly there are two simple rules in TDD:

- Write new code only if an automated test has failed
- Eliminate duplication

To apply these two rules following sequential steps are used in development [2]:

- Choose a defect to fix or an area of the design or task requirements to best drive the development.
- Design a concrete test that is as simple as possible while driving the development as required, and check whether the test fails.
- Alter the system to satisfy the new test without affecting the other tests results.
- Possibly refactor the system to remove redundancy, without breaking any tests
- Go to first step.

Explanations of the fundamental terms comprising TDD are:

<u>Development</u>: Classical method of development is “phasist”, so it is weak mainly because of long feedback time between decisions and their results. In TDD, development means a “complex dance” of these phases: analysis design, detailed design, implementation, unit test.

<u>Driven</u>: At the beginning the name of TDD is “test-first programming”. But in this methodology test is not only the first thing, it is also the driver of the development. And an impressive question that stresses the importance of TDD is that “If you don’t drive development with tests, what do you drive it with? Speculation? Specification? (Ever notice that those two words come from the same root?)”.

<u>Test</u>: Automated, fast, full, isolated unit tests. Testing is technique for structuring all the activities.

In this chapter, first, main TDD cycle is introduced. Effects of each phase of TDD are interpreted. After that some patterns of TDD are explained. These patterns can be thought as development habits. Then, for the sake of concretization, an interpretation of an example is given. Finally the challenges and environmental requirements of TDD are interpreted.

2.1.1 Main TDD cycle

TDD gives us a working style composed of three sequential and cyclic tasks to apply two main rules. This cycle may be repeated more than once in an hour. Detailed explanations of the three TDD steps, "red", "green" and "refactor", are given below:

1. Red: Select a functionality to add to software or a defect to fix and write a little test that doesn't work, and perhaps doesn't even compile at first. This test is used to show whether the functionality is added successfully or not. Writing tests before writing code has the following advantages:

   - Implies thinking on what is required before how it is implemented. So the developers' understanding of requirements becomes clear in early stages.
   - Forces any doubts about the requirements to be resolved while writing the test cases [3].
   - Supplies the tool for a developer to understand whether he/she achieves his/her goal or not.
   - Implies highly cohesive and low coupled design, by writing easily testable code. Testability requirements [4] that effect the design of a unit are the followings:

    - o Setability: The extent to which an external test driver initialize the data.
    - o Controllability: The extent to which an external test driver can direct the flow of control
    - o Visibility: The extent to which an external test driver can check the results.
  - Prevents writing insufficient unit tests. Experiments have shown that developers tend to write loose unit test, if they have already written and run the code [5].
  - Accumulates the test experiences. Adding test cases before fixing a bug leads to decreasing bugs in new releases. According to the author of this thesis, by this way, all experiences gained from integration tests, system tests and run-time user bug reports are accumulated in the unit tests. In pre-TDD development cycle, software tends to be more error prone, but by TDD it becomes more reliable and error-free. For complex, large and long-termed projects, this feature may be the most important advantage.

2. Green: Implement the code that is only enough to make tests succeeded. This stage helps to:
  - Think on how new functionality is added. Prevents loosing focus between making good design and implementing functionality.
  - Prevent speculative designs and codes, since the main goal is making test results successful. Especially, generating speculative code for the concern of optimization and generalization can be prevented [6].
  - Quickly find if other functionalities are effected from test results

- Concentrate only on what is needed and not more.

3. Refactor: Refactoring means improving the software design without affecting its functionality. The main goal of refactoring in TDD is eliminating all of the duplication. Finally this stage helps to:

    - Think on good design. In this step, developers only focus on good design.
    - Keep the code clean.
    - Improve design to adapt new functionalities.
    - Make design decisions exactly when they are needed and not before.

The TDD cycle is shown in Figure 1. Arrows show the flow of actions. As it is seen, the initial action is choosing a task, second one is writing tests, third one is implementing the solution and the last action is refactoring the code. Story is a form of software requirements in XP. Defect is a software bug that corrupts functionality. Both implementing functionality and fixing a debug require changing the code. So they are handled in the same way in TDD. In this figure the TDD cycle is separated according to status of tests whether they all pass or not. As it is seen, if there are some failed tests, the code should not be refactored or a new task should not be selected.

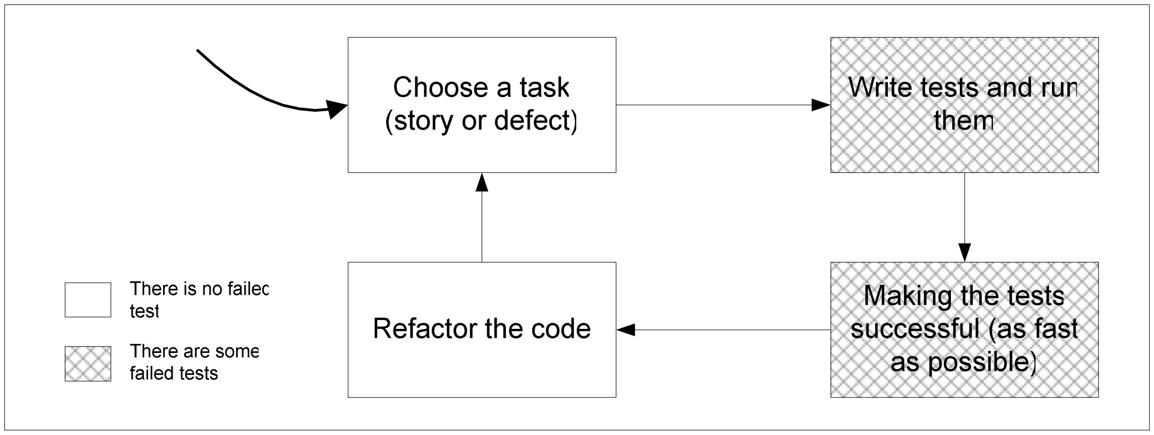

Choose a task (story or defect)
Write tests and run them
Making the tests successful (as fast as possible)
Refactor the code
There is no failed test
There are some failed tests

Figure 1: TDD cycle.

TDD is also a positive system, meaning that by applying TDD the conditions of the project become more suitable to apply TDD. To show how TDD is a self supportive activity, relations between stress, making tests and finding errors are given on an influence diagram, in Figure 2. Normally, development activities start without the stress of time, but as time elapses, time becomes the most recognizable pressure. If the classic methods are followed, developers firstly write their code as a whole. This tends to decrease the time left to tests. Therefore errors tend to increase and so this increases the stress. This is the nightmare vicious circle of the developer. TDD breaks this cycle by writing unit tests of a small functionality, before implementing that functionality. This tends to decrease errors. So stress decreases as well. This means it is possible to do more tests. As it is seen this is the virtuous cycle of TDD.

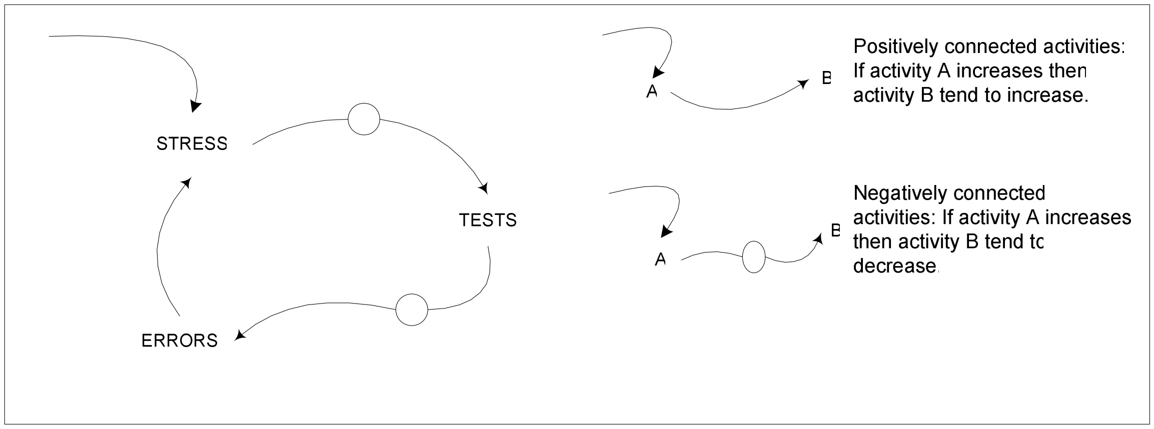

STRESS
TESTS
ERRORS
A
B
Positively connected activities: If activity A increases then activity B tend to increase.
A
B
Negatively connected activities: If activity A increases then activity B tend to decrease.

Figure 2: Influence diagram of software development [1].

TDD does not capture the whole life cycle of a software project. How TDD is integrated to the life cycle depends on the project and company conditions. For example the development cycle of the one of the software development group of the IBM Retail Store Solutions [7] is shown in Figure 3. As it is seen, not all of the unit tests are automated. Some of the unit tests may require human interaction. Especially protocol, hardware and graphical interface layers' classes may require some human interaction. Nevertheless, most of the tests (86%) are automated. Since the specifications are mature, this group has selected up-front design but test-first code strategy. In other words, work flow is started with detailed design, unit tests are written and finally the design is implemented. So automated, mature, effective unit testing and easy refactoring are the features of the TDD used by this software group. It should be noted, however, that, imposing a good and simple design on the software, one of the most powerful features of TDD, is not used in this project.

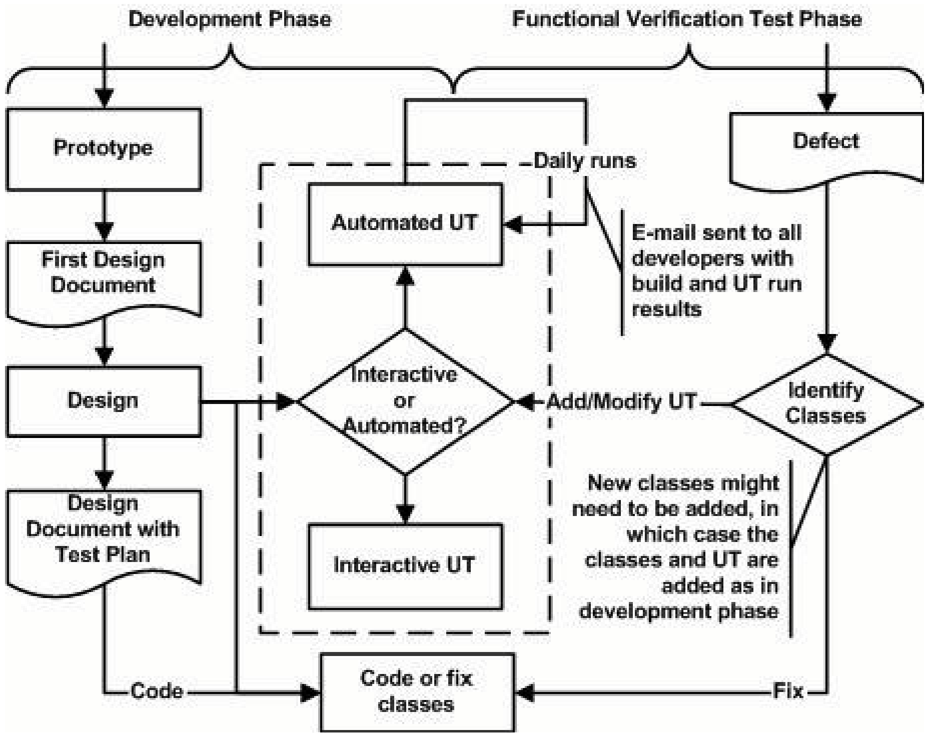

Development Phase
Functional Verification Test Phase
Prototype
First Design Document
Design
Design Document with Test Plan
Daily runs
Automated UT
Interactive or Automated?
Interactive UT
Defect
E-mail sent to all developers with build and UT run results
Add/Modify UT
Identify Classes
New classes might need to be added, in which case the classes and UT are added as in development phase
Code
Code or fix classes
Fix

Figure 3: Development life cycle that is used by the one of the software group of IBM Retail Store Solutions [7]

2.1.2 Some good practices (patterns) of TDD

Some good practices [1] of TDD are reviewed in the following paragraphs. These practices are not related only with how good tests are written but also how a developer can work effectively. Test List, Break, Broken Tests are some example of patterns that are related with working effectively. These patterns can be considered as good habits of a developer.

Tests: Automate the tests and run them as often as possible.

Isolated Tests: Result of a single test case should not affect the other test cases.

Test List: Write a list of all the tests that you know you will have to write. And if any new test comes to mind then add it to the list.

Assert First: As adding a new functionality is started with writing test case of that functionality, writing a test case is started with writing the assert statements.

Evident Data: Include expected and actual results in the test itself, and try to make their relationship apparent.

One Step Test: Pick a test that will teach you something and that you are confident you can implement to write.

Explanation Test: Give explanation of some functionality or requirement in terms of tests. By applying this pattern, requirements are intrinsically become testable.

Learning Test: Write test to outsourced software (library…) to show that it works as you expected.

Regression Test: If a defect is reported, first write a possible minimum test that fails because of that defect. Then go on with TDD.

Break: If you know what to type, type the obvious implementation. If you don't know what to type, then fake it. If the right design still isn't clear, then triangulate. If you still don't know what to type, then you can take break. Taking regular breaks are also a pattern of XP and known as Work Fresh.

Do Over: If you feel you are lost, then throw away the code and start over.

Child Test: If one of the test cases tends to be too big, then write smaller test cases which represent the big test case and delete the big one.

Mock Object: To test an object that depends on other resources, create fake versions of those resources to make isolated test of that object.

Broken Test: When leaving a programming session, write a test case and leave it broken. So you know exactly where you are.

Fake It: If you don't know the Obvious Implementation, just write the fake solution to make test success. After that you may find the solution by changing the constants with variables.

Triangulate: If there is a doubt about implementation, run test case with different parameters.

One to Many: If an operation works on a collection, first implement it for only one element, and then make it work with collection.

2.2 Interpretation of an Example

To concretize how TDD works, below, we shall compare the classic way of coding with TDD on a little program in Java that calculates the prime factors of an integer. The code and iteration cycles of this example is given at the appendix. This example is taken from R. C. Martin [8].

By the help of TDD:

- Write just what is needed: look at the prime number generator dependency example.

- Write as simple as possible: look at the needed link list in first way in the example.
- For all small steps you have a program that works and can be tested automatically.
- Don’t hesitate to refactor: Since tests are automated and always synchronized with the software (i.e. tests cover all functionality), the code can be refactored anytime. This is maybe one of the best features of TDD.

But since it is a simple example some of the features of TDD can not be shown, such as:

- Tests are a form of documentation. Compilable, executable, and always in synchronized with the software. Any one can easily learn how a class is used. Pre-conditions and post-conditions of an operation are easily seen in unit tests.
- TDD implies the more abstract and low coupled designs.
- TDD implies thinking of what is needed before how it is implemented: If there is an ambiguity on what is needed it appears immediately.
- TDD increases the reliability. Founded defects are also added to the unit tests, so as time goes, software become more reliable.

2.3 Challenges of TDD

Applying TDD has some technical implications. These are [1]:

- Developers must write their own tests, since tests are run more than once in an hour; there isn’t any time to wait for someone else to write tests.

- Development environment must provide rapid response to small changes.
- Designs must consist of many highly cohesive, loosely coupled components, just to make testing easy.

These implications and current development habits lead to some challenges. These challenges are:

- If there isn't an appropriate tool support, writing automated unit test can become a critical time consuming activity. This can cause the developers lose their motivation. Especially for Java, TDD has a very good integrated tool support. For example JUnit [www.junit.org] is integrated with the Eclipse and JBuilder.
- If the test duration isn't sufficiently reduced, developers can lose their motivation of running the tests more than once in an hour. For example if test duration is half an hour, developer wouldn't run the tests for every small change.
- If tests can not start as easy as clicking on a button, developers could lose their motivation of running the tests more than once in an hour. For example for embedded systems if unit tests require running on the target, developers could not run unit tests as frequently as TDD require.
- It is hard to change the developer's habits to use TDD. Before applying TDD developers must have a good understanding of unit tests, object oriented concepts and philosophy of simple design [9]. Also developers tend to see the TDD programming as a test technique, so they can not take the all advantages of the TDD.

Also, experiments [5, 7, 10] have shown that when applying TDD, the first build of the software requires more time than applying the classical development method. This may lead to decrease the management support for applying TDD.

2.4 Conclusion

Reducing defects, shortening the feedback loop on design decisions, continuously refactoring, implying thinking what is needed before how it is implemented and accumulating test experiences are the key advantages of TDD. Reducing defect not only decrease the maintenance cost, but also have some positive psychological and social effects such as trust of colleagues to each other. By shortening the feedback loop on design decisions, developers don’t spend much time on wrong decisions. In the classical approach, code becomes spaghetti as time elapses. But in TDD, continuous refactoring prevents this and always maintains the code in a state of high cohesion and low coupling. Leading programmers to think on what is needed before how it is implemented prevents superfluous generalization. Finally accumulating the test experiences reduces the cost of integration tests and system tests. This feature makes TDD invaluable for the large, complex and hard to test projects.

CHAPTER 3

APPLYING TDD

Engineering can be considered as applying theory to the real world specific conditions. This chapter explains the main purposes of this thesis: integrate TDD into a classical development cycle, without necessitating a transition to agile methodologies of software development and supply required framework to apply TDD to an object oriented embedded software development project with a specific development environment and specific project conditions. For reasons of privacy, the specific firm in which this exercise has been undertaken shall not be revealed.

3.1 Application Environment

The project which is considered as a basis for applying TDD consists of both hardware and software development in parallel. Software engineers who work in this project belong to one of three groups: system engineers, hardware engineers, and software engineers. System engineers are responsible from the whole product. They are also responsible from the use cases of the project. So from the point of the hardware and software developers, they can be thought as the internal customer of the project.

Software development team is grouped in three parts: high level application developers, control processing developers, and low level developers.

High level application developers are interested in graphical user interfaces, geographical information systems, etc. They develop with an object oriented programming language. Their target platform is servers. Their development platform and target platform are the same.

Low level developers are responsible from implementing algorithms and processing the ingoing data. They develop with a low level programming language such as C. Their target platform is special purpose processors such as DSP. They work on cross-development, i.e. their development platform and target platform are not same.

Control processing developers are responsible from the control processor which can be thought as the mediator [11] of the system. They are responsible from controlling all messaging between software subsystems and controlling all hardware. Each hardware and software subsystem has an interface with the control processor and they have no other interface with other subsystems. The simple system interfaces are shown in Figure 4. They are developed with an object oriented language, C++, and developed to an embedded target with a real time OS such as VxWorks. The main modeling tool used that also generates all code is Rhapsody in C++ by I-Logix Inc. [http://www.ilogix.com/rhapsody/rhapsody_c_plus.cfm].

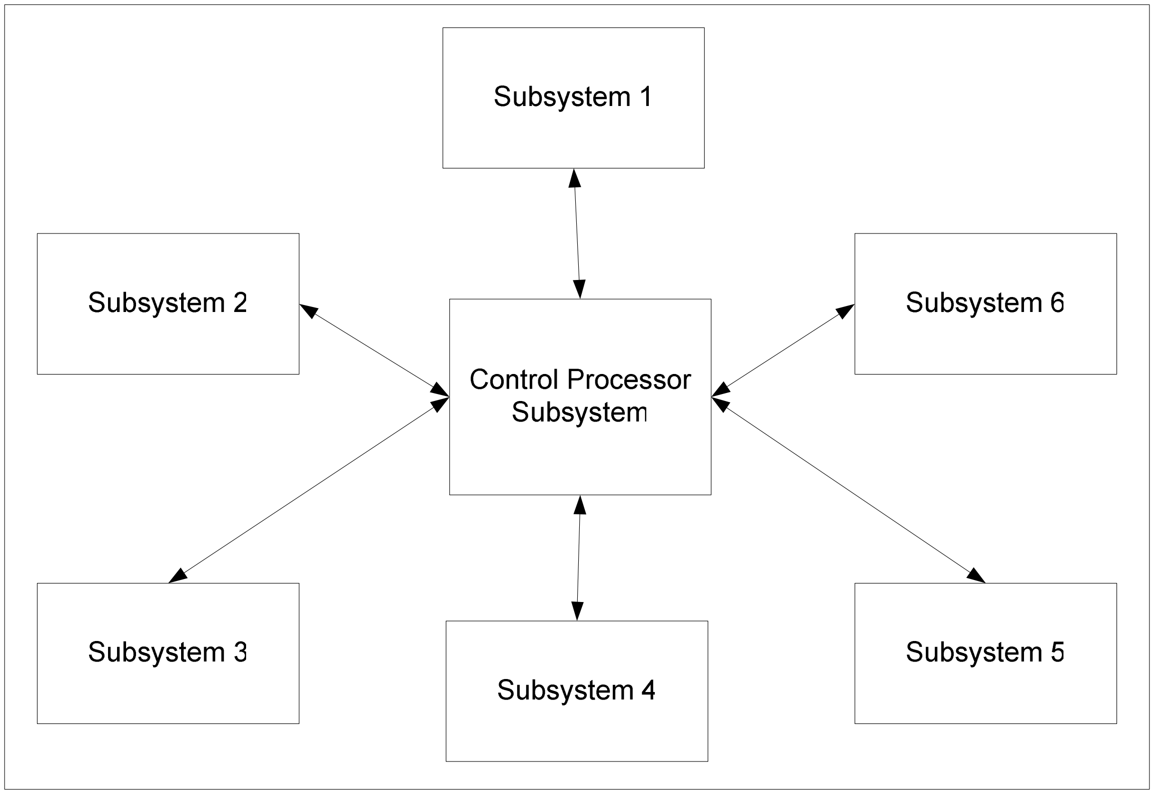


Figure 4: The simple system interfaces of the considered project.

In this thesis the intention is to supply the required framework and process to apply TDD in development of the control processor software. The author of this thesis thinks that the return on investment (ROI) of TDD is maximized in the development of control processor software. The amount of ROI of TDD on control processor software can be estimated by considering the testing and debugging cost of this software. Since this subsystem has interfaces with all other subsystems, subsystem test of this software requires simulators of the all other subsystems. Because of the interfaces with all other modules, debugging the control processor is especially costly. So, the gains in finding bugs during unit testing as opposed to integration become particularly significant.

3.2 Current Development Process

Current development process is composed of iterations. Each iteration can be thought as a small waterfall. Life cycle of an iteration is shown in Figure 5. This life cycle starts with writing software requirements. After writing software requirements first design is done. First design consists of key design decisions and software architecture. In the detailed design phase definition of the classes, relations between them, their attributes and their operations are determined. After that, implementation is started. After producing the code, unit tests are done. After passing the unit tests, it is time to integrate the units to get subsystem and test it. After accomplishing all subsystem tests, system can be produced by integrating subsystems. Finally after testing the system, it is time to deploy system and run it. As it can be seen, if a defect is detected from any tests or run-time, firstly required change is identified. The cycle is repeated from detailed design phase.

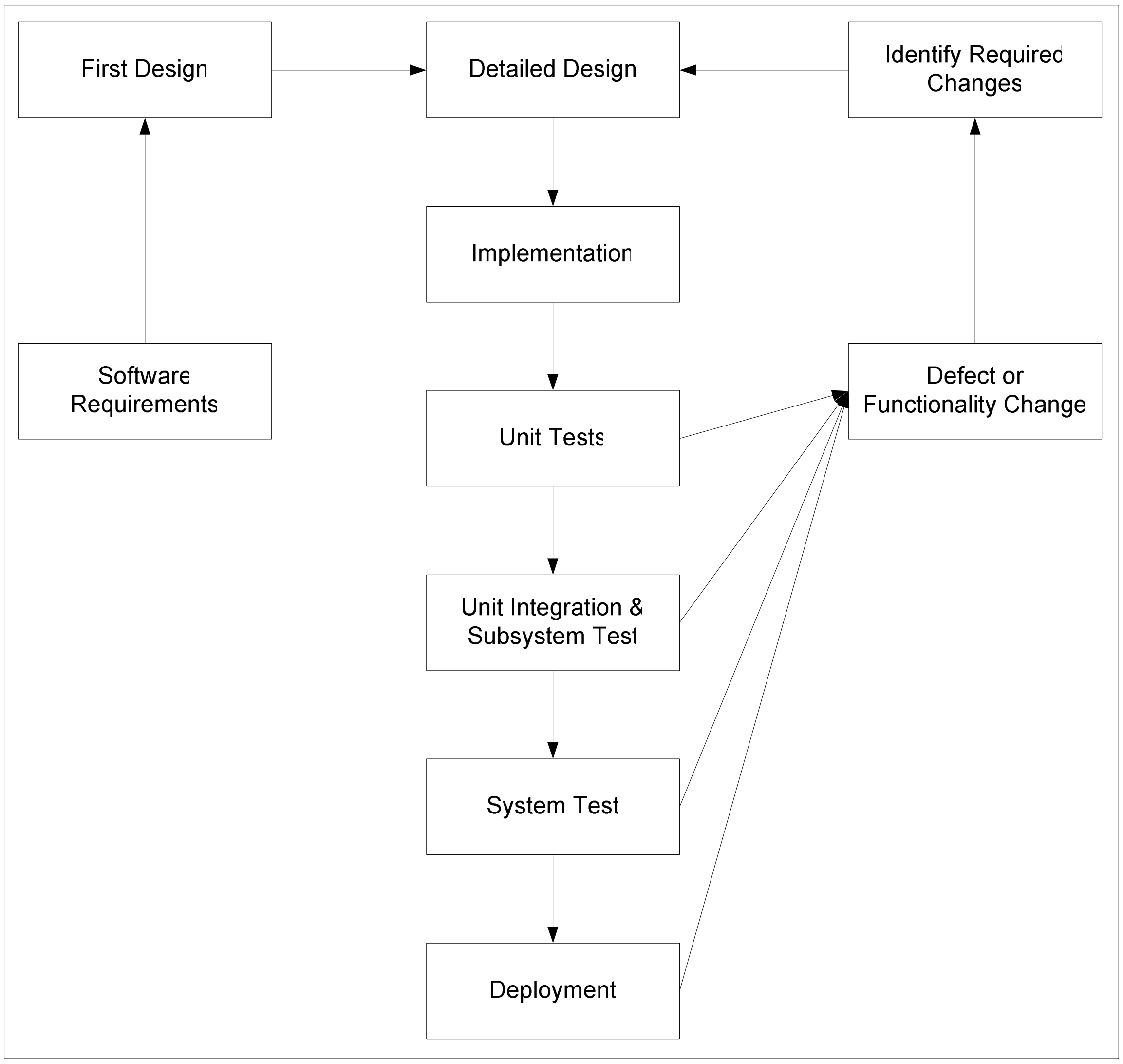


Figure 5: Planned current life cycle of an iteration.

The author of this thesis, after intensive personal experience with this process and after extensive interviews with members of the software development team, has decided that the disadvantages of the current process can be listed as below:

- The most important disadvantage of this system is the huge time elapsed between decisions and the feedback obtained on their results [1].

- Until the unit tests, it is not detected whether the code works or not.
- Since unit tests are the post-coding activity and generally there is a timing pressure, not enough effort is spent on unit tests (see Figure 2).
- The sources of bugs are not easily detected, because it must be searched in all code of unit. Every developer may have the experience of spending days to find a small bug.
- Since tests are not automated and consume too much time, it is discouraged to make small refactorings. This causes getting a code that is full of patches. So the code becomes unmanageable.
- Since unit tests are not automated, it is discouraged to make tests after small changes. This prevents early finding of bugs.
- Since the defects those are founded from unit integration, acceptance and run-time tests are not accumulated on unit tests, same defects can occur in new versions of the software.

By the help of the influence diagram shown in Figure 2, it can be seen that the stress and long time required for tests decrease the left effort to testing. Although internal dynamics of the current process tend to decrease unit testing effectiveness, if achieving good testing quality is wanted, a high amount of resources must be devoted to testing. But normally many companies tend to supply this resource only for safety critical systems. The disadvantages listed above and relations between stress, tests and errors in influence diagram generate differences between planned process and actual process. For the projects that are not safety critical the actual process is shown in Figure 6. As it is seen, most of the defects are found after system test stages. Therefore the time to fix a defect is increased. The results of a survey [12] on the amount of time that is spent on debugging activity are shown in Figure 7. As it seen in this graph, there is a huge time that is spent on debugging. So when debugging time can be decreased, the project cost also decreases significantly.

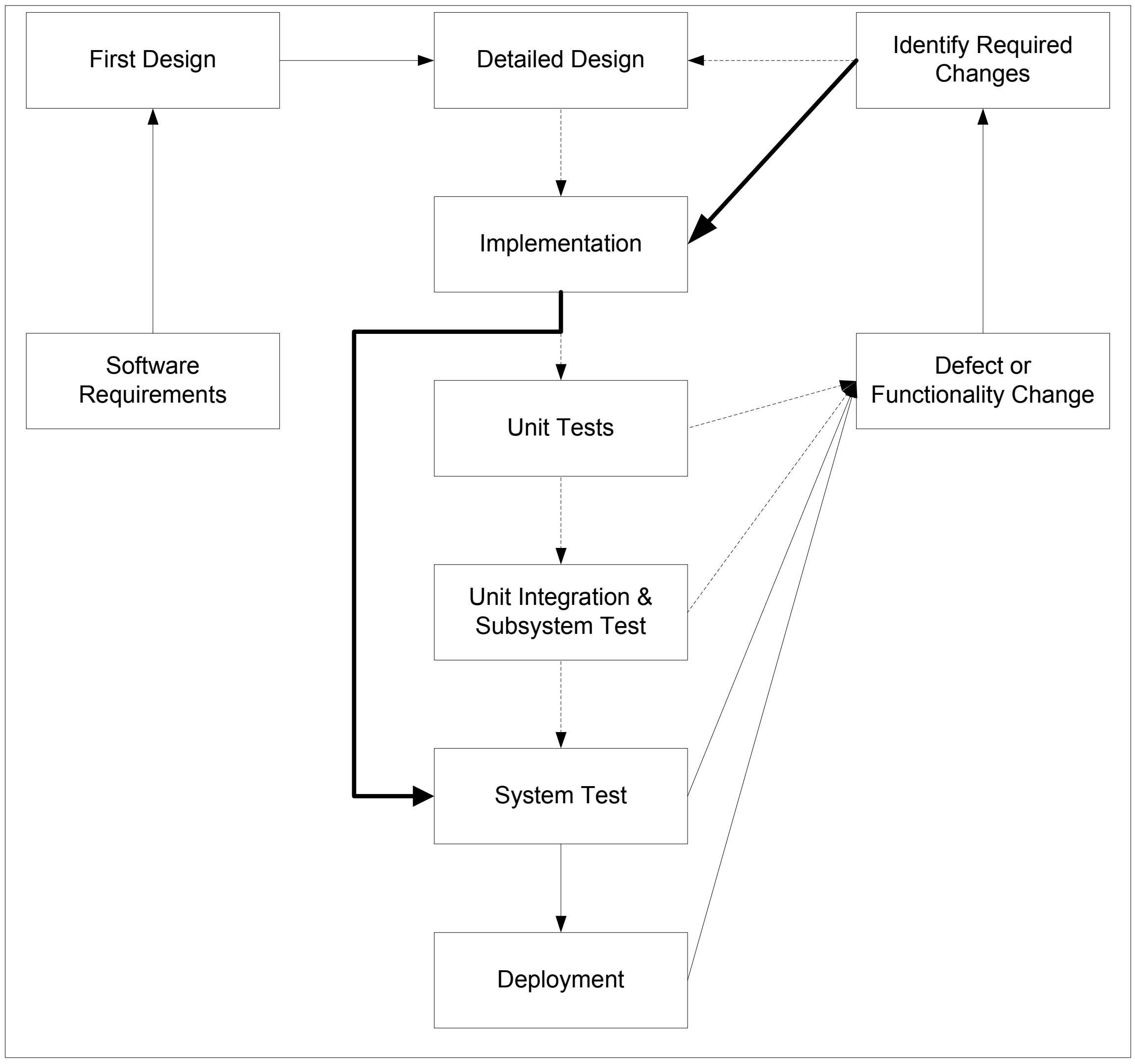


Figure 6: Actual current life cycle of an iteration.

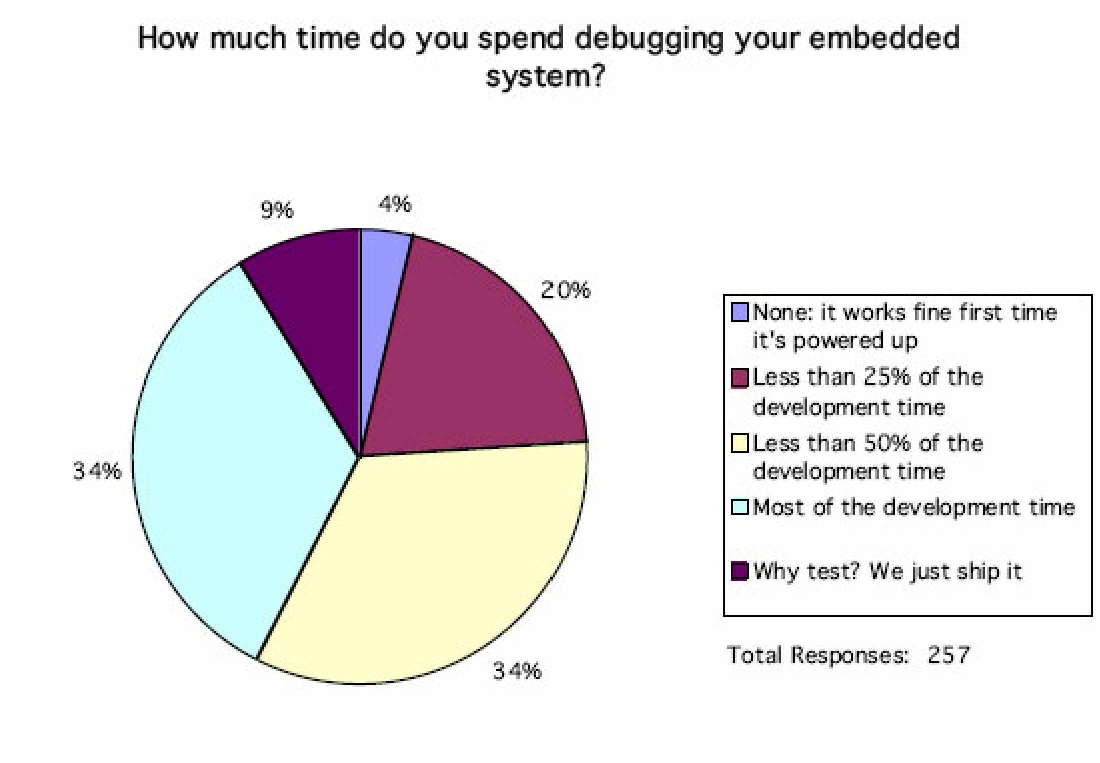


Figure 7: Result of survey of time spent for debugging activities [12].

3.3 Suggested TDD Process

In this section, the subject of where to locate TDD in the big picture is discussed. The author of this thesis recommends locating TDD to cover all of the detailed design, implementation and unit test phases. The graphical view representing this proposal is shown in Figure 8. It must be stated that TDD cycle is not a long term cycle it must be repeated more than once in a day. This recommendation aims to integrate TDD into a classical development cycle, without necessitating a transition to agile methodologies of software development.

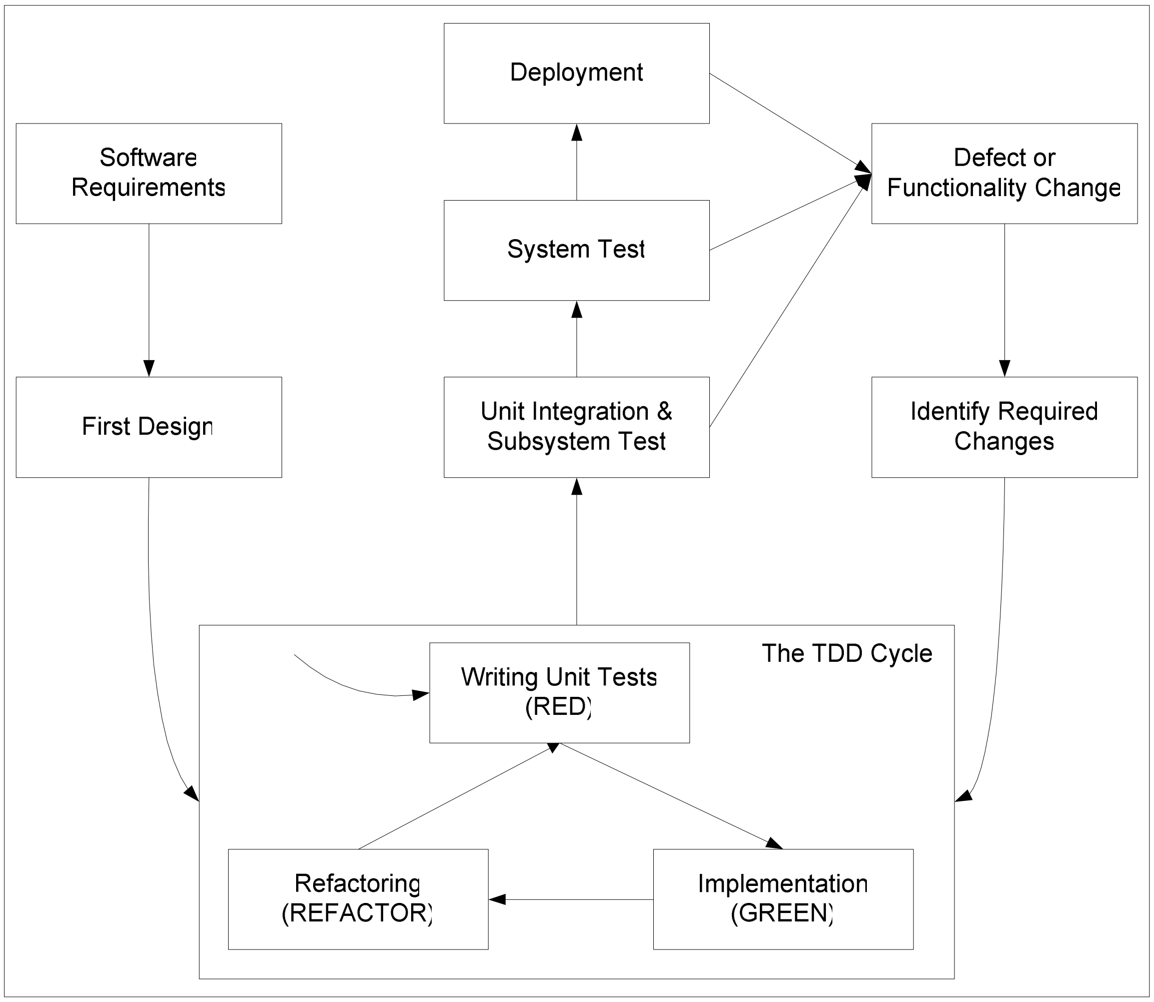


Figure 8: Recommended location of TDD cycle in the whole process.

Before explaining the process it must be stated that although unit integration should be a continuous activity, it is not stated in this process because of that unit integration is not included in TDD. For the sake of concretizing the TDD process, let us start with the interpretation of the first design phase output and consider how it is used for testing and how it must be used for easy test. As it is explained before, the output of the first design is key design decisions and software architecture. A general software architecture is shown in Figure 9.

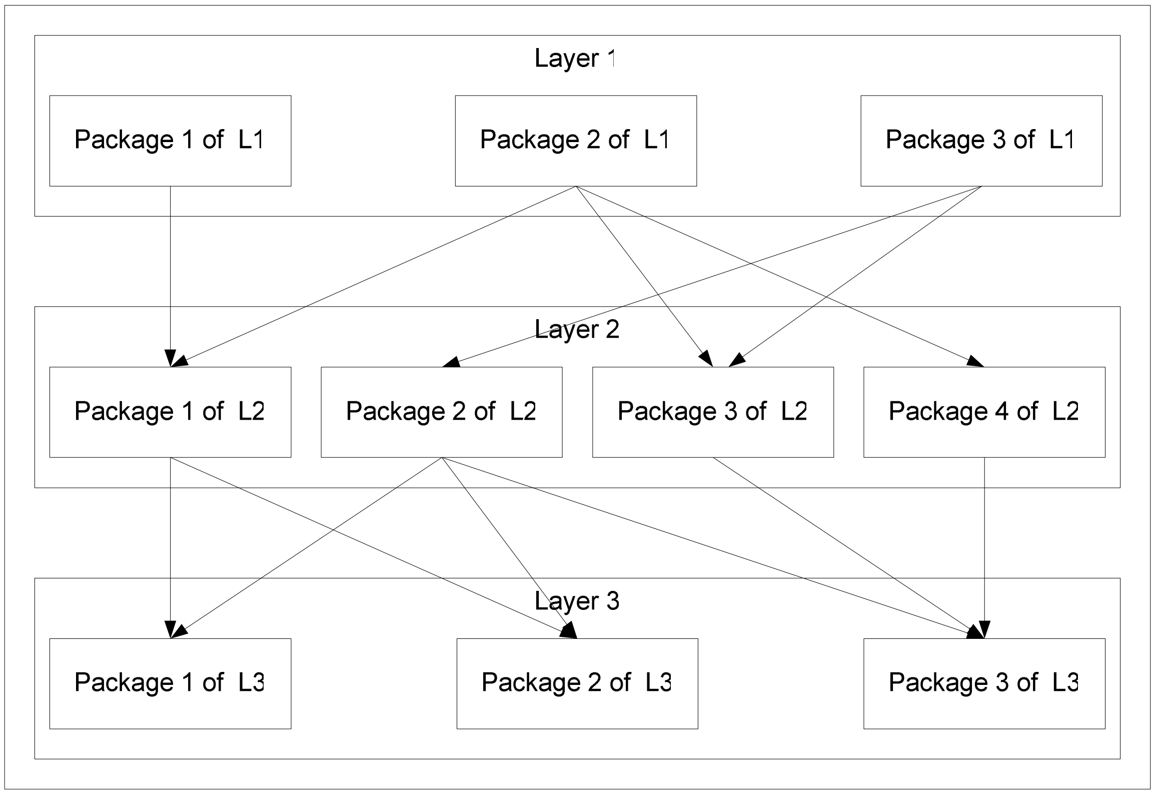


Figure 9: Sample general software architecture

These packages must have fine granularities so that they can be tested easily. To achieve this granularity it can be stated that each of these packages consists of at most 7 classes (this number can be determined by the experienced developers of the project). So this is a design decision that is implied by the concern of testability. For isolated testing of a package, dependencies of package must be on abstract (interface) classes. This is also a design decision that is implied by testability. A way to achieve this is using a Façade [11] in each package. Two examples of this type of packages are shown in Figure 10. So every package has abstract entry points. Package A is an example of a package of low complexity. An example of this type of packages is given for abstracting hardware dependencies by J. Grenning [13]. An entry point can have different implementations. Package B is an example of a package of complex functionality. This package can be an implementation of a function at the application layer. Since the packages of control processor software is not too big, the recommendation of the author of this thesis on the selection of classes for unit testing is testing these entry points (i.e. implementations of the façade classes). According to this recommendation classes A1, A2 and BController, in the example in Figure 10, must be selected for unit testing. Some advantages and disadvantages of this recommendation are given below.

Advantages:

- Package interfaces are closer to the software requirements than an ordinary class in a package.
- Internal design of a package can be refactored without the necessity of updating unit tests.
- Imply software architecture with high cohesion and low coupling.

Disadvantages:

- Preventing the using of tests as a tool that implies good design in the level of packages. So achieving good class granularity in the package is left to the developer.

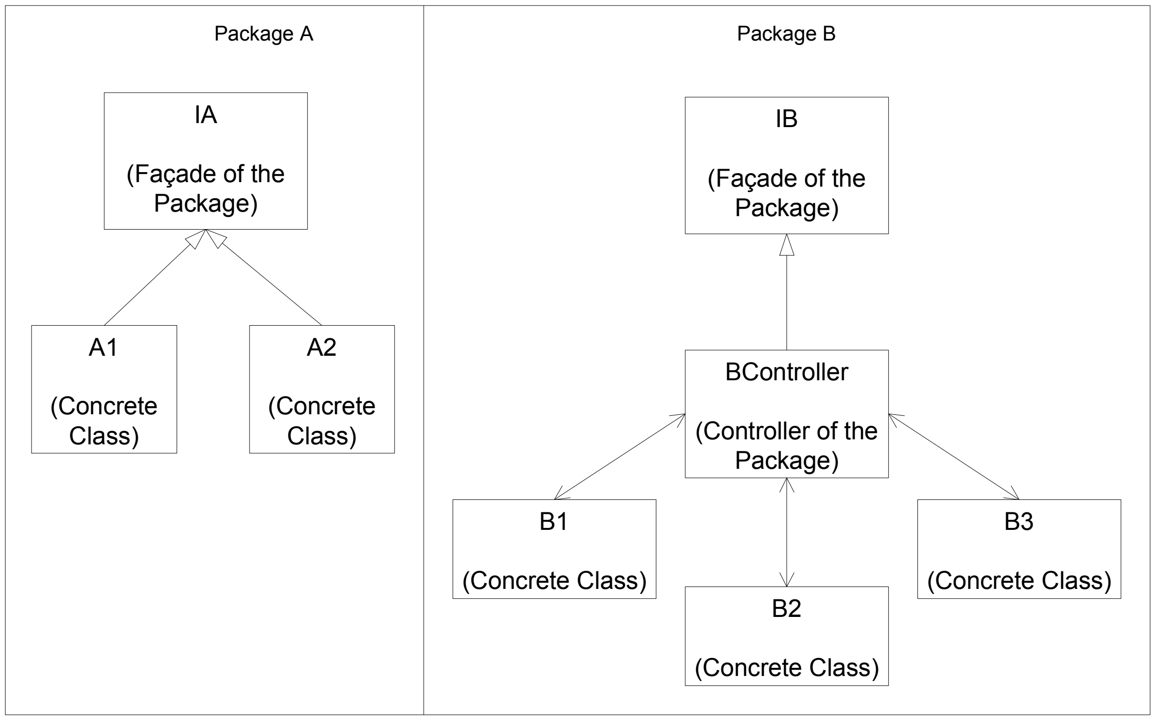


Figure 10: General Structure of two packages that allow isolated test of packages.

A possible misunderstanding here is that if all operations of the interfaces of the packages are determined in the first design phase, then development can not be claimed to have been driven according to the tests. To correct this misunderstanding it must be stated that it is not expected that the interface class (façades) of packages are completely specified after the first design phase, it is just expected that the definitions of the packages must be finalized. The operations of these façades are matured in the TDD cycle, so we can say that in this process development is driven by tests.

After determining architecture, layers, packages and making design decisions (façades, and some project specific decisions), it is time to enter the TDD cycle. J. Grenning [13] recommends a TDD cycle specifically for embedded systems. This cycle is shown in Figure 11. This cycle consists of four small cycles.

As it is seen, the first cycle is same as the recommended cycle by K. Beck [1]. This first cycle is applied on the development environment. This first cycle is the core of embedded TDD cycle and it must be run as many times as possible.

The second cycle is the compilation of code for target environment. Some compilation differences are determined and fixed in this second cycle. This cycle must be applied when there is doubt about compilation difference between development and target environments.

The third cycle is running the tests those can be performed automatically on the target.

Hardware dependent codes must be tested manually. These tests comprise the fourth cycle. Generally the bottom layer is the layer that abstracts the hardware from software. So usually the tests of the bottom layer can not be done automatically.

This embedded TDD cycle has some implications:

- To decrease the required manual tests, the packages which are manually tested must be as stable as possible. This requires the encapsulation of changeable part from the manually tested packages.
- To test as many parts as possible on the development environment, target dependent codes must be encapsulated. Usually this requires an OS abstraction layer.

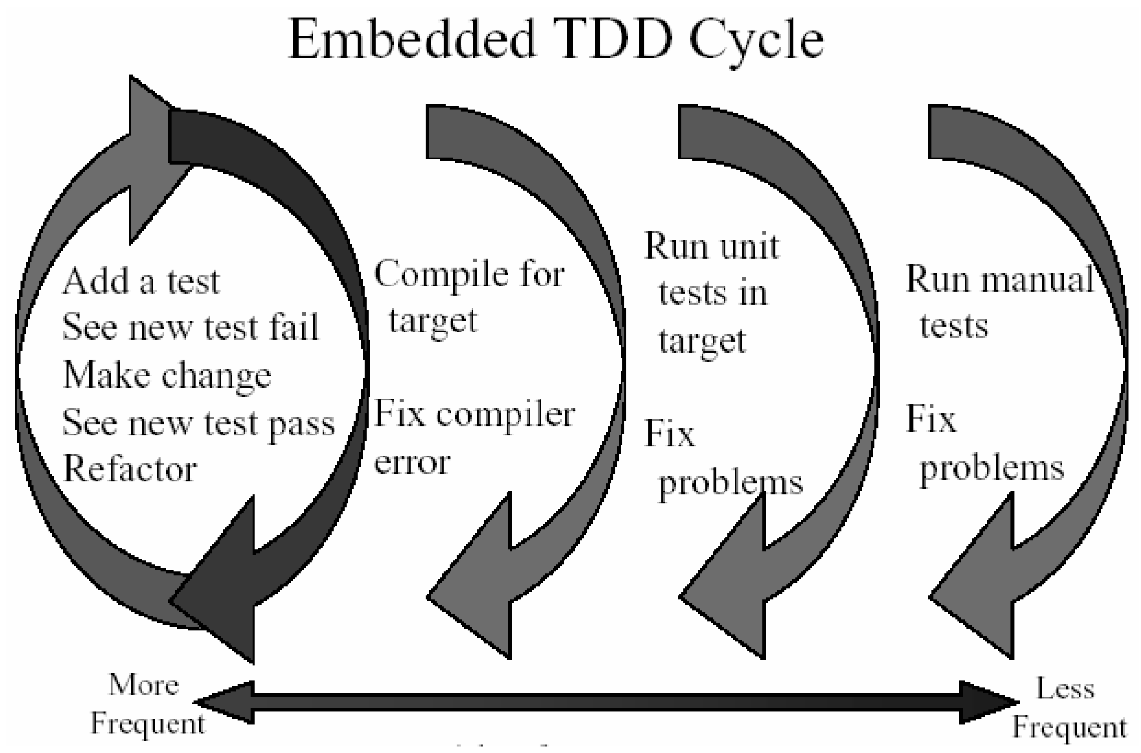


Figure 11: Recommended embedded TDD cycle by J. Grenning [13].

3.4 Required Unit Test Framework

The only required tool that is used in TDD is a unit test framework. But this unit test framework must have some properties to respond to the challenges of TDD and the conditions of TDD those are explained above. Below, the fundamental requirements from this tool shall be presented:

Tests must be written quickly. Writing of test cases can be separated in two parts. One of them is writing test methods, the other one is writing mocks. JUnit like frameworks are enough to write test methods fast enough. So the unit test framework that will be produced in the scope of this thesis must facilitate supply JUnit like properties and an easy way of mocks. Also the framework that will be produced must be easily used in Rhapsody in C++.

Tests must be able to run on the development environment. Therefore the framework that is produced for embedded systems in the scope of this thesis must be OS independent.

Event-driven (reactive) software must be testable. JUnit like frameworks are designed for testing sequential software. Therefore the framework that is supplied within the scope of this thesis must provide an easy way to test reactive software.

Timing requirements must be testable. Time is an important concern for most of embedded software. Therefore the framework that will be produced must supply a way to measure and test timing concerns.

CHAPTER 4

RhapUnit

In this chapter, the automated unit test framework, RhapUnit, is developed both to support the TDD approach and to illustrate its usage in a concrete software development project will be described. RhapUnit is an automated unit test framework. It works both on development environment and on target environment. So that unit tests can be executed on both environment. The main functional requirements of this framework are defined in section 3.4. It is mainly based on Kent Beck's xUnit [1] pattern. In conformance with the TDD philosophy, in this chapter, the description of RhapUnit is documented by using the tests used in its development. This can be thought as a verification of using tests as documentation. RhapUnit and its self-test classes are shown in Figure 12. Properties of all classes are defined in Appendix C.

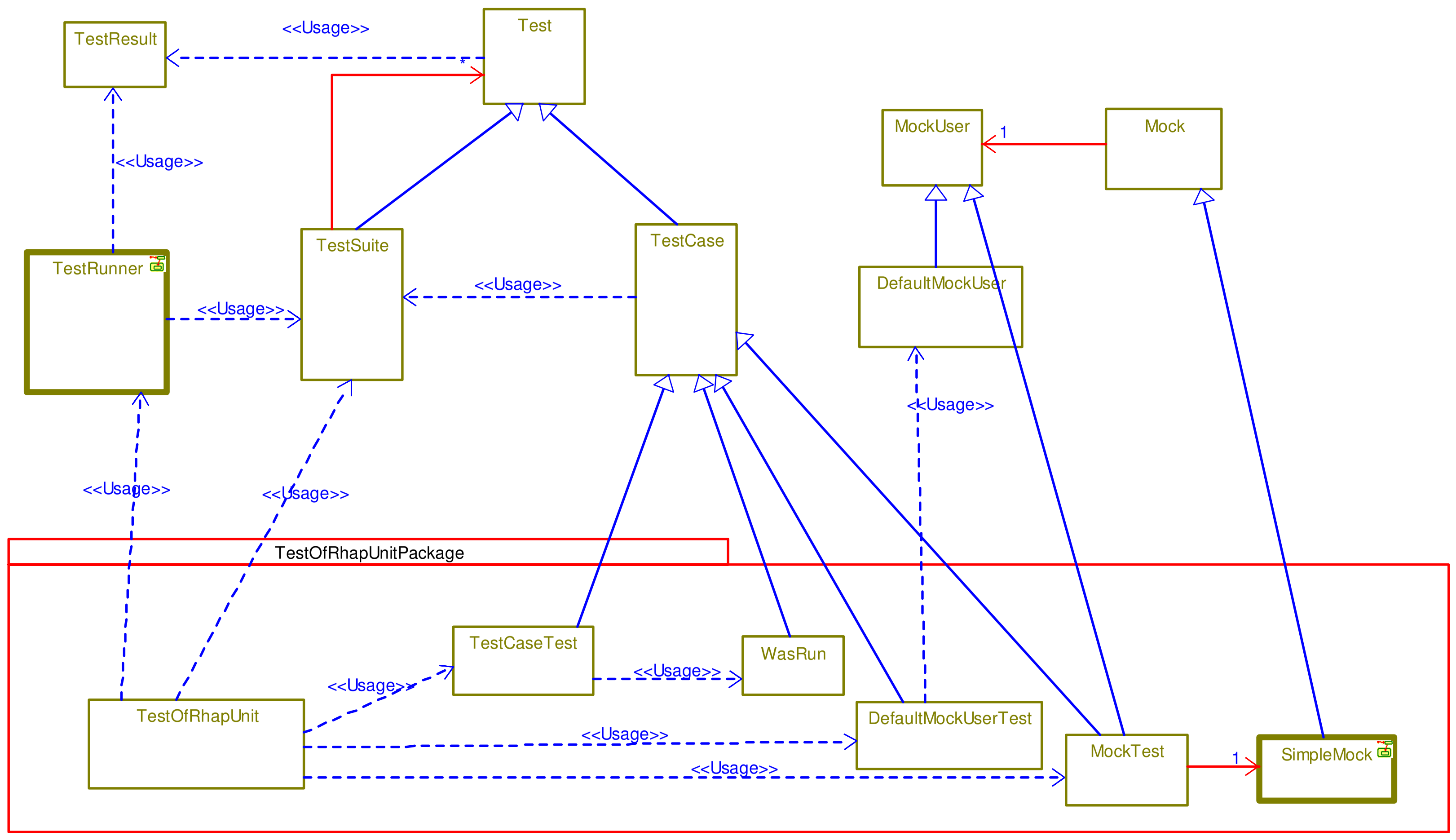


Figure 12: Object model diagram of the RhapUnit and its self-test classes

4.1 xUnit Properties

A xUnit like unit test framework should supply following requirements [1]:

- Invoke test method: User could select an operation from the test class.
- Invoke setUp first: Before running any test method setUp should be called.
- Invoke tearDown afterward: After running any test method tearDown should be called.
- Run multiple tests: Tests could be grouped to run together.
- Check post-conditions and pre-conditions: There should be a facility to check conditions.
- Report collected result: All tests results could be reported together.

In the next subsections, tests of test framework are used to explain how these requirements are captured.

4.1.1 Creation and Running of a Test Case

The body of method “testTemplateMethod” test case is the following:

```
TestCase * test;
CREATE_TEST( WasRun, testMethod, test);
WasRun * testWasRun = (WasRun *) test;
assert( testWasRun->getLog() == "" );
test->run(result);
assert( testWasRun->getLog() == "SetUp TestMethod TearDown " );
delete test;
```

This test case tests the first three requirements of xUnit. As it is seen, by the help of CREATE_TEST macro a test case is created. First parameter of this macro is the test class. Second parameter is the test method of that class. The third parameter is the pointer of test case. As it is proved by the log string, by calling method “run” of this test case, “setUp”, “testMethod” and “tearDown” methods of WasRun class are called sequentially. testMethod is the method that is given to the CREATE_TEST macro. CREATE_TEST macro is inspired from CppUnitLite [www.objectmentor.com/resources/downloads/index].

4.1.2 Result Reporting

Reporting of failed test and succeeded test are shown in following tests:

```
testFailedTest:
TestCase * test;
CREATE_TEST( WasRun, testBroken, test);
test->run(result);
CHECK( "", "1 run, 1 failed" == result.summary() );
delete test;

testSucceeded:
TestCase * test;
CREATE_TEST( WasRun, testSucceeded, test);
test->run(result);
CHECK( "", "1 run, 0 failed" == result.summary() );
delete test;
```

As it is seen in both test, result of tests are collected in result object. Test report is used to collect how many tests are run and how many of them are failed.

4.1.3 Running Multiple Tests

How multiple tests are run is shown in following test:

```
testSuite:
TestSuite * suite = new TestSuite();
ADD_TEST(TestCaseTest, testFailedResultFormatting, suite );
ADD_TEST(TestCaseTest, testTemplateMethod, suite );
ADD_TEST(TestCaseTest, testSucceeded, suite );
suite->run(result);
assert("3 run, 0 failed" == result.summary());
```

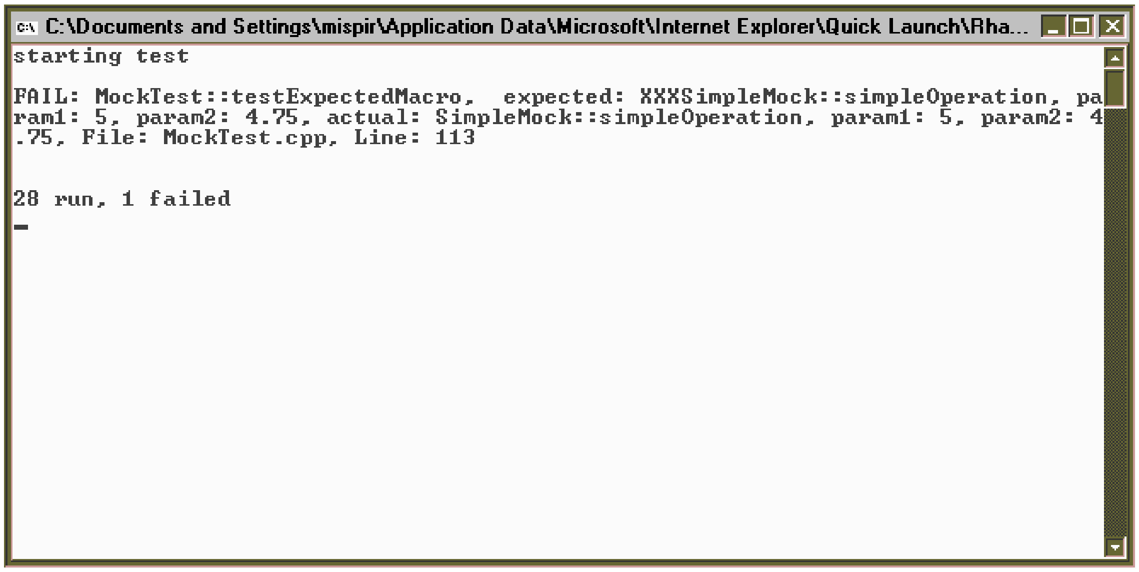
As it is seen from testSuite, by using ADD_TEST macro, new test cases are created and added to the given TestSuite. When the run method of suite is called, all attached tests are run. This is checked by result object.

4.1.4 Checking Conditions

CHECK macro is used to check conditions. Testing of this macro is following:

```
testCheckMacro:
try
{
   CHECK("",TRUE);
}catch(OMString message)
{
   assert( FALSE );
}catch(...){
   assert( FALSE );
}

try
{
   CHECK("",FALSE);
   assert( FALSE );
}catch(OMString message)
{
}catch(...){
   assert( FALSE );
}
```

As it is seen from testCheckMacro, if the given condition is false CHECK macro throw an exception with string parameter. This CHECK macro is used to check all pre-conditions and post-conditions. The failed condition report is shown in Figure 13. The fail report gives the information of test class, test method, condition, file and line number.



```
C:\Documents and Settings\mispir\Application Data\Microsoft\Internet Explorer\Quick Launch\Rha...
starting test

FAIL: MockTest::testExpectedMacro,  expected: XXXSimpleMock::simpleOperation, pa
ram1: 5, param2: 4.75, actual: SimpleMock::simpleOperation, param1: 5, param2: 4
.75, File: MockTest.cpp, Line: 113


28 run, 1 failed
_
```

Figure 13: The output of a failed condition check.

4.2 Mock Properties

Mock objects are used to isolate testing of a class from its dependent classes. This is done by creating mock objects of these classes. Detailed information about mock objects can be found in brown [14]. A mock object has to supply the following functionalities:

- Expectations could be set

- Return values could be set
- Verification of actual result should be done

In RhapUnit, there are two objects to make easy using of mocks. One of them is MockUser and the other one is Mock.

4.2.1 Mock User

The main responsibility of mock user is verification of the fact that the expected calls and actual calls are same. The MockUser interface includes the following operations:

- addExpectedMockCall: By this service, expected calls are set. Calls are representing as strings.
- actualMockCall: By this service, actual mock calls are set. Again calls are representing as strings.
- verify: By this service, mock user verifies that all expectations are accomplished.

MockUser is an abstract interface, so different implementations of mock user can be done. Default implementation of is supplied in RhapUnit. Again we can use tests to describe DefaultMockUser.

testFailLessCall

```
try
mockUser -> addExpectedMockCall( "XXX:YYY" );
OMBoolean isContinueAfterVerify = FALSE;
OMBoolean isVerifyThrowException = FALSE;
{
    mockUser -> verify();
    isContinueAfterVerify = TRUE;
} catch(...)
{
    isVerifyThrowException = TRUE;
}
CHECK( "", !isContinueAfterVerify );
CHECK( "", isVerifyThrowException );
```

As it is seen from testFailLessCall method, if the number of default mock user actual calls is less than the number of expectation calls, verify throws an exception.

```
testFailWrongCall:
mockUser -> addExpectedMockCall( "XXX:YYY" );
OMBoolean isContinueAfterActualCall = FALSE;
OMBoolean isActualCallThrowException = FALSE;
try
{
    mockUser -> actualMockCall( "XXZZ" );
    isContinueAfterActualCall = TRUE;
} catch(...)
{
    isActualCallThrowException = TRUE;
}
CHECK( "", !isContinueAfterActualCall );
CHECK( "", isActualCallThrowException );
```

As it is seen from testFailWrongCall, if actual mock call is different from the expected call, actualMockCall method throws an exception.

```
testSucceededForMany:
mockUser -> addExpectedMockCall( "Call 1" );
mockUser -> addExpectedMockCall( "Call 2" );
mockUser -> addExpectedMockCall( "Call 3" );
try
{
    mockUser -> actualMockCall( "Call 1" );
    mockUser -> actualMockCall( "Call 2" );
    mockUser -> actualMockCall( "Call 3" );
    mockUser -> verify();
} catch(...)
{
    CHECK( "", FALSE );
}
```

As it is seen from testSucceededForMany method, if actual mock calls value and sequence is same as the expectations, there isn't any exception occurs.

```
testFailMuchCall Operation:
mockUser -> addExpectedMockCall( "Call 1" );
mockUser -> addExpectedMockCall( "Call 2" );
OMBoolean isExceptionThrown = FALSE;
try
{
    mockUser -> actualMockCall( "Call 1" );
    mockUser -> actualMockCall( "Call 2" );
    mockUser -> actualMockCall( "Call 3" );
} catch(...)
{
    isExceptionThrown = TRUE;
}
CHECK( "mockUser must throw exception", isExceptionThrown );
```

As it is seen testFailMuchCall, if actual mock calls are more than the expectations, actualMockCall method throws an exception.

4.2.2 Mock

The responsibility of mock is to supply easy ways of:

- setting expectations
- setting return values
- inform mock user, about expectation and actual calls.

Mock objects are inherited from Mock. To show how mock objects are written easily, an implementation of an operation is given.

```
char* simpleDemoOperation( int iParam, float fParam ) {
    MOCK_START( SimpleDemo, simpleDemoOperation );
    MOCK_ADD_PARAM( iParam );
    MOCK_ADD_PARAM( fParam );
    MOCK_END();
    MOCK_RETURN( char* );
}
```

MOCK_START is used to log class and method name. MOCK_ADD_PARAM is used to log input parameters. MOCK_END is used to inform mock user about calling of this operation. MOCK_RETURN is used to return setted values.

Mock object has three modes. These are:

- NO_LOGGING: In this mode, calling a mock operation do nothing. This is the default mode of the mock.
- EXPECTATION: In this mode, mock calls are registered to mock user as expectations by using addExpectedMockCall method of mock user.
- ACTUAL: In this mode, mock calls are registered to mock user as actuals by using actualMockCall method of mock user.

Mock object operations:

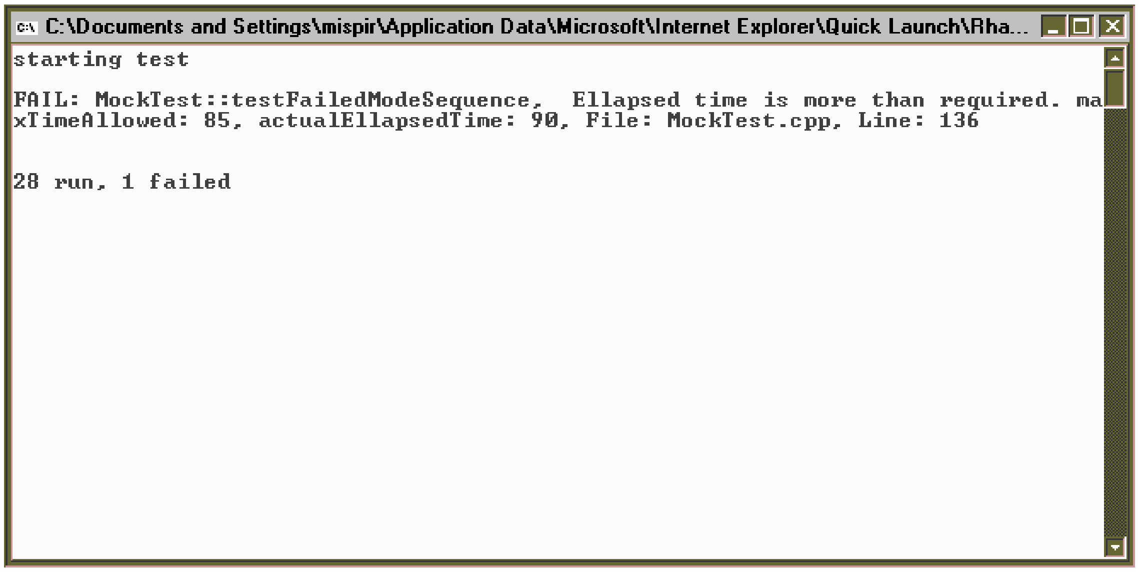
- startExpectation: Set the mode of mock as ECPECTATION.
- startActual: Set the mode of mock as ACTUAL.
- stopMockLogging: Set the mode of mock as NO_LOGGING.
- addReturnValue: Setting the return value of an expected operation. This method must be called before setting expexted operation.

4.3 Time Test

As it is described in section 3.4, RhapUnit have to supply a way to test timing requirements. Following test defines how timing requirements are tested in RhapUnit:

```
testTimeCheck:
OMBoolean isExceptionThrown = FALSE;
try
{
    START_TIMER(20);
    OXFTDelay(100);
    CHECK_TIME("", 79);
} catch (OMString & message)
{
    isExceptionThrown = TRUE;
}
CHECK( "Time control must throw an exception.", isExceptionThrown );

isExceptionThrown = FALSE;
try
{
    START_TIMER(20);
    OXFTDelay(100);
    CHECK_TIME("", 121);
} catch (OMString & message)
{
    isExceptionThrown = TRUE;
}
CHECK( "Time control shouldn't throw an exception.",
!isExceptionThrown );
```

START_TIMER macro starts the timer. It takes the time duration between ticks in milliseconds. In this test, timer is initialized with 20 milliseconds time tick interval. So time check must be done according to interval of +-20 milliseconds. CHECK_TIME checks whether actual duration is smaller than the given value or not. As it is seen from testTimeCheck, if the actual time duration is longer than the given time value, CHECK_TIME throws an exception. The output of a time fail is shown in Figure 14.



C:\Documents and Settings\mispir\Application Data\Microsoft\Internet Explorer\Quick Launch\Rha...

```
starting test

FAIL: MockTest::testFailedModeSequence,  Ellapsed time is more than required. ma
xTimeAllowed: 85, actualEllapsedTime: 90, File: MockTest.cpp, Line: 136


28 run, 1 failed
```

Figure 14: The output of a failed time check.

4.4 Reactive Object Test

As described in section 3.4, reactive objects must be testable. Reactive objects are state machine implementations. Generally these objects run on their own thread domain and are called also as active objects. The first thought that comes to mind is to run these objects in different tasks and send events to these objects. After that wait for sufficiently long to allow the reactive object to consume event. Then, check expected results. But as TDD requires a simple solution, a way to test reactive objects sequentially is searched. It is understood that by using Rhapsody framework services, sequential test of an active reactive object is possible. The detailed test of an active object is explained in the package "ExampleTestOfActiveClass" in Appendix C. To test a state chart, the following functionalities should be generated: startBehaviour, generating events and generating timeouts.

To test startBehaviour of the stateChart, instead of calling method “startBehavior”, call method “rootState_entDef”. To test consuming an event, instead of using macro “GEN”, call method “takeEvent”. To test timeout events, first create an instance OMTimeOut, then coll method “takeEvent” with this instance.

4.5 OS independency

The recommended embedded TDD cycles requires an operated system independent unit test framework. To meet this requirement the framework was developed using Rhapsody OS abstraction framework. Selftests of RhapUnit are run successfully on the following operating systems: Windows XP, VxWorks 5.4, and VxWorks 5.5.

4.6 Used TDD Patterns

During the development of RhapUnit, some TDD patterns [1] are used. These patterns are:

Learning Tests: Examples of learning tests are testTryCatch method of TestCaseTest and TestVector method of MockUserTest. To check whether try-catch functionality is run on VxWorks 5.4 or not, testTryCatch is written. To check methods of collection vector, testVector is written.

One Step Test: As it is seen from the tests, functionalities are added in small steps. This causes to focus on small problems and lead to find fine solutions for small problems. During the development of RhapUnit, Iterations do not take more than an hour.

Self Shunt: Testing an object that communicates with other object need to use mock objects of other objects. But if the TestCase class itself uses as mock object, this is called self shunt. As it is seen in Figure 12, MockTest also inherit from MockUser, and it is given to tested object (SimpleMock) as its user.

Broken Test: Before leaving a session, a new test is written for the sake of remembering starting point.

One to Many: Implementation of DefaultMockUser capabilities and Mock setting return value capability are starts with implementing for only one mock call, then go on to implement for collection of mock calls.

CHAPTER 5

CONCLUSIONS

In this thesis work, an improvement to established software development practice in a large firm has been proposed in developing embedded software by using TDD.

To achieve this goal, a methodology is recommended to integrate TDD into a classical development cycle, without necessitating a transition to agile methodologies of software development. Also the required unit test framework is written to make this process applicable for the development environment Rhapsody.

Reducing defects, shortening the feedback loop on design decisions, continuously refactoring, implying thinking what is needed before how it is implemented and accumulating test experiences are the key advantages of TDD. Reducing defect not only decrease the maintenance cost, but also have some positive psychological and social effects such as trust of colleagues to each other. By shortening the feedback loop on design decisions, developers don't spend much time on wrong decisions. In the classical approach, code becomes spaghetti as time elapses. But in TDD, continuous refactoring prevents this and always maintains the code in a state of sufficiently good design as discussed in sec. 2.1. Leading programmers to think on what is needed before how it is implemented prevent superfluous generalization. Finally accumulating the test experiences reduces the cost of integration tests and system tests. This feature makes TDD invaluable for the large, complex and hard to test projects.

There are still many questions about the advantages and disadvantages of TDD. A possible way to determining the advantages and disadvantages of TDD can be: Two large projects developed in parallel. One of them is developed with classical method, the other one is developed with TDD. The differences of these projects can be used to determine advantages and disadvantages of TDD according to classical development methodology. Unfortunately, because of its cost, this experience is not applicable, neither with in the scope of this study nor in general. Even if this were feasible, since programmers are different in two projects, it can not be guaranteed that the differences arise only from TDD. The work that can be done is comparing the metrics of a project that is developed with TDD with the old accomplished projects. As such an extensive application was not possible with in the scope of our study, we consider it sufficient to state that a definitive evaluation of TDD can be possible only after this has been done as a future work. The main thrust of our work has been simply to illustrate the applicability of TDD as a first attempt.

The author of this thesis believes that TDD can be applicable and it increases the productivity. There are two main reasons for this belief: One of them is that the problems which the author encountered in his past projects experiences are solved in TDD. Some of these problems are: huge debugging activity, rotting of the code, focusing on big units, long feedback time, and loose predictability. The other reason is the success of RhapUnit. Although one or two months were planned to develop Rhapsody, it has taken just two weeks. This has been a clear indication of productivity increase.

The projects in which TDD may not be applicable can be identified by the help of challenges described in section 2.3. If there is not an appropriate tool support, writing automated unit test can become a critical time consuming activity. If the test duration is not sufficiently reduced or tests can not start as easy as clicking on a button, developers can lose their motivation of running the tests. These conditions may prevent the application TDD methodology.

The aim of this study was simply to show the applicability of TDD in an embedded software development project. It has only been demonstrated that TDD is a viable approach within the context of a simple project. No generalizations or proofs of unconditional superiority have been attempted. That would require much more extensive studies.

REFERENCES

[1] K. Beck, “Test Driven Development: By Example”, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002.

[2] R. Mugridge, “Test Driven Development and the Scientific Method”, IEEE Agile Development Conference (ADC’03), 2003

[3] J. U. Pipka, “Development Upside Down: Following the Test First Trail,” 18th European Conference on Object Oriented Programming, 2004

[4] S. Tuffle, “Design-for-Test Strategy Eases Unit Level Testing”, COTS Journal, vol. 6 pp. 39-44, February 2004.

[5] B. George, L. Williams, “An Initial Investigation of Test Driven Development in Industry”, ACM Symposium on Applied Computing SAC 2003, 2003

[6] M. Feathers, “Emergent Optimization in Test Driven Design”, XP2002 conference in Sardinia, 2002

[7] E.M. Maximilien, L. Williams, “Assessing Test-Driven Development at IBM”, IEEE 25th International Conferance on Software Engineering (ICSE’03), 2003

[8] R.C.Martin, “Baby Steps” CMP Software Development Column Craftsman, November 2002.

[9] J. Rasmusson, “Introducing XP into Greenfield Projects: Lessons Learned”, IEEE Software May/June 2003 pp 21-28, 2003

[10] M. M. Müler, O. Hagner, “Experiment about Test-First Programming”, Empirical Assessment In Software Engineering EASE '02, Keele, April 2002

[11] E. Gamma, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Longman, Inc, 19th Printing, 2000.

[12] J. Ganssle, “Twists on Testing”, Embedded System Programming column Embedded Pulse, March 2003

[13] J. Grenning, “Progress Before Hardware”, The Agile Times, vol. 4 pp 74 – 79, February 2004.

[14] M. A. Brown, E. Tapolcsanyi, “Mock Object Patterns”, 10th Conference on Pattern Languages of Programs, 2003

[15] IEEE Standard Glossary of Software Engineering Terminology, Technical Report IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, December 1990

[16] E. V. Berard, “Issues in the Testing of Object-Oriented Software”, whitepaper at itmWEB, http://www.itmweb.com/essay555.htm

APPENDIX A

TESTING TERMINOLOGY

Following terminology is taken from the IEEE Software Engineering Terminology [15].

Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Fault: An incorrect step, process, or data definition in a computer program.

Debug: To detect, locate, and correct faults in a computer program.

Failure: The inability of a system or component to perform its required functions within specified performance requirements. It is manifested as a fault.

Testing: The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.

Static analysis: The process of evaluating a system or component based on its form, structure, content, or documentation.

Dynamic analysis: The process of evaluating a system or component based on its behavior during execution.

Correctness:

- The degree to which a system or component is free from faults in its specification, design, and implementation.
- The degree to which software, documentation, or other items meet specified requirements.
- The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.

Verification:

- The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- Formal proof of program correctness.

Validation: The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Following terminology is taken from a white paper at itmWEB [16].

White-Box Testing: White-box testing is the testing of the underlying implementation of a piece of software (e.g., source code) without regard to the specification (external description) for that piece of software. The goal of white-box testing of source code is to identify such items as (unintentional) infinite loops, paths through the code which should be allowed, but which cannot be executed, and dead (unreachable) code.

Black-Box Testing: Black-box testing is the testing of a piece of software without regard to its underlying implementation. Specifically, it dictates that test cases for a piece of software are to be generated based solely on an examination of the specification (external description) for that piece of software. The goal of black-box testing is to demonstrate that the software being tested does not adhere to its external specification. (Note that if there is no "external specification" it will be difficult to conduct black-box testing.)

Gray-Box Testing: Gray-box testing is testing based on an examination of both the specification for a piece of software, and the underlying implementation (e.g., source code) for that piece of software. A typical goal of gray-box testing is to identify singularities in a piece of software, i.e., situations in which the behavior of a piece of software become unbounded.

APPENDIX B

AN EXAMPLE

To concretize how TDD works, compare classic way of coding with TDD on a little program [8] in Java that calculates the prime factors of an integer.

In classical way:

First write a good code that hit the goal as follows:

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
public class PrimeFactorizer {
  public static void main(String[] args) {
    int[] factors = findFactors(Integer.parseInt(args[0]));
    for (int i = 0; i < factors.length; i++)
      System.out.println(factors[i]);
  }
  public static int[] findFactors(int multiple) {
    List factors = new LinkedList();
    int[]     primes     =     PrimeGenerator.generatePrimes((int)
    Math.sqrt(multiple));
    for (int i = 0; i < primes.length; i++)
      for (; multiple % primes[i] == 0; multiple /= primes[i])
        factors.add(new Integer(primes[i]));
    return createFactorArray(factors);
  }
  private static int[] createFactorArray(List factors) {
    int factorArray[] = new int[factors.size()];
```

```
        int j = 0;
        for (Iterator fi = factors.iterator(); fi.hasNext();) {
          Integer factor = (Integer) fi.next();
          factorArray[j++] = factor.intValue();
        }
        return factorArray;
      }
    }
```

Now, test the program by running the main program with several different arguments. It seems to work. After that if unit tests are wanted by the manager than write them to show that this program works. But, since you write the program you don't motivate to write enough unit tests.

In TDD:

Start with the simplest test case: The first valid case is 2. And it should return an array with just a single 2 in it.

```
public void testTwo() throws Exception {
  int factors[] = PrimeFactorizer.factor(2);
  assertEquals(1, factors.length);
  assertEquals(2, factors[0]);
}
```

Then write the simplest code that would allow the test case to succeed:

```
public class PrimeFactorizer {
  public static int[] factor(int multiple) {
    return new int[] {2};
  }
}
```

Then add the simplest test case that may not succeed:

```
public void testThree() throws Exception {
  int factors[] = PrimeFactorizer.factor(3);
  assertEquals(1, factors.length);
  assertEquals(3, factors[0]);
}
```

Then write the simplest code that would allow the test case to succeed:

```
public static int[] factor(int multiple) {
    if (multiple == 2) return new int[] {2};
    else return new int[] {3};
  }
```

Then refactor it two make the code clean:

```
  public static int[] factor(int multiple) {
    return new int[] {multiple};
  }
```

Then add the simplest test case that may not succeed:

```
  public void testFour() throws Exception {
    int factors[] = PrimeFactorizer.factor(4);
    assertEquals(2, factors.length);
    assertEquals(2, factors[0]);
    assertEquals(2, factors[1]);
  }
```

Then write the simplest code that would allow the test case to succeed:

```
  public class PrimeFactorizer {
    public static int[] factor(int multiple) {
      int currentFactor = 0;
      int factorRegister[] = new int[2];
      for (; (multiple % 2) == 0; multiple /= 2)
        factorRegister[currentFactor++] = 2;
      if (multiple != 1)
        factorRegister[currentFactor++] = multiple;
      int factors[] = new int[currentFactor];
      for (int i = 0; i < currentFactor; i++)
        factors[i] = factorRegister[i];
      return factors;
    }
  }
```

Then refactor it:

```
  public class PrimeFactorizer {
    private static int factorIndex;
    private static int[] factorRegister;
    public static int[] factor(int multiple) {
      initialize();
      findPrimeFactors(multiple);
      return copyToResult();
    }
    private static void initialize() {
      factorIndex = 0;
      factorRegister = new int[2];
    }
    private static void findPrimeFactors(int multiple) {
      for (; (multiple % 2) == 0; multiple /= 2)
        factorRegister[factorIndex++] = 2;
      if (multiple != 1)
        factorRegister[factorIndex++] = multiple;
    }
    private static int[] copyToResult() {
      int factors[] = new int[factorIndex];
      for (int i = 0; i < factorIndex; i++)
        factors[i] = factorRegister[i];
      return factors;
    }
  }
```

Then add the simplest test case that may not succeed:

```
  public void testEight() throws Exception {
    int factors[] = PrimeFactorizer.factor(8);
    assertEquals(3, factors.length);
    assertContainsMany(factors, 3, 2);
  }
```

Then write the simplest code that would allow the test case to succeed:

```
    private static void initialize() {
      factorIndex = 0;
      factorRegister = new int[100];
    }
```

Then add the simplest test case that may not succeed:

```
  public void testNine() throws Exception {
    int factors[] = PrimeFactorizer.factor(9);
    assertEquals(2, factors.length);
    assertContainsMany(factors, 2, 3);
  }
```

Then write the simplest code that would allow the test case to succeed:

```
  private static void findPrimeFactors(int multiple) {
    for (int factor = 2; multiple != 1; factor++)
      for (; (multiple % factor) == 0; multiple /= factor)
        factorRegister[factorIndex++] = factor;
  }
```

Then add the simplest test case that may not succeed:

```
public void testThousand() throws Exception {
  int factors[] = PrimeFactorizer.factor(1000);
  assertEquals(6, factors.length);
  assertContainsMany(factors, 3, 2);
  assertContainsMany(factors, 3, 5);
}
```

It is passed, so the goal is achieved.

APPENDIX C

RhapUnit TEST FRAMEWORK

Following documentation is the output of Rhapsody reporter.

RhapUnit Package

Type information for Package RhapUnit

CREATE_TEST:

```
#define CREATE_TEST ( testClass, testMethod, pTestCase ) \
 { \
  class testClass##testMethod##Test : public testClass \
        { \
                public:  \
                        testClass##testMethod##Test () {testName =
OMString(#testClass) + "::" + OMString(#testMethod); } \
        void _run () { testMethod(); }; \
   }; \
   pTestCase = new testClass##testMethod##Test; \
 }
```

ADD_TEST:

```
#define ADD_TEST ( testClass, testMethod, pSuite ) \
{ \
        TestCase * test; \
        CREATE_TEST(testClass, testMethod, test ); \
   pSuite -> add( test ); \
}
```

CHECK:

```
#define CHECK ( message, condition ) \
{ \
        if ( !( condition ) ) \
        { \
                ostrstream buffer; \
                buffer << message << ", Condition: " << #condition << ", File: " <<
__FILE__ << ", Line: " << __LINE__ << '\0'; \
                char* bufferOutput = buffer.str(); \
                OMString exceptionString( bufferOutput ); \
                delete [] bufferOutput; \
                throw exceptionString; \
        } \
}
```

CHECK_EQUAL:

```
#define CHECK_EQUAL ( message, expected, actual ) \
{ \
        if ( !(expected == actual) ) \
        { \
                ostrstream buffer;\
                buffer << message << " expected: " << expected << ", actual: " << actual
<< ", File: " << __FILE__ << ", Line: " << __LINE__ << '\0'; \
                char* bufferOutput = buffer.str(); \
                OMString exceptionString( bufferOutput ); \
                delete [] bufferOutput; \
                throw exceptionString; \
        } \
}
```

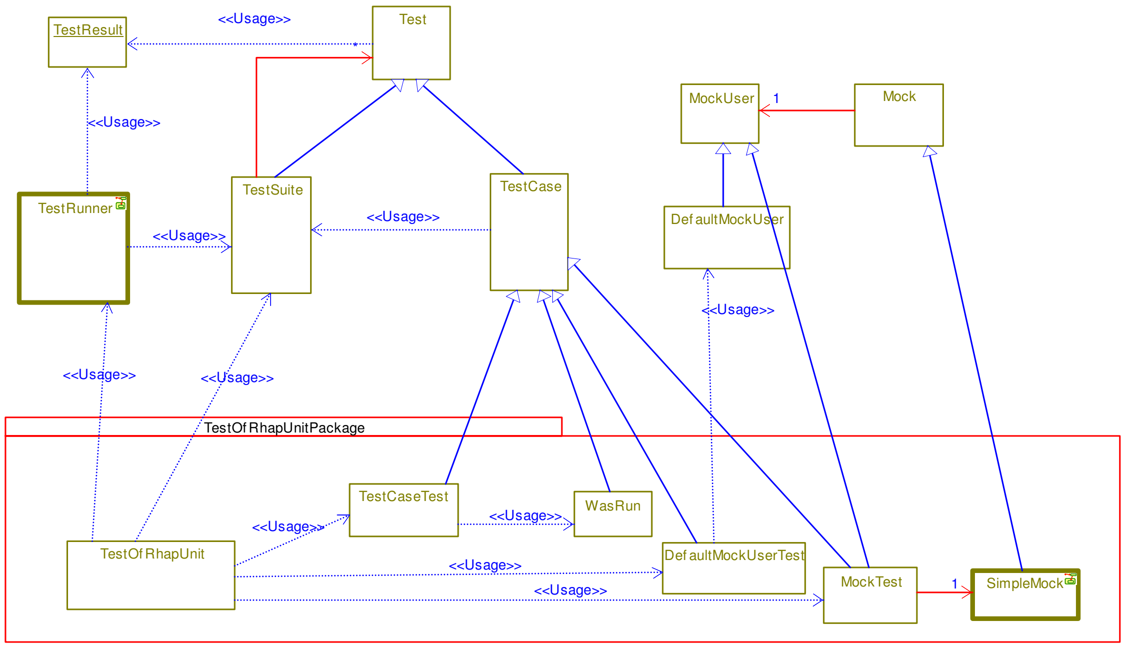
START_TIMER:

```
#define START_TIMER (tickUnitInMili) \
   OMTimerManager timerTimeCheckMacro(tickUnitInMili); \
   timerTimeCheckMacro.init(); \
   timeUnit firstTimeRecorded = timerTimeCheckMacro.getElapsedTime();
```

CHECK_TIME:

```
#define CHECK_TIME ( message, maxTimeValue ) \
   timeUnit ellapsedTimeAmount = timerTimeCheckMacro.getElapsedTime() -
firstTimeRecorded; \
        if ( !( maxTimeValue > ellapsedTimeAmount ) ) \
        { \
                ostrstream buffer; \
                buffer << message << " Ellapsed time is more than required.
maxTimeAllowed: " << maxTimeValue <<", actualEllapsedTime: " <<
ellapsedTimeAmount << ", File: " << __FILE__ << ", Line: " << __LINE__ << '\0'; \
                char* bufferOutput = buffer.str(); \
                OMString exceptionString( bufferOutput ); \
                delete [] bufferOutput; \
                throw exceptionString; \
        }
```

Object Model Diagrams



Contained Elements

Element Type	**Element Name**
Class	DefaultMockUser
Class	TestRunner
Class	TestResult
Class	MockUser
Class	Test
Class	Mock
Class	TestCase
Class	TestSuite
Package	TestOfRhapUnitPackage
Class	DefaultMockUserTest
Class	WasRun
Class	TestOfRhapUnit
Class	MockTest

Class	SimpleMock
Class	TestCaseTest
Relation	itsTest
Relation	itsSimpleMock
Relation	itsMockUser

TestCase Class

Generalization of Class TestCase

Name	**Base**	**Derived**
Test	Test	TestCase

Attributes of Class TestCase

Name	**Type**	**Visibility**
testName	OMString	protected

Operations of Class TestCase

Name	**Return Type**	**Visibility**
setUp	void	protected
_run	void	protected
tearDown	void	protected
run	void	public

public void run (TestResult & _result):

```
_result.testStarted();

try
{
	setUp();
} catch (...)
{
	_result.testFailed();
	cerr << "FAIL: " << testName << ", Unexpected exception in SetUp" << endl
<< endl; cerr.flush();
	return;
}

try
{
	_run();
} catch (OMString& message)
{
        _result.testFailed();
        cerr << "FAIL: " << testName << ", " << message << endl << endl; cerr.flush();
} catch (...)
{
        _result.testFailed();
        cerr << "FAIL: " << testName << ", Unexpected exception" << endl << endl;
cerr.flush();
}

try
{
        tearDown();
} catch (...)
{
        _result.testFailed();
        cerr << "FAIL: " << testName << ", Unexpected exception in TearDown" <<
endl << endl; cerr.flush();
        return;
}
```

TestResult Class

Attributes of Class TestResult

Name	Type	Visibility
runCount	int	protected
errorCount	int	protected

Operations of Class TestResult

Name	Return Type	Visibility
summary	OMString	public
testStarted	void	public
testFailed	void	public

public OMString summary():

```
ostrstream buffer;

buffer << runCount << " run, " << errorCount << " failed" << '\0';

char* str = buffer.str();
OMString sonuc( str );
delete [] str;

return sonuc;
```

public void testStarted():
runCount++;

public void testFailed():
errorCount++;

TestSuite Class

Generalization of Class TestSuite

Name	Base	Derived
Test	Test	TestSuite

Operations of Class TestSuite

Name	Return Type	Visibility
TestSuite		public
~TestSuite		public
getItsTest	OMIterator<Test*>	protected
clearItsTest	void	protected
add	void	public
run	void	public

public ~TestSuite():
```
clearItsTest();
cleanUpRelations();
```

protected OMIterator<Test*> getItsTest():
```
OMIterator<Test*> iter(itsTest);iter.reset();
return iter;
```

protected void clearItsTest():
```
OMIterator<class Test *>  tests = getItsTest();
while ( tests.value() != NULL)
{
        Test* temp = tests.value();
        delete temp;
        temp = NULL;
        tests++;
}
itsTest.removeAll();
```

public void add(Test * p_Test):

```
if (p_Test != NULL)
        itsTest.add(p_Test);
```

public void run(TestResult & _result):

```
OMIterator<class Test *>  tests = getItsTest();
while ( tests.value() != NULL)
{
        Test* temp = tests.value();
        temp->run(_result);
        tests++;
}
```

Relation information for Class TestSuite

Name	Inverse	Source	Target
itsTest		TestSuite	Test

Test Class

Operations of Class Test

Name	Return Type	Visibility
run	void	public

TestRunner Class

Attributes of Class TestRunner

Name	Type	Visibility
suite	TestSuite *	protected

Operations of Class TestRunner

Name	Return Type	Visibility
~TestRunner		public
runTests	void	public
add	void	public

public ~TestRunner():

```
delete suite;
```

public void runTests():

cout << "starting test" << endl << endl; cout.flush();

TestResult result;
suite->run(result);

cout << endl << result.summary() << endl; cout.flush();

public void add(Test * p_Test):
suite -> add(p_Test);

DefaultMockUser Class

Generalization of Class DefaultMockUser

Name	Base	Derived
MockUser	MockUser	DefaultMockUser

Attributes of Class DefaultMockUser

Name	Type	Visibility
actualMockCallCount	int	protected
expectedMockCalls	std::vector<OMString>	protected

Operations of Class DefaultMockUser

Name	Return Type	Visibility
actualMockCall	void	public
verify	void	public
DefaultMockUser		public
addExpectedMockCall	void	public

public void actualMockCall(const OMString & actualCall):
CHECK("More than expected mock call. call: " + actualCall, expectedMockCalls.size() > actualMockCallCount);
OMString actualCallNonConst(actualCall);
CHECK_EQUAL("Actual mock call is different from expected.", expectedMockCalls[actualMockCallCount], actualCallNonConst);

actualMockCallCount++;

public void verify():

CHECK_EQUAL("Less then expected mock call.", expectedMockCalls.size(), actualMockCallCount);

public DefaultMockUser():
expectedMockCalls.reserve(50);

public void addExpectedMockCall(const OMString & expectedCall):
expectedMockCalls.push_back(expectedCall);

Mock Class

Type information for Class Mock

```
enum MOCK_STATE { NO_LOGGING, EXPECTED, ACTUAL };

#define MOCK_START ( className, methodName ) \
        { \
                ostrstream bufferMockCall; \
                bufferMockCall << #className << "::" << #methodName;

#define MOCK_ADD_PARAM ( parameter ) \
                bufferMockCall << ", " << #parameter << ": " << parameter;

#define MOCK_END () \
                bufferMockCall << '\0'; \
                char* bufferOutput = bufferMockCall.str(); \
                mockCall( OMString( bufferOutput ) ); \
                delete [] bufferOutput; \
        }

#define MOCK_RETURN ( typeOfReturnValue ) \
        typeOfReturnValue * result = (typeOfReturnValue*)
        getPointerOfNextMockReturnValue();  \
        return *result ;
```

Attributes of Class Mock

Name	Type	Visibility
pointerOfReturnValues	std::vector<void*>	protected
returnValueIndex	int	protected
stateOfMock	int	protected

Operations of Class Mock

Name	**Return Type**	**Visibility**
Mock		public
startExpectation	void	public
mockCall	void	protected
startActual	void	public
addReturnValue	void	public
getPointerOfNextMockReturnValue	void *	protected
stopMockLogging	void	public
resetReturnValueIndex	void	protected
clearReturnValues	void	protected

public Mock():
```
stateOfMock = NO_LOGGING;
pointerOfReturnValues.reserve(50);
```

public void startExpectation():
```
stateOfMock = EXPECTED;
clearReturnValues();
```

protected void mockCall(const OMString& callValue):
```
CHECK( "Before using a Mock, Mock User must be setted.", itsMockUser != NULL );

switch ( stateOfMock ) {
        case NO_LOGGING: return;
        case EXPECTED:
                itsMockUser -> addExpectedMockCall ( callValue );
                return;
        case ACTUAL:
                itsMockUser -> actualMockCall ( callValue );
                return;
        default:
                return;
}
```

public void startActual():
```
CHECK_EQUAL("Before going to actual mode, mock must be in expected mode.",
stateOfMock, EXPECTED );
stateOfMock = ACTUAL;
resetReturnValueIndex();
```

public void addReturnValue(void * pointerOfReturnValue):
```
pointerOfReturnValues.push_back( pointerOfReturnValue );
```

protected void * getPointerOfNextMockReturnValue():

CHECK("Require more return value than entered return value.",
pointerOfReturnValues.size() > returnValueIndex);
void * nextReturn = pointerOfReturnValues[returnValueIndex];
returnValueIndex++;
return nextReturn;

public void stopMockLogging():
stateOfMock = NO_LOGGING;
clearReturnValues();

protected void resetReturnValueIndex():
returnValueIndex = 0;

protected void clearReturnValues():
pointerOfReturnValues.clear();
resetReturnValueIndex();

MockUser Class

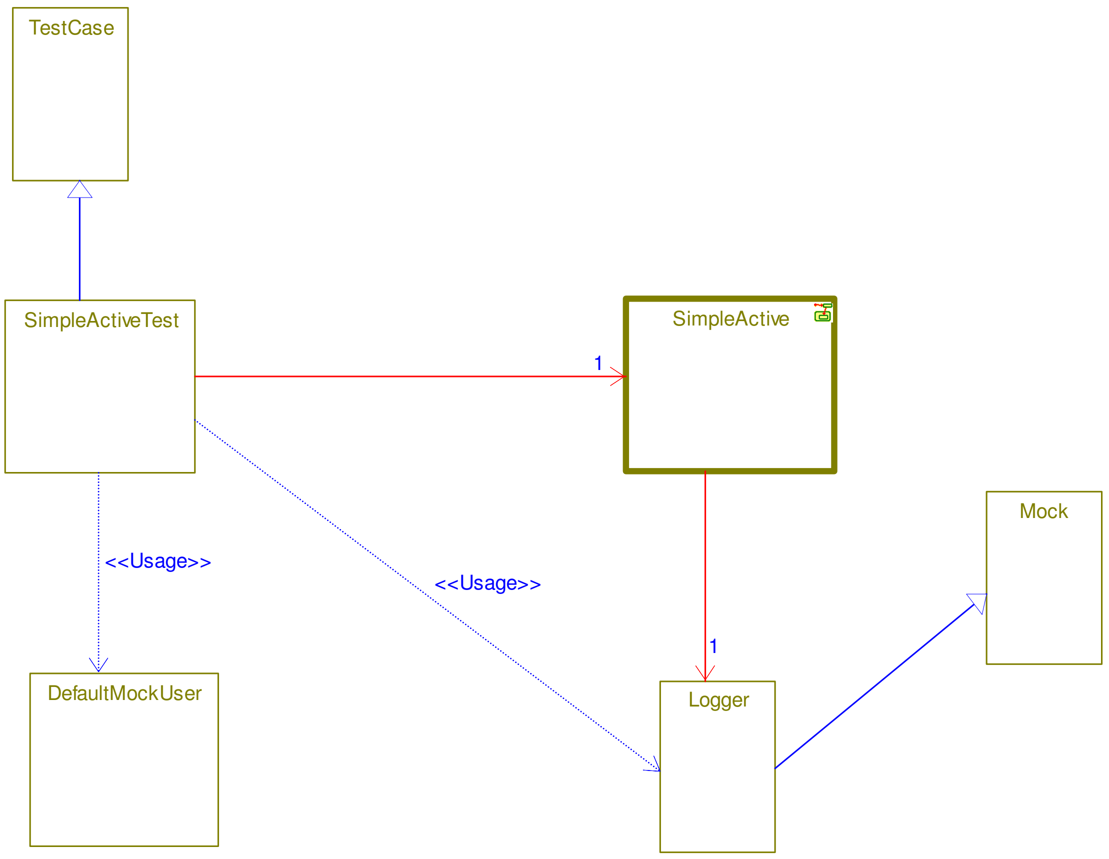
Operations of Class MockUser

Name	Return Type	Visibility
actualMockCall	void	public
addExpectedMockCall	void	public
Verify	void	public

ExampleTestOfActiveClass Package

Object Model Diagrams

ActiveClassTest Object Model Diagram



Contained Elements

Element Type	Element Name
Class	Mock
Class	DefaultMockUser
Class	SimpleActiveTest
Class	Logger
Class	SimpleActive
Class	TestCase
Relation	itsLogger
Relation	itsSimpleActive

SimpleActive Class

Attributes of Class SimpleActive

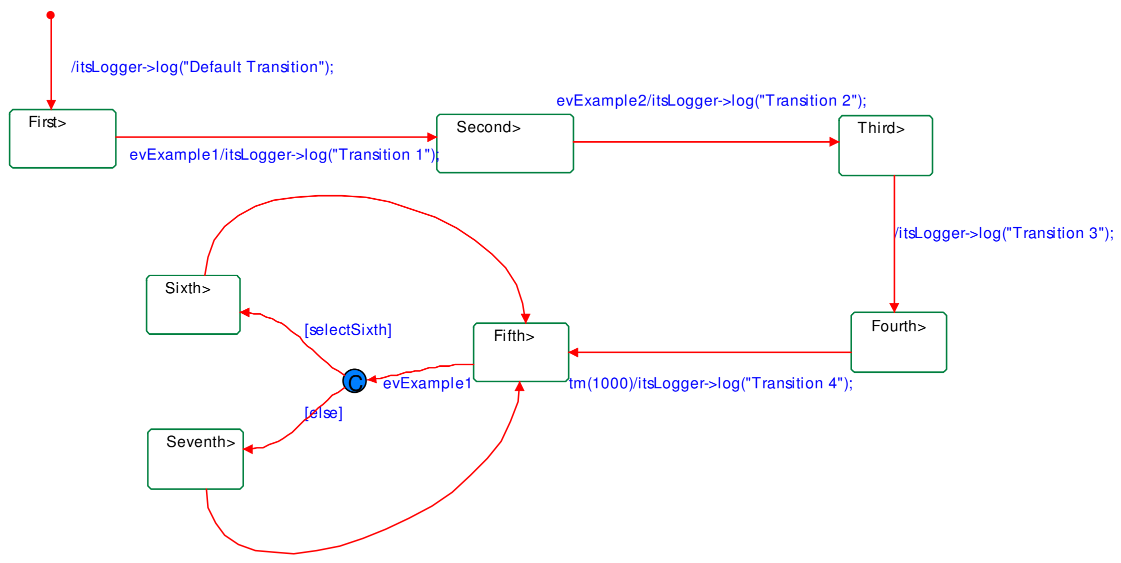
Name	Type	Visibility
selectSixth	OMBoolean	public

Relation information for Class SimpleActive

Name	Inverse	Source	Target

itsLogger	SimpleActive	Logger

Statechart information for Class:SimpleActive



/itsLogger->log("Default Transition");
First>
evExample1/itsLogger->log("Transition 1");
Second>
evExample2/itsLogger->log("Transition 2");
Third>
/itsLogger->log("Transition 3");
Fourth>
tm(1000)/itsLogger->log("Transition 4");
Fifth>
evExample1
C
[selectSixth]
[else]
Sixth>
Seventh>

Contained Elements

Element Type	Element Name
Event	evExample1
Event	evExample2
State	Third
State	Fifth
State	Second
State	Fourth
State	Seventh
State	Sixth
State	First

SimpleActiveTest Class

Generalization of Class SimpleActiveTest

Name	Base	Derived
TestCase	TestCase	SimpleActiveTest

Attributes of Class SimpleActiveTest

Name	Type	Visibility
mockUser	DefaultMockUser *	protected

logger	Logger *	protected

Operations of Class SimpleActiveTest

Name	Return Type	Visibility
createTests	Test *	public
testStartBehaviour	void	public
testTransitionWithEvent	void	public
setUp	void	protected
tearDown	void	protected
testReactionInState	void	public
testTransitionWithoutEvent	void	public
testTimeout	void	public
testCondition	void	public

static public Test * createTests():

```
TestSuite * suite = new TestSuite;

ADD_TEST(SimpleActiveTest, testStartBehaviour, suite );
ADD_TEST(SimpleActiveTest, testTransitionWithEvent, suite );
ADD_TEST(SimpleActiveTest, testReactionInState, suite );
ADD_TEST(SimpleActiveTest, testTransitionWithoutEvent, suite );
ADD_TEST(SimpleActiveTest, testTimeout, suite );
ADD_TEST(SimpleActiveTest, testCondition, suite );

return suite;
```

public void testStartBehaviour():

```
logger->startExpectation();
logger->log("Default Transition");
logger->log("First State Entry");

logger->startActual();

itsSimpleActive->rootState_entDef();

mockUser->verify();
```

public void testTransitionWithEvent():

```
//goto first state
itsSimpleActive->rootState_entDef();
```

logger->startExpectation();
logger->log("First State Exit");
logger->log("Transition 1");
logger->log("Second State Entry");

logger->startActual();

evExample1 ev1;
itsSimpleActive->takeEvent(&ev1);

mockUser->verify();

protected void setUp():
itsSimpleActive = new SimpleActive();
logger = new Logger();
mockUser = new DefaultMockUser();
logger->setItsMockUser(mockUser);
itsSimpleActive -> setItsLogger(logger);

protected void tearDown():
delete itsSimpleActive;
delete logger;
delete mockUser;

public void testReactionInState():

//goto second state
itsSimpleActive->rootState_entDef();
evExample1 ev1;
itsSimpleActive->takeEvent(&ev1);

logger->startExpectation();
logger->log("Second State Reaction In State");

logger->startActual();

evExample1 ev2;
itsSimpleActive->takeEvent(&ev2);

mockUser->verify();

public void testTransitionWithoutEvent():

//goto second state
itsSimpleActive->rootState_entDef();

```
evExample1 ev1;
itsSimpleActive->takeEvent(&ev1);

logger->startExpectation();
logger->log("Second State Exit");
logger->log("Transition 2");
logger->log("Third State Entry");
logger->log("Third State Exit");
logger->log("Transition 3");
logger->log("Fourth State Entry");

logger->startActual();

evExample2 ev2;
itsSimpleActive->takeEvent(&ev2);

mockUser->verify();
```

public void testTimeout():

```
//goto Fourth state
itsSimpleActive->rootState_entDef();
evExample1 ev1;
itsSimpleActive->takeEvent(&ev1);
evExample2 ev2;
itsSimpleActive->takeEvent(&ev2);

logger->startExpectation();
logger->log("Fourth State Exit");
logger->log("Transition 4");
logger->log("Fifth State Entry");

logger->startActual();

OMTimeout timeOut(SimpleActive_Timeout_Fourth_id, itsSimpleActive, 1000,
"ROOT.Fourth"); //this values can be found by looking at SimpleActive code.
itsSimpleActive->takeEvent(&timeOut);

mockUser->verify();
```

public void testCondition():

```
//goto Fifth state
itsSimpleActive->rootState_entDef();
```

```
evExample1 ev1;
itsSimpleActive->takeEvent(&ev1);
evExample2 ev2;
itsSimpleActive->takeEvent(&ev2);
OMTimeout timeOut(SimpleActive_Timeout_Fourth_id, itsSimpleActive, 1000,
"ROOT.Fourth");
itsSimpleActive->takeEvent(&timeOut);

logger->startExpectation();
logger->log("Fifth State Exit");
logger->log("Sixth State Entry");
logger->log("Sixth State Exit");
logger->log("Fifth State Entry");
logger->log("Fifth State Exit");
logger->log("Seventh State Entry");
logger->log("Seventh State Exit");
logger->log("Fifth State Entry");

logger->startActual();

itsSimpleActive->setSelectSixth( TRUE );
evExample1 ev3;
itsSimpleActive->takeEvent(&ev3);
itsSimpleActive->setSelectSixth( FALSE );
evExample1 ev4;
itsSimpleActive->takeEvent(&ev4);

mockUser->verify();
```

Relation information for Class SimpleActiveTest

Name	Inverse	Source	Target
itsSimpleActive		SimpleActiveTest	SimpleActive

Logger Class

Generalization of Class Logger

Name	Base	Derived
Mock	Mock	Logger

Operations of Class Logger

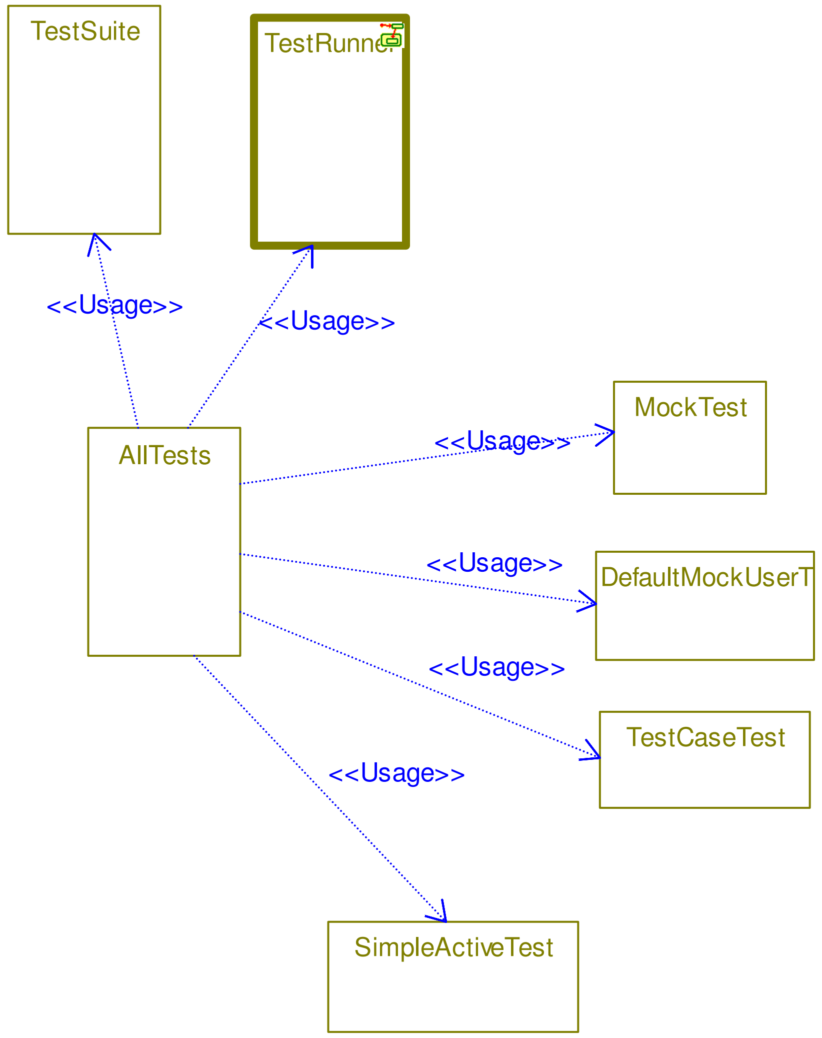
Name	Return Type	Visibility
log	void	public

public void log(const OMString & logValue):
MOCK_START(Logger, log);
MOCK_ADD_PARAM(logValue);
MOCK_END();

AllTestPackage Package

Object Model Diagrams

AllTest Object Model Diagram



Contained Elements

Element Type	Element Name
Class	AllTests

Class	TestSuite
Class	TestCaseTest
Class	MockTest
Class	DefaultMockUserTest
Class	TestRunner
Class	SimpleActiveTest

AllTests Class

Operations of Class AllTests

Name	Return Type	Visibility
AllTests		public

public AllTests():

```
TestSuite * suite = new TestSuite();
suite -> add( TestCaseTest::createTests() );
suite -> add( DefaultMockUserTest::createTests() );
suite -> add( MockTest::createTests() );
suite -> add( SimpleActiveTest::createTests() );
TestRunner::add(suite);
```

TestOfRhapUnitPackage Package

Type information for Package TestOfRhapUnitPackage

StructExample:

```
struct StructExample { int intValue; float floatValue; };
```

Operation information for Package TestOfRhapUnitPackage

ostream & operator<< (ostream & out, const struct StructExample & structInput)

```
return ( out << "intValue: " << structInput.intValue << ", " << "floatValue: " <<
structInput.floatValue );
```

WasRun Class

Generalization of Class WasRun

Name	Base	Derived
TestCase	TestCase	WasRun

Attributes of Class WasRun

Name	Type	Visibility
log	OMString	public

Operations of Class WasRun

Name	Return Type	Visibility
testMethod	void	public
setUp	void	public
tearDown	void	public
testBroken	void	public
testSucceeded	void	public

public void testMethod():
log += "TestMethod ";

protected void setUp():
log += "SetUp ";

protected void tearDown():
log += "TearDown ";

public void testBroken():
throw OMString("Ignore this fail report");

TestCaseTest Class

Generalization of Class TestCaseTest

Name	Base	Derived
TestCase	TestCase	TestCaseTest

Attributes of Class TestCaseTest

Name	Type	Visibility
result	TestResult	public

Operations of Class TestCaseTest

Name	Return Type	Visibility
testTemplateMethod	void	public
testFailedResultFor	void	public

matting		
testSuite	void	public
testTryCatch	void	public
testFailedTest	void	public
testSucceeded	void	public
createTests	Test *	public
testCheckMacro	void	public
testTimeCheck	void	public

public void testTemplateMethod():

```
TestCase * test;
CREATE_TEST( WasRun, testMethod, test);
WasRun * testWasRun = (WasRun *) test;

assert( testWasRun->getLog() == "" );
test->run(result);
assert( testWasRun->getLog() == "SetUp TestMethod TearDown " );

delete test;
```

public void testFailedResultFormatting():

```
result.testStarted();
result.testFailed();
assert( "1 run, 1 failed" == result.summary() );
```

public void testSuite():

```
TestSuite * suite = new TestSuite();

ADD_TEST(TestCaseTest, testFailedResultFormatting, suite );
ADD_TEST(TestCaseTest, testTemplateMethod, suite );
ADD_TEST(TestCaseTest, testSucceeded, suite );

suite->run(result);
assert("3 run, 0 failed" == result.summary());

delete suite;
```

public void testTryCatch():

```
try
{
        int i = 6;
        throw i;
        assert( FALSE );
} catch (int num)
{
        assert( 6 == num );
}
```

public void testFailedTest():

```
TestCase * test;
CREATE_TEST( WasRun, testBroken, test);

test->run(result);
CHECK( "", "1 run, 1 failed" == result.summary() );

delete test;
```

public void testSucceeded():

```
TestCase * test;
CREATE_TEST( WasRun, testSucceeded, test);

test->run(result);
CHECK( "", "1 run, 0 failed" == result.summary() );

delete test;
```

static public Test * createTests():

```
TestSuite * suite = new TestSuite;

ADD_TEST(TestCaseTest, testFailedResultFormatting, suite );
ADD_TEST(TestCaseTest, testTemplateMethod, suite );
ADD_TEST(TestCaseTest, testSucceeded, suite );
ADD_TEST(TestCaseTest, testSuite, suite );
ADD_TEST(TestCaseTest, testTryCatch, suite );
ADD_TEST(TestCaseTest, testCheckMacro, suite );
ADD_TEST(TestCaseTest, testFailedTest, suite );
ADD_TEST(TestCaseTest, testTimeCheck, suite );

return suite;
```

public void testCheckMacro():

```
try
{
        CHECK("",TRUE);
}catch(OMString message)
{
        assert( FALSE );
}catch(...){
        assert( FALSE );
}
```

```
try
{
	CHECK("",FALSE);
	assert( FALSE );
}catch(OMString message)
{
}catch(...){
	assert( FALSE );
}
```

public void testTimeCheck():

```
OMBoolean isExceptionThrown = FALSE;

try
{
	START_TIMER(20);
	OXFTDelay(100);
	CHECK_TIME("", 79);
} catch (OMString &)
{
	isExceptionThrown = TRUE;
}
CHECK( "Time control must throw an exception.", isExceptionThrown );

isExceptionThrown = FALSE;
try
{
	START_TIMER(20);
	OXFTDelay(100);
	CHECK_TIME("", 121);
} catch (OMString &)
{
	isExceptionThrown = TRUE;
}
CHECK( "Time control shouldn't throw an exception.", !isExceptionThrown );
```

Relation information for Class TestCaseTest

TestOfRhapUnit Class

Operations of Class TestOfRhapUnit

Name	Return Type	Visibility
TestOfRhapUnit		public
~TestOfRhapUnit		public

public TestOfRhapUnit():

```
TestSuite * suite = new TestSuite();
suite -> add( TestCaseTest::createTests() );
suite -> add( DefaultMockUserTest::createTests() );
suite -> add( MockTest::createTests() );
TestRunner::add(suite);
```

Relation information for Class TestOfRhapUnit

DefaultMockUserTest Class

Generalization of Class DefaultMockUserTest

Name	Base	Derived
TestCase	TestCase	DefaultMockUserTest

Attributes of Class DefaultMockUserTest

Name	Type	Visibility
mockUser	DefaultMockUser *	protected

Operations of Class DefaultMockUserTest

Name	Return Type	Visibility
tearDown	void	protected
setUp	void	protected
createTests	Test *	public
testFailLessCall	void	public
testFailWrongCall	void	public
testSucceededForOne	void	public
testSucceededForMany	void	public
testVector	void	public
testFailMuchCall	void	public

protected void tearDown():

```
delete mockUser;
```

protected void setUp():

```
mockUser = new DefaultMockUser;
```

static public Test * createTests():

```
TestSuite * suite = new TestSuite;
```

```
ADD_TEST(DefaultMockUserTest, testFailLessCall, suite );
ADD_TEST(DefaultMockUserTest, testFailWrongCall, suite );
ADD_TEST(DefaultMockUserTest, testSucceededForOne, suite );
ADD_TEST(DefaultMockUserTest, testSucceededForMany, suite );
ADD_TEST(DefaultMockUserTest, testVector, suite );
ADD_TEST(DefaultMockUserTest, testFailMuchCall, suite );

return suite;
```

public void testFailLessCall():

```
mockUser -> addExpectedMockCall( "XXX:YYY" );
OMBoolean isContinueAfterVerify = FALSE;
OMBoolean isVerifyThrowException = FALSE;

try
{
        mockUser -> verify();
        isContinueAfterVerify = TRUE;
} catch(OMString &)
{
        isVerifyThrowException = TRUE;
}

CHECK( "", !isContinueAfterVerify );
CHECK( "", isVerifyThrowException );
```

public void testFailWrongCall():

```
mockUser -> addExpectedMockCall( "XXX:YYY" );
OMBoolean isContinueAfterActualCall = FALSE;
OMBoolean isActualCallThrowException = FALSE;

try
{
        mockUser -> actualMockCall( "XXZZ" );
        isContinueAfterActualCall = TRUE;
} catch(...)
{
        isActualCallThrowException = TRUE;
}

CHECK( "", !isContinueAfterActualCall );
CHECK( "", isActualCallThrowException );
```

public void testSucceededForOne():

```
mockUser -> addExpectedMockCall( "XXX:YYY" );

try
{
        mockUser -> actualMockCall( "XXX:YYY" );
        mockUser -> verify();
} catch(...)
{
        CHECK("", FALSE);
}
```

public void testSucceededForMany():

```
mockUser -> addExpectedMockCall( "Call 1" );
mockUser -> addExpectedMockCall( "Call 2" );
mockUser -> addExpectedMockCall( "Call 3" );

try
{
        mockUser -> actualMockCall( "Call 1" );
        mockUser -> actualMockCall( "Call 2" );
        mockUser -> actualMockCall( "Call 3" );
        mockUser -> verify();
} catch(...)
{
        CHECK( "", FALSE );
}
```

public void testVector():

```
std::vector<OMString> sentence;
sentence.reserve(50);

sentence.push_back("Hello,");
sentence.push_back("how");

CHECK_EQUAL( "", "Hello,", sentence[0] );
CHECK_EQUAL( "", "how", sentence[1] );
CHECK_EQUAL( "", 2, sentence.size() );
```

public void testFailMuchCall():

```
mockUser -> addExpectedMockCall( "Call 1" );
mockUser -> addExpectedMockCall( "Call 2" );

OMBoolean isExceptionThrown = FALSE;
try
{
        mockUser -> actualMockCall( "Call 1" );
        mockUser -> actualMockCall( "Call 2" );
        mockUser -> actualMockCall( "Call 3" );
} catch(...)
{
        isExceptionThrown = TRUE;
}

CHECK( "mockUser must be throw exception", isExceptionThrown );
```

Relation information for Class DefaultMockUserTest

MockTest Class

Generalization of Class MockTest

Name	Base	Derived
TestCase	TestCase	MockTest
MockUser	MockUser	MockTest

Attributes of Class MockTest

Name	Type	Visibility
actualCall	OMString	protected
expectedCall	OMString	protected

Operations of Class MockTest

Name	Return Type	Visibility
testExpectedMacro	void	public
addExpectedMockCall	void	public
verify	void	public
actualMockCall	void	public
setUp	void	protected
testActualMacro	void	public
testEvent	void	public

tearDown	void	protected
testSetingReturnValues	void	public
createTests	Test *	public
testStartStopExecution	void	public
testUserDefinedType	void	public
testReturnsAccordingToModes	void	public
testSuccessfulModeSequence	void	public
testFailedModeSequence	void	public

public void testExpectedMacro():
itsSimpleMock -> setItsMockUser(this);
itsSimpleMock -> startExpectation();
itsSimpleMock -> simpleOperation(5, 4.75);

CHECK_EQUAL("", "SimpleMock::simpleOperation, param1: 5, param2: 4.75", expectedCall);
CHECK_EQUAL("", "", actualCall);

public void addExpectedMockCall(const OMString & expectedCall):
this->expectedCall = expectedCall;

public void actualMockCall(const OMString & actualCall):
this->actualCall = actualCall;

protected void setUp():
itsSimpleMock = new SimpleMock();

public void testActualMacro():

itsSimpleMock -> setItsMockUser(this);
itsSimpleMock -> startExpectation();
itsSimpleMock -> startActual();
itsSimpleMock -> simpleOperation(5, 4.75);

CHECK_EQUAL("", "SimpleMock::simpleOperation, param1: 5, param2: 4.75", actualCall);
CHECK_EQUAL("", "", expectedCall);

public void testEvent():

```
itsSimpleMock -> setItsMockUser( this );
itsSimpleMock -> startExpectation();
itsSimpleMock -> GEN( evExample(5,4.75) );

CHECK_EQUAL( "", "SimpleMock::evExample, ev->param1: 5, ev->param2: 4.75",
expectedCall );
```

protected void tearDown():

```
delete itsSimpleMock;
```

public void testSetingReturnValues():

```
itsSimpleMock -> setItsMockUser( this );

itsSimpleMock -> startExpectation();

int expectedIntReturn = 65;
itsSimpleMock -> addReturnValue( &expectedIntReturn );
itsSimpleMock -> simpleReturnInt();

char* expectedCharPReturn = "char first ";
itsSimpleMock -> addReturnValue( &expectedCharPReturn );
itsSimpleMock -> simpleReturnCharP();

itsSimpleMock -> simpleOperation( 5, 4.75 );

char* expectedCharPReturn2 = "char second";
itsSimpleMock -> addReturnValue( &expectedCharPReturn2 );
itsSimpleMock -> simpleReturnCharP();

itsSimpleMock -> startActual();
CHECK_EQUAL( "Return value", expectedIntReturn, itsSimpleMock ->
simpleReturnInt() );
CHECK_EQUAL( "Return value", expectedCharPReturn , itsSimpleMock ->
simpleReturnCharP() );

itsSimpleMock -> simpleOperation( 5, 4.75 );

CHECK_EQUAL( "Return value", expectedCharPReturn2 , itsSimpleMock ->
simpleReturnCharP() );
```

static public Test * createTests():

```
TestSuite * suite = new TestSuite;
```

```
ADD_TEST(MockTest, testActualMacro, suite );
ADD_TEST(MockTest, testEvent, suite );
ADD_TEST(MockTest, testExpectedMacro, suite );
ADD_TEST(MockTest, testSetingReturnValues, suite );
ADD_TEST(MockTest, testStartStopExecution, suite );
ADD_TEST(MockTest, testUserDefinedType, suite );
ADD_TEST(MockTest, testReturnsAccordingToModes, suite );
ADD_TEST(MockTest, testSuccessfulModeSequence, suite );
ADD_TEST(MockTest, testFailedModeSequence, suite );

return suite;
```

public void testStartStopExecution():

```
itsSimpleMock -> setItsMockUser( this );

itsSimpleMock -> simpleOperation( 5, 4.75 );
CHECK_EQUAL( "", "", expectedCall );
CHECK_EQUAL( "", "", actualCall );

itsSimpleMock -> startExpectation();
itsSimpleMock -> stopMockLogging();
itsSimpleMock -> simpleOperation( 5, 4.75 );
CHECK_EQUAL( "", "", expectedCall );
CHECK_EQUAL( "", "", actualCall );


itsSimpleMock -> startExpectation();
itsSimpleMock -> startActual();
itsSimpleMock -> stopMockLogging();
itsSimpleMock -> simpleOperation( 5, 4.75 );
CHECK_EQUAL( "", "", expectedCall );
CHECK_EQUAL( "", "", actualCall );
```

public void testUserDefinedType():

```
itsSimpleMock -> setItsMockUser( this );

itsSimpleMock -> startExpectation();

StructExample structExample;
structExample.intValue = 5;
structExample.floatValue = 4.75;
itsSimpleMock -> simpleOperationWithUserDefinedTypeInput( structExample );
```

```
CHECK_EQUAL( "", "SimpleMock::simpleOperationWithUserDefinedTypeInput,
structInput: intValue: 5, floatValue: 4.75", expectedCall );
```

public void testReturnsAccordingToModes():

```
itsSimpleMock -> setItsMockUser( this );

int expectedIntReturn = 65;
itsSimpleMock -> addReturnValue( &expectedIntReturn );
CHECK_EQUAL("Return facility must be run in NoLogging mode also.",
expectedIntReturn, itsSimpleMock -> simpleReturnInt() );

itsSimpleMock -> startExpectation();
int expectedIntReturn2 = 80;
itsSimpleMock -> addReturnValue( &expectedIntReturn2 );
CHECK_EQUAL("When entering the expectation mode, return values must be
reinitialized.", expectedIntReturn2, itsSimpleMock -> simpleReturnInt() );

itsSimpleMock -> startActual();
CHECK_EQUAL("Return values that are entered in expectation mode must be used for
actual mode.", expectedIntReturn2, itsSimpleMock -> simpleReturnInt() );
```

public void testSuccessfulModeSequence():

```
OMBoolean isExceptionThrown = FALSE;

try
{
        itsSimpleMock -> startExpectation();
        itsSimpleMock -> startActual();
        itsSimpleMock -> stopMockLogging();
} catch (...)
{
        isExceptionThrown = TRUE;
}

CHECK("Above sequence is wright. Exception should not be thrown.",
!isExceptionThrown );
```

public void testFailedModeSequence():

```
OMBoolean isExceptionThrown = FALSE;

isExceptionThrown = FALSE;
try
{
        itsSimpleMock -> startActual();
} catch (...)
```

```
{
	isExceptionThrown = TRUE;
}

CHECK("Before going to actual mode, mock must be in expected mode.",
isExceptionThrown );
```

Relation information for Class MockTest

Name	Inverse	Source	Target
itsSimpleMock		MockTest	SimpleMock

SimpleMock Class

Generalization of Class SimpleMock

Name	Base	Derived
Mock	Mock	SimpleMock

Operations of Class SimpleMock

Name	Return Type	Visibility
simpleOperation	void	public
gen	OMBoolean	public
simpleReturnInt	int	public
simpleReturnCharP	char *	public
simpleOperationWithUserDefinedTypeInput	void	public

public void simpleOperation(int param1, float param2):

```
MOCK_START( SimpleMock, simpleOperation );
MOCK_ADD_PARAM( param1 );
MOCK_ADD_PARAM( param2 );
MOCK_END();
```

public OMBoolean gen(OMEvent * event, OMBoolean genFromISR):

```
if ( event -> isTypeOf(
evExample_TestOfRhapUnitPackage_RhapUnitTestFramework_id ) )
{
	evExample * ev = ((evExample *) event );
```

```
        MOCK_START( SimpleMock, evExample );
        MOCK_ADD_PARAM( ev->param1 );
        MOCK_ADD_PARAM( ev->param2 );
        MOCK_END();
}

if (!genFromISR) delete event;
return true;
```

public int simpleReturnInt():

```
MOCK_START( SimpleMock, simpleReturnInt );
MOCK_END();

MOCK_RETURN( int );
```

public char * simpleReturnCharP():

```
MOCK_START( SimpleMock, simpleReturnCharP );
MOCK_END();

MOCK_RETURN( char* );
```

public void simpleOperationWithUserDefinedTypeInput(const struct StructExample & structInput):

```
MOCK_START( SimpleMock, simpleOperationWithUserDefinedTypeInput );
MOCK_ADD_PARAM( structInput );
MOCK_END();
```

APPENDIX D

RhapUnit TUTORIAL

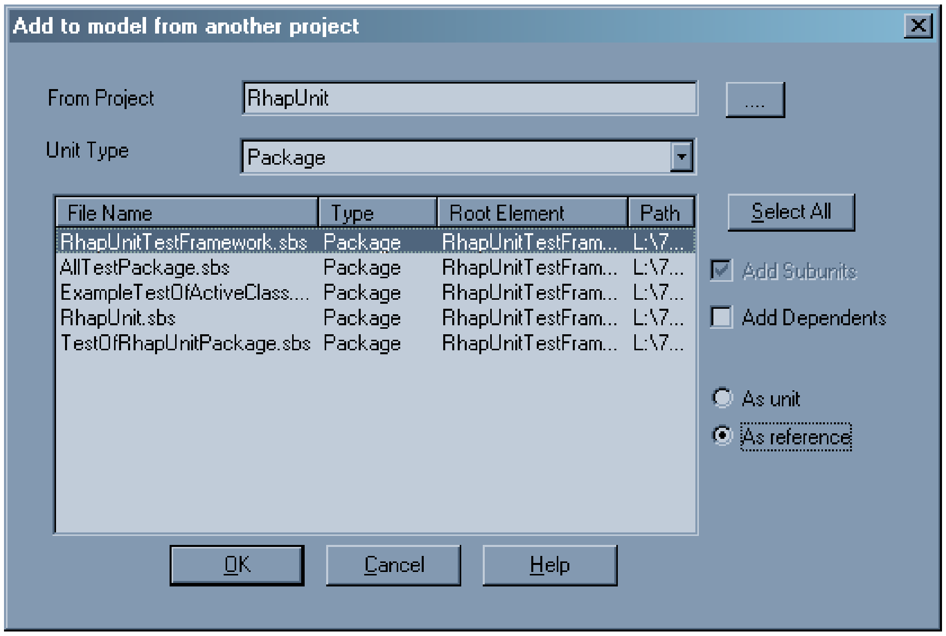
RhapUnit is a combination of xUnit framework and Mock functionalities. It is developed for Rhapsocy C++ from I-Logix. Its main features are developed according to xUnit pattern from the Kent Beck's book "Test Driven Development By Example". CREATE_TEST macro is mainly got from cppUnitLite from ObjectMentor. In this tutorial using of RhapUnit is shown step by step. You can also use self tests of RhapUnit as help documentation. I recommend looking at self tests of RhapUnit firstly.

The abilities of RhapUnit are followed:

- xUnit properties:
    - Check conditions.
    - Running tests automatically.
    - Isolation of tests from each other.
    - Reporting fail conditions
    - Reporting total test result
    - Grouping tests
- Mock properties:
    - Easy writing of mock objects
    - Easy way of setting expectation from mock objects
    - Easy way of setting return values of mock objects
    - Verifying that whether the expectations are same as actual calls or not.
- Checking time criterions.
- Testing active objects sequentially:
    - Starting behavior of active objects
    - Sending event to the active objects
    - Creating timeouts to the active objects

Importing RhapUnit

First step is selecting “add to model” from “file” menu in Rhapsody. As it is seen from the Figure 15, the next step is importing RhapUnitTestFramework from RhapUnit project. As it is seen in Figure 16, after this import operation the RhapUnit packages should be seen in the browser of the Rhapsody.


Add to model from another project

From Project: RhapUnit

Unit Type: Package

File Name	Type	Root Element	Path
RhapUnitTestFramework.sbs	Package	RhapUnitTestFram...	L:\7...
AllTestPackage.sbs	Package	RhapUnitTestFram...	L:\7...
ExampleTestOfActiveClass....	Package	RhapUnitTestFram...	L:\7...
RhapUnit.sbs	Package	RhapUnitTestFram...	L:\7...
TestOfRhapUnitPackage.sbs	Package	RhapUnitTestFram...	L:\7...

☑ Add Subunits
☐ Add Dependents

○ As unit
● As reference

Figure 15: Importing RhapUnitTestFramework.

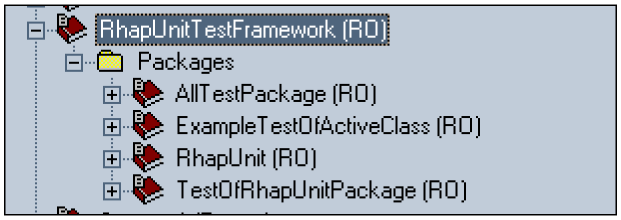
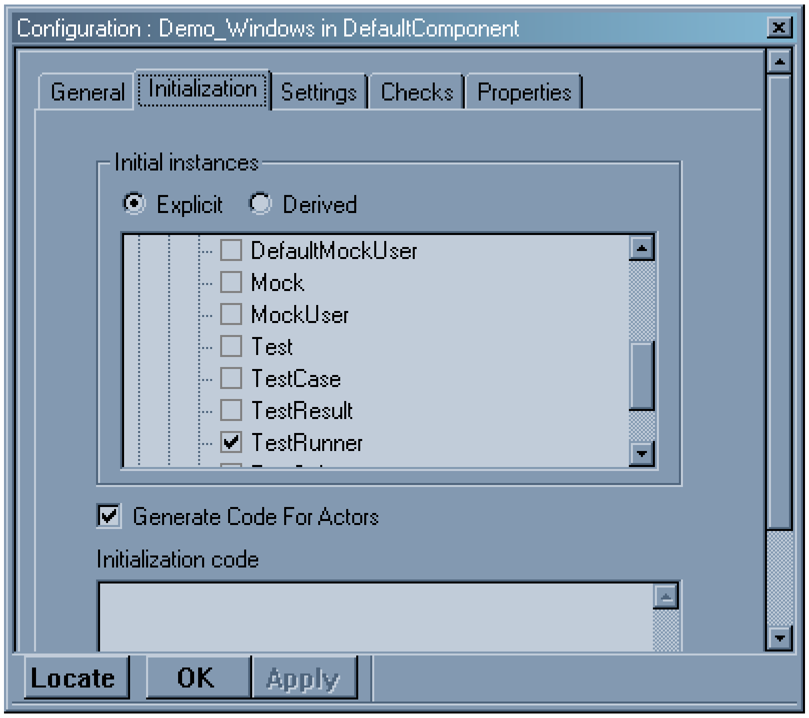

```
RhapUnitTestFramework (RO)
  Packages
    AllTestPackage (RO)
    ExampleTestOfActiveClass (RO)
    RhapUnit (RO)
    TestOfRhapUnitPackage (RO)
```

Figure 16: The imported packages of RhapUnit in browser.

Creating Configuration

After importing RhapUnit, you should create your test configuration. In the initialization, select TestRunner from the RhapUnit package of RhapUnitTestFramework package. Adding string stream by including "strstream.h" to the standard headers. But for windows configurations, you should add "strstrea.h" instead of "strstream.h". These steps are shown in Figure 17 and Figure 18.



Initial instances

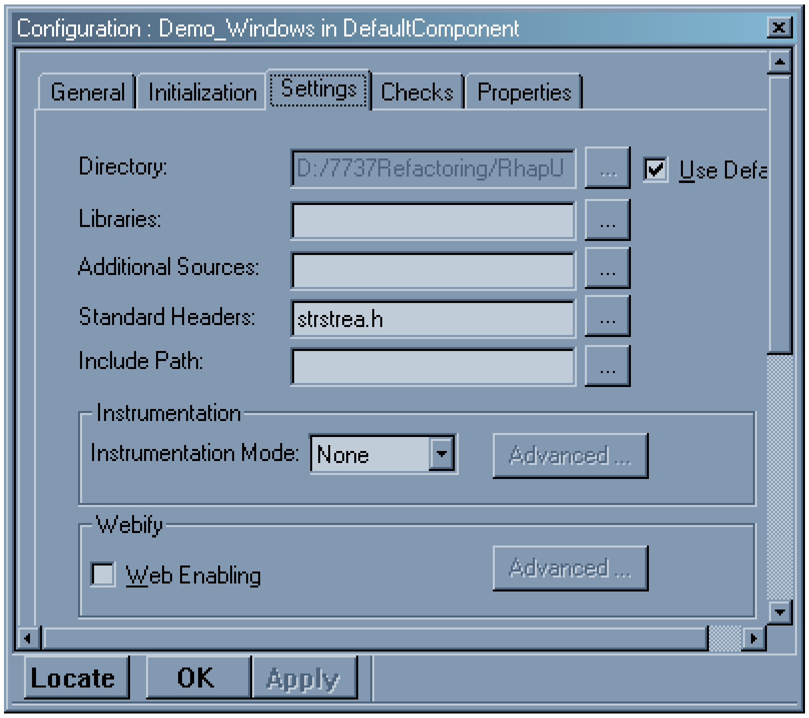
(● Explicit) ○ Derived

- ☐ DefaultMockUser
- ☐ Mock
- ☐ MockUser
- ☐ Test
- ☐ TestCase
- ☐ TestResult
- ☑ TestRunner

☑ Generate Code For Actors

Initialization code

Figure 17: Selecting TestRunner.


Configuration : Demo_Windows in DefaultComponent

Directory: D:/7737Refactoring/RhapU ☑ Use Defa

Libraries:

Additional Sources:

Standard Headers: strstrea.h

Include Path:

Instrumentation

Instrumentation Mode: None

Webify

☐ Web Enabling

Figure 18: Including string stream.

Creating All Test Package

To register tests to the TestRunner, you may create an AllTests package and AllTestRegistor class. You should create dependencies from AllTests to TestRunner and TestSuite classes under package RhapUnit. After that you should select AllTestRegistor for initialization. These package and dependencies are shown in Figure 19 and Figure 20.

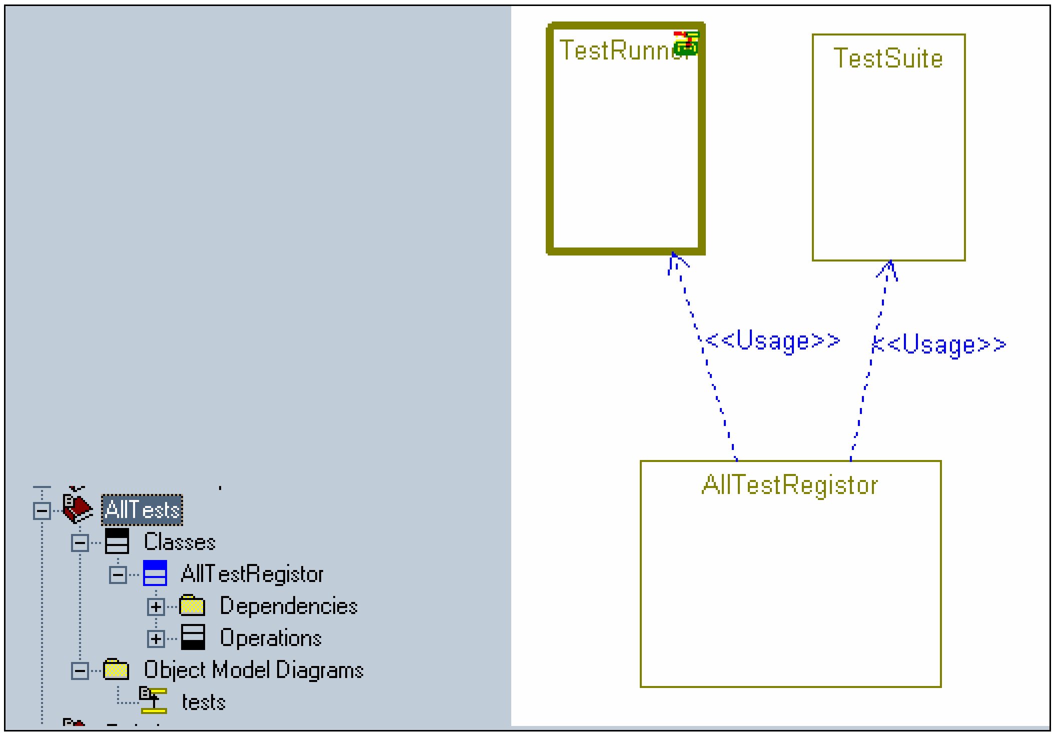
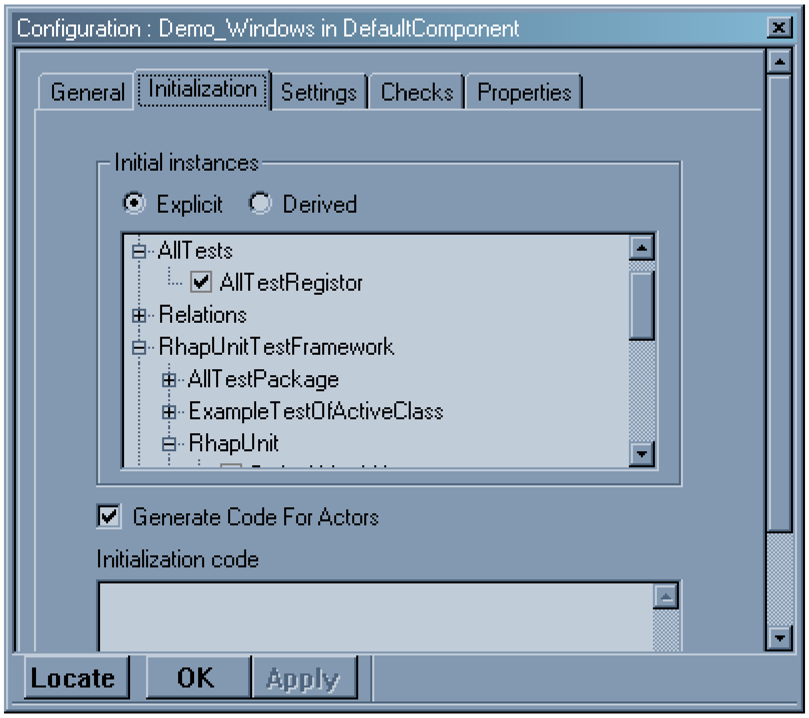


Figure 19: Package AllTests and its dependencies



Initial instances

[✓ Explicit] Derived

- AllTests
  - ✓ AllTestRegistor
- Relations
- RhapUnitTestFramework
  - AllTestPackage
  - ExampleTestOfActiveClass
  - RhapUnit

✓ Generate Code For Actors

Initialization code

Figure 20: Inserting AllTestRegistor.

Testing Sequential Object

Following sections describe how a sequential object is tested. The tested sequential object is shown in Figure 21. SequentialDemo has relations with two classes. These are SequentialRelation which is a sequential object, and ActiveRelation which is an active object.

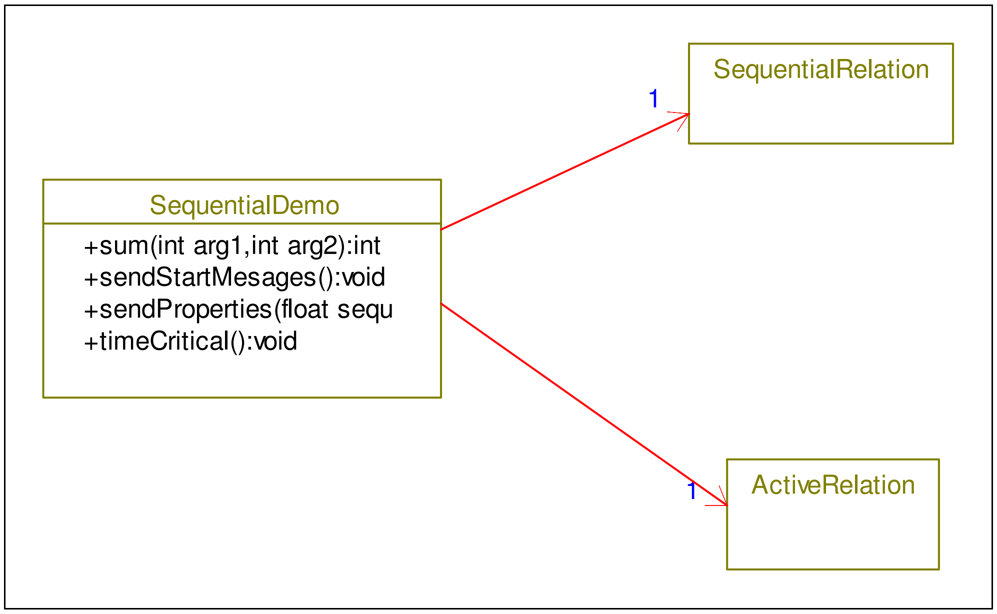


Figure 21: Tested sequential object, SequentialDemo.

SequentialDemoTest Class

SequentialDemoTest is the test class of SequentialDemo.

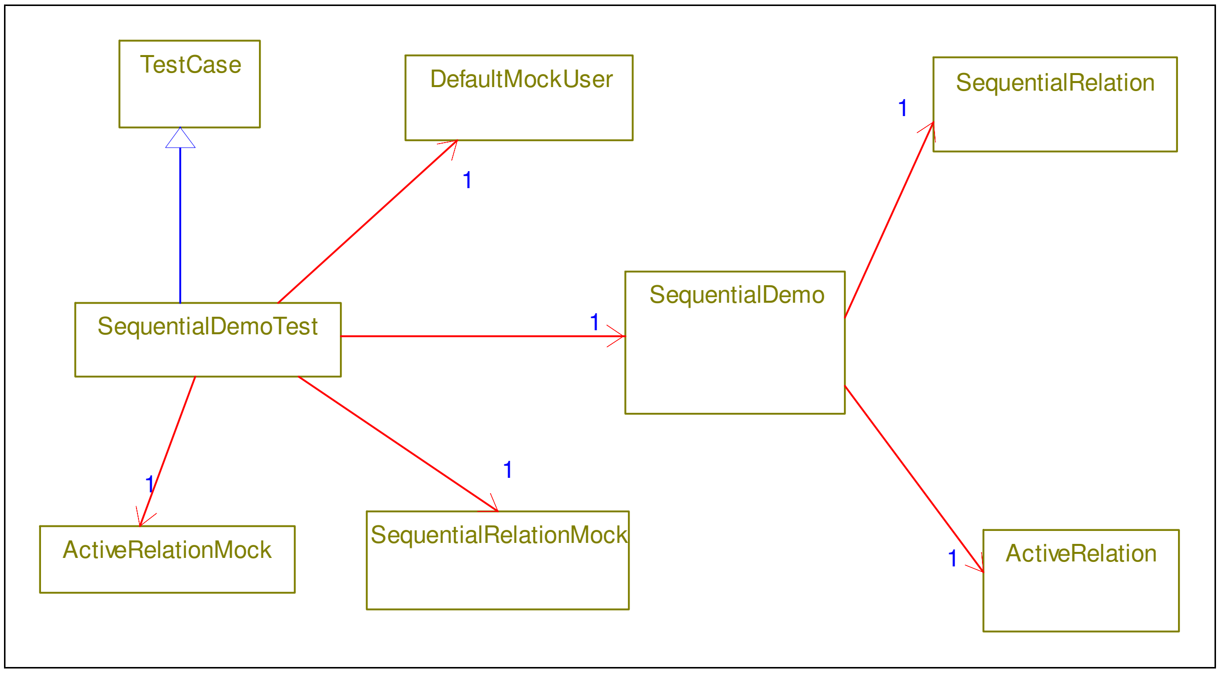


Figure 22: Relations of SequentialDemoTest

As it is seen from Figure 22, SequentialDemoTest has following relations:

- SequentialDemoTest is inherited from TestCase which is under the package RhapUnit.
- SequentialDemoTest has a directed association to:
    - the SequentialDemo,
    - the ActiveRelationMock which is the mock of the class ActiveRelation,
    - the SequentialRelationMock which is the mock of the clasas SequentialRelation,
    - the DefaultMockUser which is given to the mocks as MockUser and is under the package RhapUnit.

The operations of SequentialDemoTests are described in following sections in the sequence of createTests, setUp, tearDown, testSum, testTimeCritical, testSendPropertiesTrueCase, testSendPropertiesFalseCase. These operations shows how RhapUnit is used.

createTests

In the createTests, it seen that how tests are grouped. createTest is used to create a group of tests of SequentialDemo.

```
public static Test * createTests() {
		TestSuite * suite = new TestSuite;
		ADD_TEST(SequentialDemoTest, testSum, suite );
		ADD_TEST(SequentialDemoTest, testSendStartMessages, suite );
		ADD_TEST(SequentialDemoTest, testSendPropertiesTrueCase, suite );
		ADD_TEST(SequentialDemoTest, testSendPropertiesFalseCase, suite );
		ADD_TEST(SequentialDemoTest, testTimeCritical, suite );
		return suite;
	}
```

As it seen from the code of create test, a TestSuite is created. After that tests of SequentialDemoTest are created and added to the this suite, by using ADD_TEST macro. testSum, testSendStartMessages, testSendPropertiesTrueCase, testSendPropertiesFalseCase, testTimeCritical are test methods of the SequentialDemoTest.

Registering Tests of SequentialDemoTest

To register SequentialDemoTest, first you should create a dependency between AllTestRegistor and SequentialDemoTest. This is shown in Figure 23. And after that, you should add tests of SequentialDemoTest to the TestRunner in the constructor of the AllTestRegistor.

Constructor of AllTestRegistor:

```
	public AllTestRegistor(){
		TestSuite * suite = new TestSuite();
		suite -> add( SequentialDemoTest::createTests() );
		TestRunner::add(suite);
	}
```

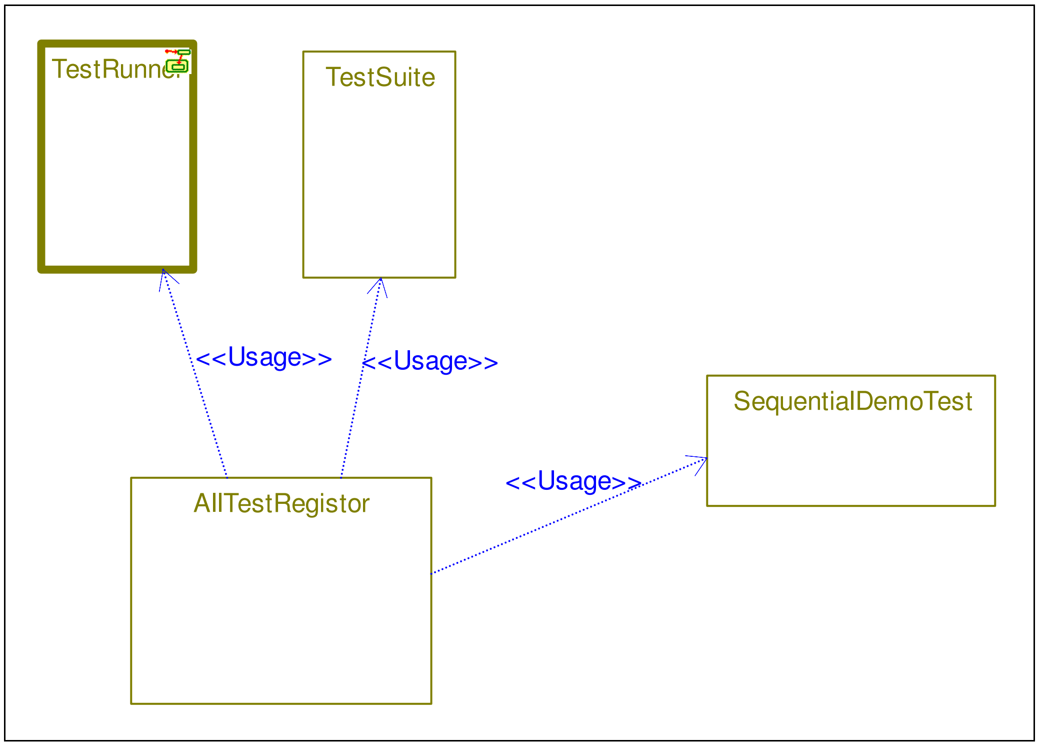


Figure 23: Dependency from AllTestRegistor to the SequentialDemoTest

setUp and tearDown

Operation setUp is called before the all tests. In setUp, you can create all common objects and setting their relations. Operation tearDown is called after the all tests. In tearDown you can delete all objects. As it is seen from the setUp, user of the mock objects should be setted.

```
protected void setUp(){
        itsSequentialDemo = new SequentialDemo();
        itsDefaultMockUser = new DefaultMockUser();

        itsSequentialRelationMock = new SequentialRelationMock();
        itsSequentialRelationMock -> setItsMockUser( itsDefaultMockUser );

        itsActiveRelationMock = new ActiveRelationMock();
        itsActiveRelationMock -> setItsMockUser( itsDefaultMockUser );

        itsSequentialDemo -> setItsSequentialRelation(itsSequentialRelationMock);
        itsSequentialDemo -> setItsActiveRelation(itsActiveRelationMock);
}
```

```
protected void tearDown(){
        delete itsSequentialDemo;
        delete itsActiveRelationMock;
        delete itsSequentialRelationMock;
        delete itsDefaultMockUser;
}
```

testSum

In the testSum method, sum method of the SequentialRelation is tested. How conditions are checked is seen in this test method. The sum method and its test are below:

```
public int sum(  int arg1, int arg2 ){
        return arg1 + arg2;
}

public void testSum(){
        int sum = itsSequentialDemo -> sum( 4, 6);
        CHECK_EQUAL("sum must be ten.", 10, sum);
}
```

testTimeCritical

In the testTimeCritical method, timeCritical method of the SequentialRelation is tested. How time criterions are tested is seen in this test method. The timeCritical method and its test are below:

```
public void timeCritical(){
        OXFTDelay( 450 );
}

public void testTimeCritical(){
        START_TIMER( 10 );
        itsSequentialDemo -> timeCritical();
        CHECK_TIME( "", 500 );
}
```

testSendStartMessages

In the testSendStartMessages method, sendStartMessages method of the SequentialRelation is tested. Mock objects are used is seen in this test method. The sendStartMessages method and its test are below:

```
public void sendStartMessages(){
        itsSequentialRelation -> start();
        itsActiveRelation -> GEN( evStart() );
}
```

```
public void testSendStartMessages (){
        // To set expectations from mock objects, set their mode as expectation.
        itsSequentialRelationMock -> startExpectation();
        itsActiveRelationMock -> startExpectation();

        // Set expectations by calling expected methods.
        itsSequentialRelationMock -> start();
        itsActiveRelationMock -> GEN( evStart );

        // To collect actual calls, set their mode as actual.
        itsActiveRelationMock -> startActual();
        itsSequentialRelationMock -> startActual();

        itsSequentialDemo -> sendStartMessages();

        // To verify that all expectations are accomplished.
        itsDefaultMockUser -> verify();
}
```

As it is seen from the code of testSendStartMessages, using a mock object has three part:

- setting expectations
- getting actual results
- verifying results

testSendPropertiesTrueCase

In the testSendPropertiesTrueCase method, one case of sendProperties method of the SequentialRelation is tested. How return values of mock mock objects are set is seen in this test method. The sendProperties method and its test are below:

```
public void sendProperties ( float sequentialProperty, OMString sequentialProperty2,
int activeProperty){

        OMBoolean sendSecondProperty = itsSequentialRelation -> firstProperty(
        sequentialProperty ) ;

        if (sendSecondProperty) {
                itsSequentialRelation -> property( sequentialProperty2 );
        }
        itsActiveRelation -> GEN(evProperty(activeProperty));
}

public void testSendPropertiesTrueCase(){

        itsSequentialRelationMock -> startExpectation();
        itsActiveRelationMock -> startExpectation();

        // setting of return value of method firstProperty.
        OMBoolean return1 = TRUE;
        itsSequentialRelationMock -> addReturnValue( &return1 );
        itsSequentialRelationMock -> firstProperty((float) 3.26);
```

```
		itsSequentialRelationMock -> property( "test ediliyorsun" );
		itsActiveRelationMock -> GEN( evProperty(2654) );

		itsSequentialRelationMock -> startActual();
		itsActiveRelationMock -> startActual();

		itsSequentialDemo -> sendProperties( (float) 3.26, "test ediliyorsun", 2654 );

		itsDefaultMockUser -> verify();
	}
```

As it is seen from the code of testSendPropertiesTrueCase, method addReturn value is used to set return values of a mock objects method.

testSendPropertiesFalseCase

In the testSendPropertiesFalseCase method, one case of sendProperties method of the SequentialRelation is tested.. The testSendPropertiesFalseCase method is below:

```
	public void testSendPropertiesFalseCase(){

		itsSequentialRelationMock -> startExpectation();
		itsActiveRelationMock -> startExpectation();

		OMBoolean return1 = FALSE;
		itsSequentialRelationMock -> addReturnValue( &return1 );
		itsSequentialRelationMock -> firstProperty((float) 3.26);
		itsActiveRelationMock -> GEN( evProperty(2654) );

		itsSequentialRelationMock -> startActual();
		itsActiveRelationMock -> startActual();

		itsSequentialDemo -> sendProperties( (float) 3.26, "test ediliyorsun", 2654 );

		itsDefaultMockUser -> verify();
	}
```

Writing Mock Objects

As it seen from the Figure 24, mock objects are inherited from class Mock under the package RhapUnit.

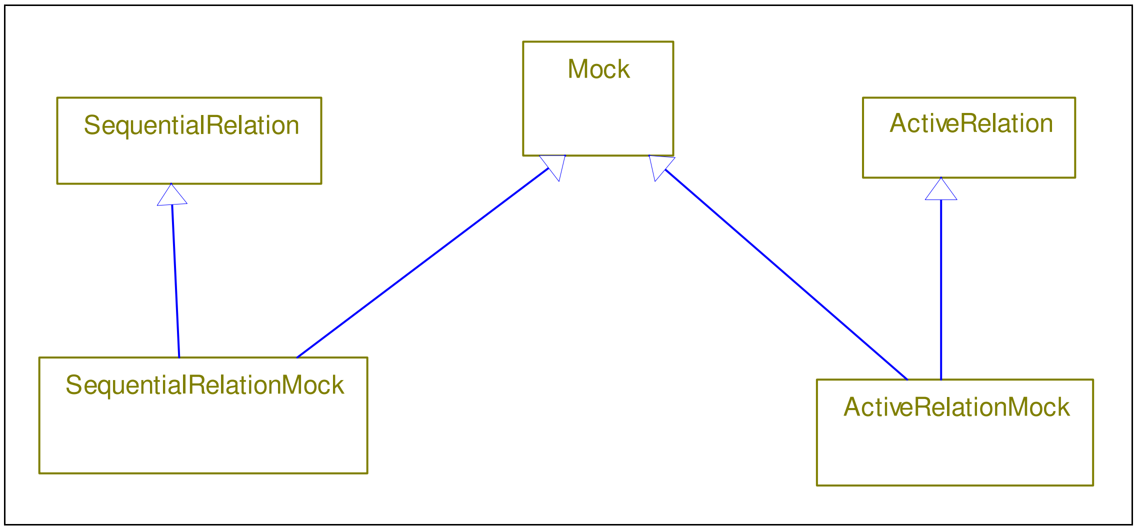


Figure 24: Mocks of classes SequentialRelation and ActiveRelation.

SequentialRelationMock

The property methods of SequentialRelationMock are shown below:

```
public void property( int arg ) {
        MOCK_START( SequentialRelation, property );
        MOCK_ADD_PARAM( arg );
        MOCK_END();
}

public void property( OMString arg ) {
        MOCK_START( SequentialRelation, property );
        MOCK_ADD_PARAM( arg );
        MOCK_END();
}
```

By the help of macros MOCK_START, MOCK_ADD_PARAM the information of method is converting to a string value. By the help of MOCK_END this value is sending to the mock user according the state of the mock. Macro MOCK_ADD_PARAM should be called for each parameter.

The firstProperty method of SequentialRelationMock are shown below:

```
public OMBoolean firstProperty( float arg ) {
        MOCK_START( SequentialRelation, firstProperty );
        MOCK_ADD_PARAM( arg );
        MOCK_END();

        MOCK_RETURN( OMBoolean );
}
```

By the help of macro MOCK_RETURN, return type of method is set.

ActiveRelationMock

The class under the test sends events to the active relations. This events could be cathed by overriding method gen of class OMReactive. The method gen of ActiveRelationMock is below:

```
public OMBoolean gen ( OMEvent * event, OMBoolean genFromISR = FALSE ) {
        if ( event -> isTypeOf( evStart_Relations_id ) )
        {
                evStart * ev = ((evStart *) event );
                MOCK_START( ActiveRelation, evStart );
                MOCK_END();
        }
        else if ( event -> isTypeOf( evProperty_Relations_id ) )
        {
                evProperty * ev = ((evProperty *) event );
                MOCK_START( ActiveRelation, evProperty );
                MOCK_ADD_PARAM( ev->arg );
                MOCK_END();
        }

        if (!genFromISR) delete event;
        return true;
}
```

Testing Active Object

The state chart of class ActiveDemo is shown in Figure 25. The default state is Idle. State transition from state Idle to state Execution occur by consuming event evStartExecution. Action of this transition is calling method start of the attribute itsSequentialRelation. In the execution mode event evSendProperty(OMString arg) is consumed. The action of evSendProperty is calling method property of attribute itsSequentialRelation with the input argument. In the state execution property method of itsSequentialRelation is called once with the calculated value (turnNumber * 50) for each timeout. State transition from state Execution to state Stop occur by consuming event evStopExecution.

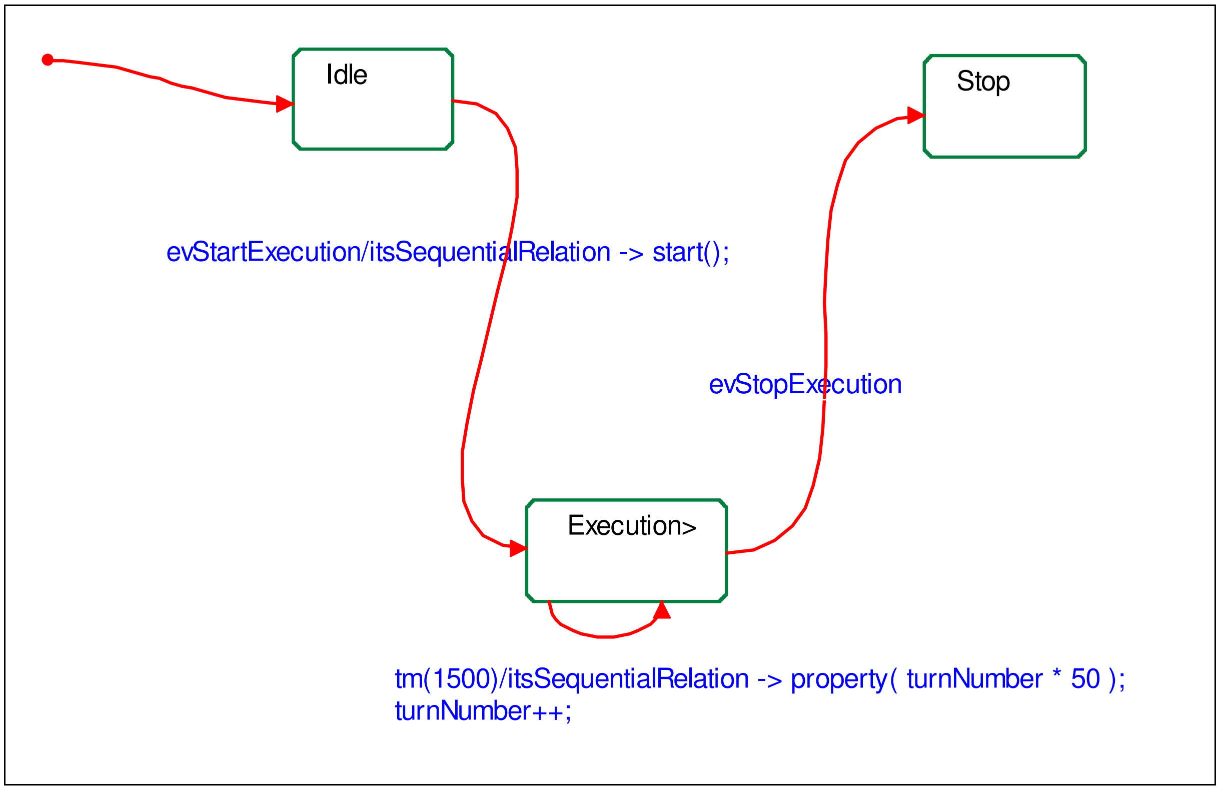

Idle
Stop
evStartExecution/itsSequentialRelation -> start();
evStopExecution
Execution>
tm(1500)/itsSequentialRelation -> property(turnNumber * 50);
turnNumber++;

Figure 25: State chart of the class ActiveDemo.

ActiveDemoTest Class

ActiveDemoTest is the test class of ActiveDemo. The relations of ActiveDemoTest is shown in Figure 26. The explanations of relations are same as SequentialDemoTest

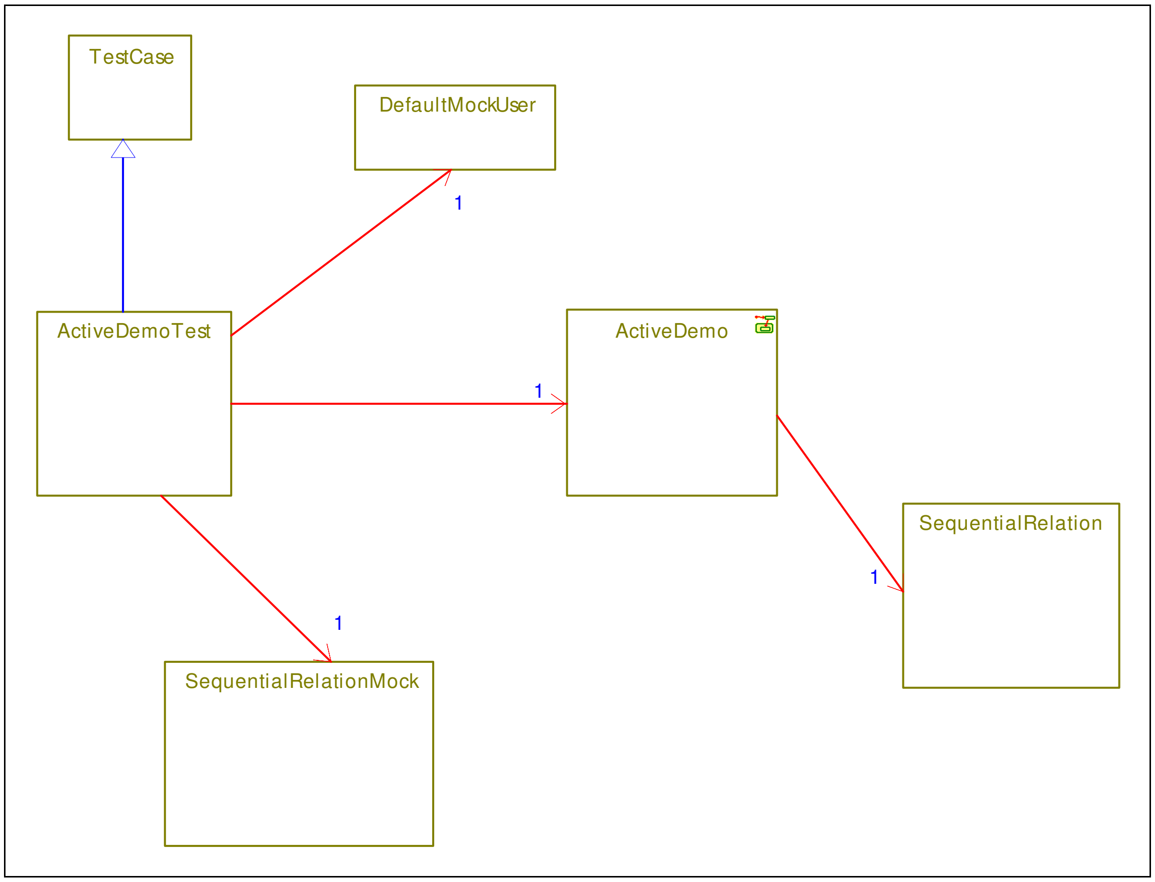


Figure 26: Relations of ActiveDemoTest

The operations of ActiveDemoTests are described in following sections in the sequence of createTests, setUp, tearDown, testIdle, testStartExecution, testExecutionReactionInState, testExecutionTimeOut, testStop. These operations shows how an active object is tested.

createTests

In the createTests, it seen that how tests are grouped. createTest is used to create a group of tests of ActiveDemo.

```
public static Test * createTests() {
        TestSuite * suite = new TestSuite;
        ADD_TEST(ActiveDemoTest, testIdle, suite );
        ADD_TEST(ActiveDemoTest, testStartExecution, suite );
        ADD_TEST(ActiveDemoTest, testExecutionReactionInState, suite );
        ADD_TEST(ActiveDemoTest, testExecutionTimeOut, suite );
        ADD_TEST(ActiveDemoTest, testStop, suite );
        return suite;
}
```

Registering Tests of ActiveDemoTest

To register ActiveDemoTest, first you should create a dependency between AllTestRegistor and ActiveDemoTest. This is shown in Figure 27. And after that, you should add tests of ActiveDemoTest to the TestRunner in the constructor of the AllTestRegistor. As it is seen, now tests of SequentialDemoTest and ActiveDemoTest are running together.

Constructor of AllTestRegistor:

```
public AllTestRegistor(){
        TestSuite * suite = new TestSuite();
        suite -> add( SequentialDemoTest::createTests() );
        suite -> add( ActiveDemoTest::createTests() );
        TestRunner::add(suite);
}
```

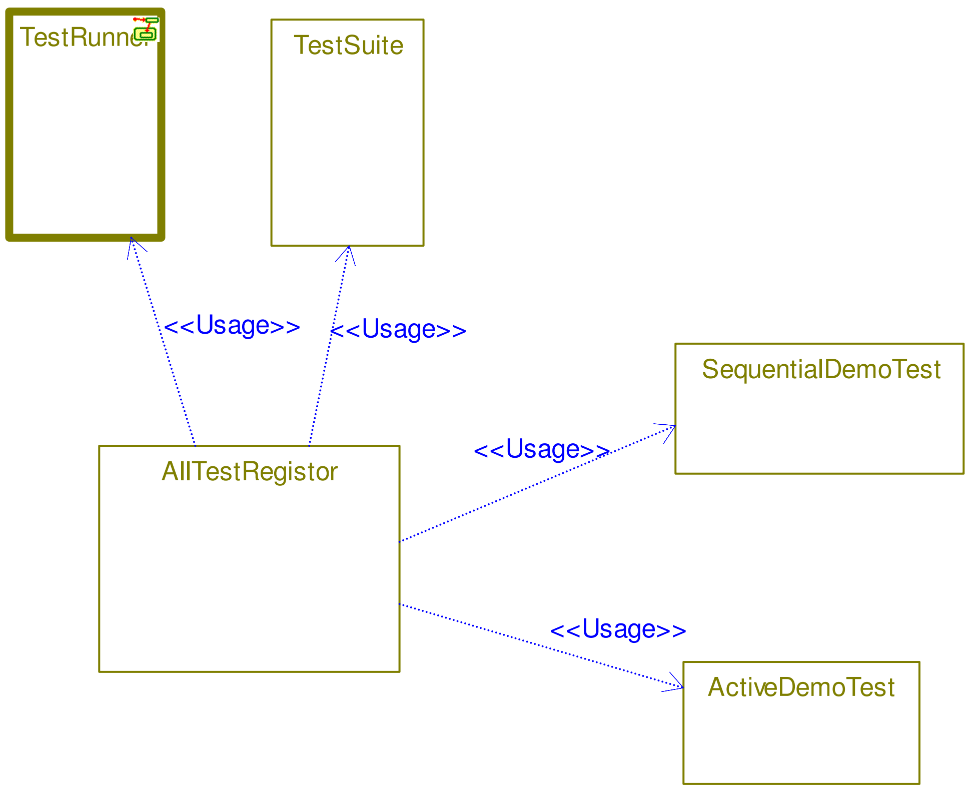


Figure 27: Dependency from AllTestRegistor to the ActiveDemoTest

setUp and tearDown

Operations setUp and tearDown are shown below:

```
protected void setUp(){
        itsActiveDemo = new ActiveDemo();
        itsDefaultMockUser = new DefaultMockUser();
        itsSequentialRelationMock = new SequentialRelationMock();
        itsSequentialRelationMock -> setItsMockUser( itsDefaultMockUser );

        itsActiveDemo -> setItsSequentialRelation( itsSequentialRelationMock );
}

protected void tearDown(){
        delete itsActiveDemo;
        delete itsDefaultMockUser;
        delete itsSequentialRelationMock;
}
```

testIdle

In the testIdle method, startBehavior of the ActiveDemo is tested. The testIdle method is below:

```
public void testIdle (){
        itsSequentialRelationMock->startExpectation();
        // since calling of sequential relation is not expected, it is directly started actual
        mode.
        itsSequentialRelationMock->startActual();

        //startBehaviour
        itsActiveDemo -> rootState_entDef();
        //it is not expected to consume evSendProperty
        evSendProperty ev( "xxyyxx" );
        itsActiveDemo -> takeEvent( &ev );
}
```

Since there is not any mock call expected, it is not required to call "verify" from default mock user.

testStartExecution

In the testStartExecution method, transition from Idle to Execution is tested. The testStartExecution method is below:

```
public void testStartExecution (){
        //startBehaviour
        itsActiveDemo -> rootState_entDef();
        itsSequentialRelationMock->startExpectation();
        itsSequentialRelationMock->start();
        itsSequentialRelationMock->startActual();

        evStartExecution ev;
        itsActiveDemo -> takeEvent( &ev );

        itsDefaultMockUser->verify();
}
```

testExecutionReactionInState

In the testExecutionReactionInState method, reaction in state of Execution is tested. The testExecutionReactionInState method is below:

```
public void testExecutionReactionInState (){
        //goto Execution
        itsActiveDemo -> rootState_entDef();
        evStartExecution ev1;
        itsActiveDemo -> takeEvent( &ev1 );

        itsSequentialRelationMock->startExpectation();
        itsSequentialRelationMock->property("xxyyxx");
        itsSequentialRelationMock->startActual();

        evSendProperty ev2("xxyyxx");
        itsActiveDemo -> takeEvent( &ev2 );

        itsDefaultMockUser->verify();
}
```

testExecutionTimeOut

In the testExecutionTimeOut method, timeouts of Execution are tested. The testExecutionTimeOut method is below:

```
public void testExecutionTimeOut (){
        //goto Execution
        itsActiveDemo -> rootState_entDef();
        evStartExecution ev1;
        itsActiveDemo -> takeEvent( &ev1 );

        itsSequentialRelationMock->startExpectation();
        itsSequentialRelationMock->property(0);
        itsSequentialRelationMock->property(50);
        itsSequentialRelationMock->property(100);
        itsSequentialRelationMock->startActual();

        OMTimeout    timeOut1(ActiveDemo_Timeout_Execution_id,    itsActiveDemo,
        1500, "ROOT.Execution");
        itsActiveDemo -> takeEvent( &timeOut1 );
        OMTimeout    timeOut2(ActiveDemo_Timeout_Execution_id,    itsActiveDemo,
        1500, "ROOT.Execution");
```

```
                itsActiveDemo -> takeEvent( &timeOut2 );
                OMTimeout    timeOut3(ActiveDemo_Timeout_Execution_id,    itsActiveDemo,
                1500, "ROOT.Execution");
                itsActiveDemo -> takeEvent( &timeOut3 );

                itsDefaultMockUser->verify();
        }
```

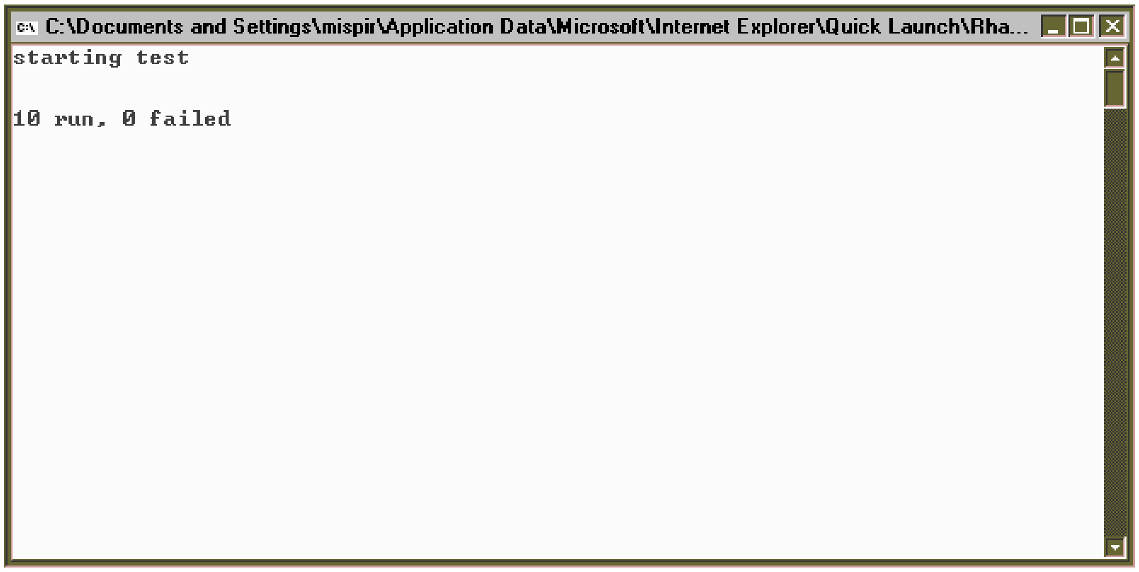
testStop

In the testStop method, timeouts of Execution are tested. The testStop method is below:

```
        public void testStop (){
                //goto Execution
                itsActiveDemo -> rootState_entDef();
                evStartExecution ev1;
                itsActiveDemo -> takeEvent( &ev1 );

                itsSequentialRelationMock->startExpectation();
                // since calling of sequential relation is not expected, it is directly started actual
                mode.
                itsSequentialRelationMock->startActual();

                evStopExecution ev2;
                itsActiveDemo -> takeEvent( &ev2 );

                //it is not expected to consume evSendProperty
                evSendProperty ev3( "xxyyxx" );
                itsActiveDemo -> takeEvent( &ev3 );
        }
```

Running Of Test

If you generate/make/run the test configuration, test result is shown in command line. The succesfull tests output for this tutorial is shown in Figure 28.



```
C:\Documents and Settings\mispir\Application Data\Microsoft\Internet Explorer\Quick Launch\Rha...
starting test

10 run, 0 failed
```

Figure 28: The output of tests.