VV	Course:	Software Verification and Validation, DAD404		
	Teacher:	Jilles van Gurp	Department:	IDE, University Of Karlskrona/Ronneby

Object Oriented Testing Report

Prepared for

Jilles van Gurp
Jilles.van.Gurp@ide.hk-r.se
IDE, University Of Karlskrona/Ronneby

Prepared by

Christian Bucanac
c.bucanac@computer.org
Software Engineering Student,
University Of Karlskrona/Ronneby

1998-12-09

Contents

Abstract

Object-oriented software is developed in iterative and recursive increments. After each increment the software is put together. This is where object-oriented integration testing plays it's role. It tests the interactions and interfaces of the components that are put together.

The main problem with object-oriented integration testing is that the software is too complex. This leads to many possible test cases and high cost of testing.

There are several strategies for performing object-oriented integration testing. In this report the scenario testing strategy is discussed in more detail.

Introduction

This is the last report in the Software Verification and Validation course, DAD 404 given at the University of Karlskrona/Ronneby. This report is about object-oriented testing.

The report contains a discussion of a topic related to object-oriented testing. I have chosen to discuss the topic Object-Oriented Integration Testing.

This report together with the other student's reports is used in the object-oriented testing seminar held during the course.

Object-Oriented Integration Testing

What is it?

Object-oriented testing can be done on four different levels:
- Testing a single method.
- Testing an object.
- Testing sets of object-oriented components (objects or sets of objects).
- Testing an entire system.

Object-oriented integration testing is about testing sets of object-oriented components.

Object-oriented integration testing is the testing of the interaction between object-oriented components. Integration testing assumes that unit testing has been done on the components and that the defects have been removed. These fault free components are integrated. The goal of integration testing is to find defects that arise when these fault free components interact with each other in an incorrect way.

Object-oriented integration testing is when you test the interaction between two objects. An interaction between objects takes place when an object calls another objects method, by sending a message. These two objects must be instantiated from two different classes. If the objects are instantiated from the same class, then it is considered to be unit testing.

Object-oriented software is developed incrementally with iterative and recursive cycles of planning, analysis, design, implementation and testing. Integration testing plays a special role here, since it is done after each increment.
The is a small difference between integration and regression testing, since the testing is done after each increment, it can also be considered to be regression testing. I see it as integration testing. Regression testing is done when a change is made in software, usually after it has been developed. When I build software I do not change it after each increment. I build the new increment on previous increment.

There are some implications for object-oriented integration testing:
- Individual methods within an object are imperative. All object-oriented languages return control to the calling object when a message is finished. There is no single execution trace in object-oriented software. An execution trace can be divided into several smaller or larger execution traces.
- There is no clearly defined integration structure. There is no tree that decides the order of integration testing the objects. The reason for this is that there might not be a topmost or bottommost component in object-oriented software.
- There is a difficulty in testing object-oriented software. An object may be in many different states and have many relations to other objects.

Differences with Traditional Testing

Software implemented in an object-oriented language has a different structure and behavior than software implemented in a traditional procedural language. Object-oriented software is not functionally decomposed into separate procedures. It is composed of objects and classes, which interact via message passing and share definitions via inheritance. Object-oriented software is not developed with the traditional waterfall development cycle. It is developed incrementally with iterative and recursive cycles of analysis, design, implementation and testing. These fundamental differences have a large impact on how object-oriented software is tested.

Problems with it

Testing object-oriented software is much more complex than testing traditional software. The productivity gained from object-oriented technology is almost lost during testing. Objects communicate and influence each other through messages. A message changes an object's state. How an object responds to a message depends on it's state. Objects are created and deleted dynamically during runtime. Their interrelationship is also built up during runtime. The possible interactions between objects/components increases quadratic in object-oriented software compare to traditional software:

Object-oriented complexity compare to traditional software.

The complexity in object-oriented software makes it impossible to test all interactions. It would cost too much. To minimize cost you should try to test the software in a smart way. Take the inheritance relationship for example. Assume that you have tested the interaction between object A_1 and B_1. Assume that you have a third object A_2 that inherits from A_1. When you test the A_2 object, you should not have to test the A_1 object again. You should only have to test the new and redefined methods. After the testing of the A_2 object, the B_1 object should be able to interact with it.

Integration testing the software after each increment may add up to a very large cost. There are various strategies that can minimize this cost. One strategy is to minimize the need for drivers and stubs. Another strategy is to try to isolate the changes from the unchanged parts of the system. If this can not be done, then you will have to do a more costly regression test of the software rather than a less costly integration testing.

There is a problem with cyclic dependencies in object-oriented software, when there are mutual dependencies. The question is how should you do the integration testing when there is no top or bottom in the object structure? Where should you start to test? Where should it end? Does the testing end?

The test cases may be too many, since there can be many complex interactions between objects. The question is what should you test and how much should you test?
One tactic is to select a minimal set of revealing test cases. You should skip those test cases that are not affected by the changes since the last iteration of integration testing. Microsoft and Mozilla [Moz98] use this tactic, where the developers synchronize the project each day. The testing is done during the night.
The question with this tactic is coverage. How do you know if you have selected the right minimal set of revealing test cases? How do you know that you have not induced an error in the part of software that is not tested?

There are several coverage criteria that have to be fulfilled to cover the whole object-oriented software:

- Method coverage – All method nodes of a class message diagram* have to be visited at least once, that is, each method in the system has to executed at least once to achieve 100 percent coverage.
- Message coverage – Messages can be divided into partitions according to the source of the message. At least one message from each source should be executed.
- Polymorphic message coverage – One message should be executed at least once for each possible polymorphic binding.

*A class message diagram is composed of nodes and arcs. The nodes represent the methods and the arcs represent a message between two methods.

It may not be possible to cover and execute all test cases. Therefore the most critical and important test cases should be executed. An analysis of the test cases must be done to discover these test cases. The selection of test cases may be based on:

- How important is the test case for the customer? You should try to prioritize the test cases so that they test the functionality that is most important for the customer.
- How critical is the path that the test case tests? You should prioritize the test cases after which test case tests a more critical path.
- How deep is the trace of the test case in the software? The longer the trace is, the more it tests. The test cases should be prioritized after this.

Testing Strategies

There are several testing strategies that can be used during integration testing. Different strategies find different kinds of errors. Some general strategies are:

- Subassembly and scenario testing – The subassembly testing tests the collaboration of objects by sending messages. Scenario testing tests the interaction of several objects that fulfill a specified use case. Example of errors associated with subassembly and scenario testing are:
  - Messages sent to objects that no longer exist.
  - Incorrect message passed to an object.
- Inheritance and classification testing – The inheritance testing tests the polymorphism of a class hierarchy. This is a very difficult part of testing. Because of the polymorphism, you do not now until runtime how the classes are called. Classification testing tests the relationship between instances of classes. Example of errors associated with inheritance and classification testing are:
  - Generic classes are improperly instantiated.
  - Wrong class is inherited.
- Aggregation testing – Tests to find errors and inconsistencies in the integration of the components into the aggregate. Example of errors associated with aggregates testing are:
  - Components are missing.
  - There are inconsistent components.
  - Components not destroyed when the aggregate is destroyed.

Some deeper testing strategies are:

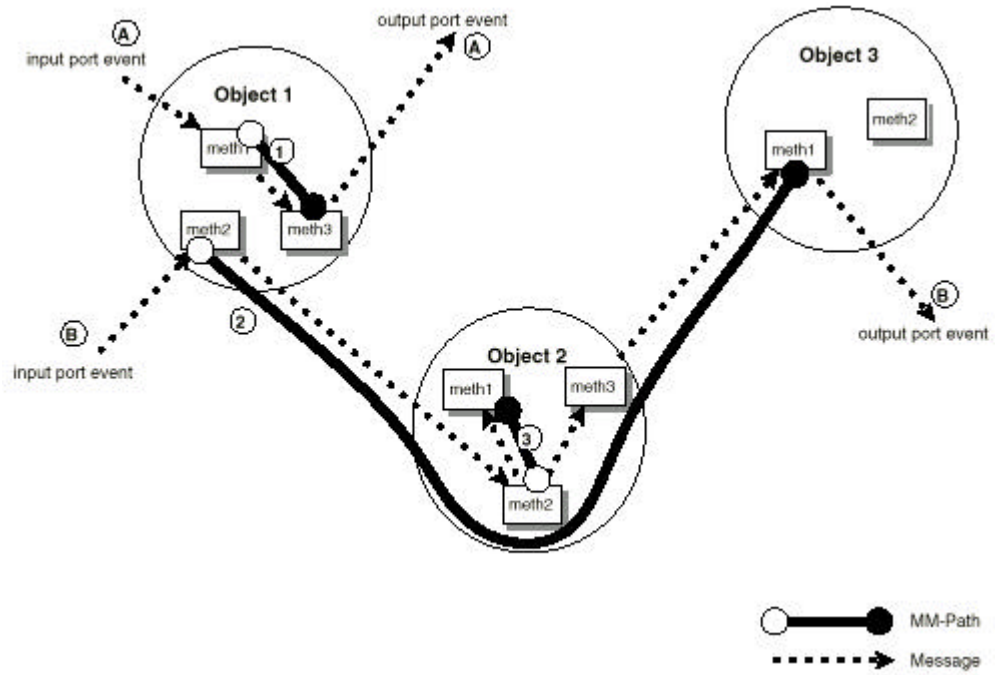
- Execution based integration test – Traces the execution of an interaction. This testing strategy finds control flows that can not be executed.
- Value based integration test – Executes the interaction between components with certain values. This testing corresponds to the traditional boundary value, input validation and syntax testing. This testing strategy finds errors like passing of illegal parameters and interpretation problems of parameters.
- Function based integration testing – Tests the correct provision of functionality through the component's collaboration. This testing strategy focuses on detecting mismatches between the interpretation of the interaction between components.

Scenario testing

In this section I will discuss the scenario testing strategy in more detail. I will start the discussion from a detailed level and go up towards a more general level. I will first discuss the Method-Message Paths (MM-Path) and Atomic System Functions (ASF) that can be found in [EriJor94]. Thereafter I will discuss use cases described in [Bin96].

Method-Message Path

A MM-Path is a sequence of method executions linked by messages. Take a look at this figure:



A MM-Path start with a method and ends when it reaches a method that does not call another method. When there are no more subsequent method calls, the control returns to the start method. The start method has then executed it's task and the system goes into a quiescence state.

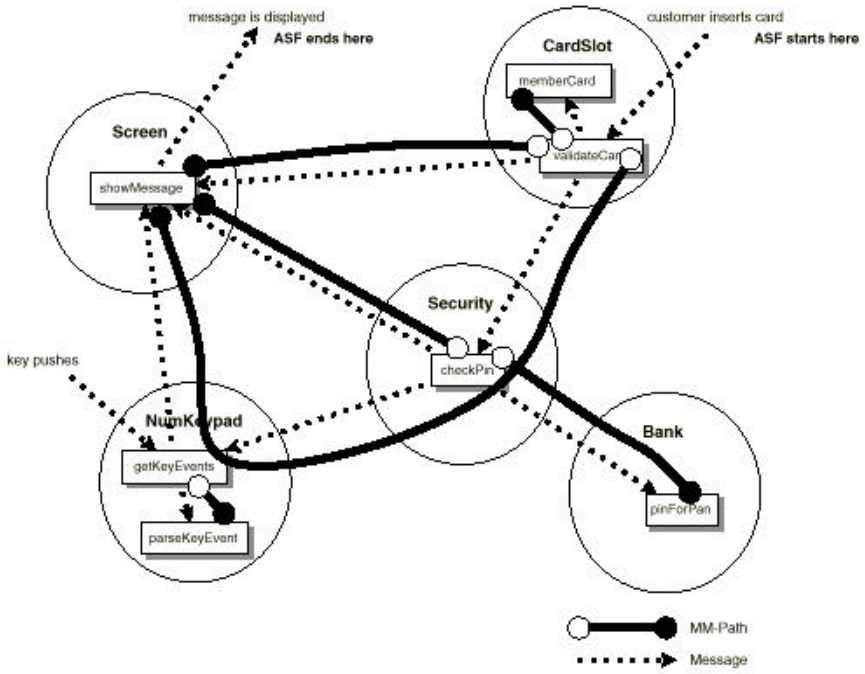
The execution in object-oriented software begins with an event. This event can be seen as an input port to the software, for example input port event B in the figure above. The input port event triggers the method message sequence of a MM-Path. This MM-Path may in it's turn trigger other MM-Paths for completing it's task. The MM-Path ends up with an output port event. This output port event might trigger a window to open or change the state of the system.

A MM-Path is a small building block in a scenario. Even if it is a small building block, a MM-Path tests the integration and interaction of several objects. Several methods pass messages to each other, their message interaction must work if a MM-Path is to work. When a MM-Path, that is on a low level, tests so much of the integration between objects, imagine how much of the integration an ASF or a use case, which are on a higher level, tests.

Atomic System Function

An ASF is an input port event followed by a set of MM-Paths and ends with an output port event. This sounds like a MM-Path consisting of other MM-Paths. The difference between an ASF and a MM-Path is that an ASF is an elemental function visible at the system level. A MM-Path on the contrary is not visible at the system level. An ASF is the point at where the integration and system testing meet each other.

Here is an example of an ASF:



The picture shows the ASF of entering a Personal Identification Number (PIN) in an automated teller machine (ATM). This figure shows the interaction of objects and the message passing when a correct PIN is entered. If you are going to handle errors, then there might be a more complex structure of objects with several output port events.

The PIN entry ASF consists of 6 MM-Paths. It starts with the customer inserting his/her card. The ATM ends the ASF by showing the customer an okay message of having accepted the PIN entry. The ASF consists of these steps:

1. The card is validated and checked if it is a member card.
2. An OK message is displayed that asks the customer for his/her PIN.
3. The customer enters his/her PIN and the ATM parses it.
4. By sending the Personal Account Number (PAN) from the card to the bank, the right PIN is obtained by the ATM.
5. The entered PIN is checked against the PIN obtained from the bank.
6. The ATM displays a message asking the customer of the amount of money s/he wants to withdraw.

A simple PIN entry ASF like this is a small part of a larger ATM session. It tests the interaction of a small part of the system with few objects involved. It is therefore considered to be integration testing. When you test a whole ATM session then you are doing system testing.

[EriJor94] made an ATM simulator and did integration testing as described above. They found several errors during their testing. One for example was the lack of invocation of the superclass init method in the NumKeypad init method. This resulted in that the entered PIN was shown on the screen. Doing unit testing on the objects did not reveal this error. It was revealed by integration testing when messages were sent between the NumKeypad, Screen and Security objects.

Use cases

Use case testing is normally done while testing the whole system. As with ASF, testing a use case that is part of a larger use case is considered to be integration testing. For example, a use case that tests the communication through all protocol layers in an OSI stack is considered to be system testing. A use case that tests the communication between the TCP and IP protocols is considered to be integration testing.

A use case is a dialog between the system and an external actor. The actor may be another system or a human. A scenario is a specific instance of a use case. A use case defines the steps in a use case, but the scenario defines what happens at each step, the inputs and outputs.
A use case is an abstraction of a sequence of operations (messages passed between objects). A use case consists of many operations that the user does not see. This sequence of operations accomplishes a task from a user's point of view. The user only sees the input and output.

The MM-Path and ASF testing is white box testing of object-oriented software. Use case testing is on the other hand black box testing of object-oriented software. This is because you are doing the testing from the user's point of view. You only care of the input and output of the system.

A use case for the PIN entry ASF can be defined. For example:
1. Input: The user inserts his/her card. Output: Show a screen that asks the users for his/her PIN.
2. Input: The user enters his/her PIN. Output: Show a screen that asks the user of the amount of money s/he want to withdraw.

As you can see, the more abstract the test cases are the fewer steps you have in your test case. This means that the test cases can be executed fast. There is a disadvantage of this. When you find an error, you might not know where it occurred since you are testing the software on a too abstract level. The time you gain by testing on a high level is lost in the time spent to find the specific bug that caused the error. It is important to find the right level of abstraction for the test cases. Generally, the more confidence you have in the software the more abstract level of testing you can perform.

As mentioned earlier, object-oriented software is very complex. You can't possibly perform all possible test cases. A way to handle this problem is to perform statistical usage testing.
From the software specification you create use cases. Since different people will use the software in different ways you have to develop operational profiles for each use case. An operational profile consists of the operations that compromise all the use cases in the system and the relative frequency of each operation. A Markov Model [WhiPoo93] can model an operational profile. A Markov model consists of states and arcs. The states represent the operations and the arcs from a state represent the frequency of the operation.
You should test the most likely use cases first, because the most frequently used operations have the greatest impact on reliability. If a heavily used operation is buggy then the software will fail frequently. If a rarely used operation is buggy then the system will fail infrequently. Selecting to test use cases that are most likely and most frequently used decreases the failure rate rapidly.
By doing statistical testing on object-oriented software you find out statistically how reliable it is. You find out how much you can trust the software and how confident you can be about the software. With statistical testing you can also calculate the failure rate of the software.

Conclusion

Object-oriented integration testing plays an important role in object-oriented software development. The software is developed incrementally in iterative and recursive cycles. After each cycle the parts of the software are put together and integration testing is performed.

There is no clear structure in object-oriented software. There might not be a topmost or bottommost component. Components may send messages to any other components. Each message changes a component's state. This leads to nearly infinite number of test cases. There are too many possible states and interactions. This complexity leads to high cost in testing.

Scenario testing is a good strategy for testing object-oriented software. The level of abstraction can be chosen to fit the particular software. If you have high confidence in a particular piece of software then use case testing can be used. The use case test cases can be executed very fast. There is a disadvantage if you find an error. It may be hard to locate since you are testing on a too high level.

References

[Bin96]
The FREE approach for System testing
Robert V. Binder
Object Magazine, February 1996

[EriJor94]
Object-Oriented Integration Testing
Carl Erickson, Paul C. Jorgenson
http://www.csis.gvsu.edu/~erickson/OO_Testing/ACMpaper.ps.Z

[Fir93]
Testing Object-Oriented Software
Donald G. Firesmith
Advanced Software Technology Specialists

[Moz98]
Mozilla
The open source project Mozilla that develops Netscape Communicator.
http://www.mozilla.org

[Ove94]
Integration Testing for Object-Oriented Software
Jan Overbeck, Ph.D. Thesis Dissertation
Vienna University of Technology, 1994
http://www.dbai.tuwien.ac.at/ftp/papers/overbeck-diss.ps.Z

[WhiPoo93]
Markov Analysis of Software Specifications
James A. Whittaker, J. H. Poore
ACM Transactions on Software Engineering and Methodology, Vol. 2, No.1, January 1993

[Win98]
Managing Object-Oriented Integration and Regression Testing
Mario Winter
euroSTAR 98, Munich, Germany