

Issues in Testing Object Orientated Systems

James Gawn
329338

January 12, 2007

Abstract

In this document you will find an overview of the object orientated paradigm and how implementing software with the paradigm affects the testing process. This will include discussing the issues surrounding inheritance, polymorphism as well the different levels of testing.

Contents

1	Introduction	3
1.1	Motivation	3
1.2	Testing	3
1.2.1	Functional Testing	4
1.2.2	Structural Testing	5
2	The Object Orientated Paradigm	6
2.1	Introduction	6
2.2	The Basics	6
2.3	Encapsulation	7
2.4	Inheritance	9
2.4.1	Basics of Inheritance	9
2.4.2	Composite Inheritance	11
2.4.3	Multiple Inheritance	12
2.5	Polymorphism	13
3	Testing Levels for Object Orientated Systems	15
3.1	Introducing Testing Levels	15
3.2	Method Level	16
3.3	Class Level	16
3.4	Integration Level	17
3.5	System Level	18
4	Testing Object Orientated Systems	19
4.1	Overview	19
4.2	Encapsulation	19
4.3	Inheritance	20
4.3.1	Class Flattening	22
4.4	Polymorphism	24
5	Summary	25
5.1	Conclusions	25
5.2	Futher Work	25

1 Introduction

The area of software testing is important one in software development, just as it is important for an engineer to ensure that a bridge will not fall down, or a building will not collapse. It is not only important to make sure that you don't have potentially fatal errors such as a building falling down, but also ensuring that any system that is developed actually performs correctly the task for which it is designed. In this document an overview of the effects in using the object orientated paradigm during the software development process with the focus being on the testing element.

In order to get a full understanding of the issues that will be discussed an understanding of the paradigm itself will be needed as such the first section of this document includes an object orientated introduction. This then continues into a section that outlines the different levels of testing during the development process, when using this model. Further to this the next section then goes into further detail into the various aspects of the object orientated system and how each of these in turn present their own individual challenges and opportunities when performing testing.

Finally the document is finished off with a summary of the important aspects, as well as a look into the study that can be further made into the area.

1.1 Motivation

The motivation of this document is to provide the reader with a basic understanding of the testing issues that arise when dealing with the object orientated paradigm. This is ever more important when you bare in mind the growing in popularity in using languages such as Java, C++, as well as a growing number of others which all make use of object orientated techniques. This combined with the fact that there are some significant issues that need to be address when working within it, make it important to at least consider and understand these in order to perform suitable levels of testing.

1.2 Testing

This chapter will outline some of the basic testing terms that are used throughout this paper. This will then provide some background knowledge of testing to facilitate understanding of various points that are made later.

First you have to define what it is testing is trying to achieve, and simply put it is ensuring that the an implementation matches a given specification. This therefore implicitly requires that a detailed specification of a system is created in order to allow effective testing. The next step being the creation of a variety of test cases, which are sets of inputs which then are matched with a set of expected outputs (based on the mentioned specification). Therefore one of the main goals in testing is to find suitable ways to generate or create ways of finding test cases that as far as possible are able to identify errors (where an error is a mistake made during the implementation of a specification) and to prevent faults. Faults are key in testing, these being the actual realisation of an error during the running of a system.

There are two distinct ways which have emerged to perform testing, with these being the functional and structural methods. It should be noted that neither of these is better than the other and both can be used to find test cases, although they often both are better at finding different types of errors.

1.2.1 Functional Testing

The first of these testing methods being functional, which relates to the generation of test cases solely based on the inputs and outputs of a given program without any regard for the manner in which the interior of the program operates. Of course this is very dependant on an accurate specification, if any part of a specification is incorrect it would be impossible to find any errors. It can be noted that this style of testing can often also be compared with the idea of blackbox testing [4, P7].

This method has a number of downfalls if used without the second method, the main one being that it can be impossible to find undocumented interfaces and features. This is simply because you pay no attention to the actual code, but rather, just the inputs and outputs of the known features, and therefore would be impossible for these to be found.

This downfall of functional testing can provide to be very troublesome, if start by considering a program written in java that stores passwords for you. Now if you then receive this program, and perform your functional testing on it, and it seems to be secure, so you start using it. If you then consider that the person who created the program was malicious in nature so they put an extra function into the program which was not specified by you that returns all the passwords without authentication, this would not be able to be found simply because it is not in the specification.

So a more tangible example of this might be if you are testing a light switch, in functional testing you might test to ensure that the light switch turns on if you press it, and then turns off when you press it again. Though you would not then open the light switch up to make sure that when you perform the various operations the circuitry inside is performing as you would expect.

1.2.2 Structural Testing

The second of the methods of testing is structure testing which focuses rather less on the actual specification but rather more on the implementation and programming code. This type of testing is also sometimes referred to as white box testing[4, P8].

The main focus of structural testing is often a number of test coverage metrics[1, P324], and by this it is meant the quantity of the code that has been executed during a test suite (where a test suite is a collection of test cases). This is achieved by performing detailed analysis on the source code to find a suitable level of coverage for the particular programs purposes.

There are like with functional testing a number of limitations if used as the sole way of testing a system, the most prominent of these being the most obvious which is testing for the absence of features. Due to the detachment from comparing the specification and the implementation you would be unable to find features that are missing, and in fact as a whole structural testing will only show that an implementation does what it intends but not what it is actually meant to be achieving.

Again if we consider the simplistic light switch example that was provided previously, in structural testing you might open up the light switch and measuring the voltage on the different wires within the switch while turning it on and off, to ensure that all the different wires have the expected voltages.

2 The Object Orientated Paradigm

2.1 Introduction

The first step in understanding the issues surrounding testing of systems programmed using the object orientated paradigm is to get an understanding of the paradigm itself. This can then give us some of the building blocks required to properly understand the issues that affect testing these types of systems.

Within this section some of the features of object orientated paradigm are discussed, as well as how that these differ from traditional procedural programming. This will include an overview of the concepts of encapsulation, the different approaches to inheritance, as well as polymorphism.

As well as discussing the differences I also plan to draw some parallels between existing languages, and the methodologies used in object orientated languages.

2.2 The Basics

The basis for the object orientated paradigm is by using the more modern approaches to data abstraction you eventually come across the concept of abstract types or classes. In object orientated programming we use these classes and introduce the concepts of hierarchies.

These classes then define the characteristics of the objects within a system, where these objects are simply a set of variables with a number of methods (or sometimes called functions) that are associated with them; with these methods allowing the manipulation of the data that is stored in these variables. It is often the case with these objects that the variables themselves are not accessible directly from outside the object without accessing them through the methods. If you now take this a step further and you populate the variables with values then you can consider that an object is an instantiation of a class. This process is often performed by a constructor which is often thought of as a special method that initialises or creates the object based on the class. These classes of objects can then be composed together to create series of co-operating objects that work together to make a whole system.

These objects can then provide a way to model a real world system or environment as collections of virtual entities in a simple way. If you consider

an object for example a radio, it may have a large number of states based on a number of different settings. If you take these setting on the radio and make these variables in an object, for instance if the radio is on or off could be a variable within a radio object, or the frequency currently tuned to. But then if you consider the methods used to change the frequency currently tuned, if the radio has some constraints, maybe you can't got above or below certain frequency's this can be enforced onto this object model through the methods. This can provide a reliable way to ensure that the model (when testing to it works as intended of course) can never enter an invalid state simple because the variable is not directly accessed but is changed by methods which the object has.

There are a number of key components that will be discussed in the next few sections which are used to construct object orientated languages. Further than just the idea of having classes of objects, the concepts of encapsulation (touched on already), inheritance, and polymorphism are important.

2.3 Encapsulation

The concept of encapsulation is not exclusively for the object orientated paradigm, and in fact, it is not even essential to it; although it is so often used in object orientated languages so it is important to make sure its implications in testing are understood, hence this section.

The concept of encapsulation have been in use for some time with non object orientated languages, for example in ADA packages [2, P137], although this is not in the scope of this document it is important to note that the testing issues related with encapsulation can be relevent for these languages as well as object orientated ones.

The general idea of encapsulation in object orientated system is to only make certain methods and/or variables from within an object visible to objects externally. This gives rise to the idea of public and private methods and variables. This is where public methods are invocable by other objects as well as the object in question, and public variables are accessible and modifiable by other objects as well as the source object. Private methods and variables on the other hand are strictly accessable only by the object itself, and are not accessable externally unless through some other methods. For example it is often deamed good practice with object orinetated programming to make variables private and mediate access to them always with accessor methods, or in other words have public methods to set and return the contents of

the private variable. In essence encapsulation is an access control system for objects, enabling objects to provide some kind of limited interface to its functionality without bearing all of its inner workings to other objects.

A simplistic abstract example of this could be a ticket system, consider that it has a certain number of tickets that it can give out before it runs out. Now someone accessing the system simply needs to know if it is able to dispense a ticket or not, but does not need to know how many tickets there are, or even how many are left, just that they are able to get one ticket (See Java code example 1).

Listing 1: A Simple Ticket Dispenser

```
public class TicketDispenser {  
  
    private int    numberOfTickets;  
    private int    numberOfTicketsDispensed;  
  
    //Creates a ticket dispenser with a no of tickets.  
    public TicketDispenser(int numberOfTickets) {  
        this.numberOfTickets = numberOfTickets;  
        numberOfTicketsDispensed = 0;  
    }  
  
    //Returns if tickets available.  
    public boolean returnTicketsAvailable() {  
        if (numberOfTicketsDispensed <  
            numberOfTickets) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}  
  
//Prints a ticket if any available.  
public boolean printTicket() {  
    if (returnTicketsAvailable()) {  
        numberOfTicketsDispensed++;  
    }  
}
```

```
        return true;
    } else {
        return false;
    }
}
```

As you can see in the preceding example, both the fields that hold the information about the number of tickets and tickets available are set to private. This is encapsulating this information so that is only accessible by this class, but this idea of encapsulation is not just limited to the variables. It is perfectly possible that you have a method that is private, in fact you could in this example make the "returnTicketsAvailable" method private, so that the ticket machine simply prints a ticket if there is one available but someone is unable to query in advance if a ticket is available.

In fact the concept of encapsulating methods can be a useful one, if you have a method that performs some complex task it is often useful to be able to break this down into a number of distinct methods. Although you don't want the external entity to have to call each of these methods in turn to perform the task, so if you make all these building block methods private and have just one public method taking the inputs require for all the methods then you get a simpler interface.

2.4 Inheritance

2.4.1 Basics of Inheritance

In contrast to encapsulation inheritance is one of the foundations of the object orientated paradigm, the basics for this idea is one class is able to inherit/extend the functionality of another class. This gives rise to the ideas of a superclass as well as a subclass, where a superclass refers to the class which a another class is inheriting from, and where this inheriting class a subclass.

So if you continue the example of the ticket machine from the previous example, suppose that you now want to construct a more sophisticated ticket

machine. Rather than starting from scratch you can simply extend the new ticket machine by the old one and add the new functionality that you require. In this example the ticket machine now can be queried to find out how many tickets are remaining:

```
Listing 2: An Improved Ticket Dispenser
public class NewTicketDispenser extends
    TicketDispenser {

    public NewTicketDispenser(int
        numberOfTickets) {
        super(numberOfTickets);
    }

    public int returnNoOfTicketsAvailable() {
        return numberOfTickets -
            numberOfTicketsDispensed;
    }
}
```

If you notice in this example it is making use of variables that are contained within the superclass, now if you bare in mind the concepts of encapsulation that were stated earlier these might not necessarily be accessible. The variable can sometimes be accessible but this is not always the case and depends on the programming languages implementation of the paradigm. It is often the case that a different modifier is used in order to allow access to variables by subclasses; and this modifier can then be used in place of the "private" modifier. In many languages this is often called a "protected" variable.

It is also possible for a subclass to inherit a method from a superclass but then actually override it (so called method overriding), this can be for a number of reasons but is often just to provide a more specialised method to achieve a similar task. So again continuing with the improved ticket dispenser example, if you would like to make it so the new ticket dispenser only gives 2 tickets at a time then you can override the methods to allow this:

Listing 3: A Different Improved Ticket Dispenser

```
public class NewTicketDispenser extends
    TicketDispenser {

    public boolean returnTicketsAvailable () {
        if (numberOfTicketsDispensed < (numberOfTickets
            - 1)) {
            return true;
        } else {
            return false;
        }
    }

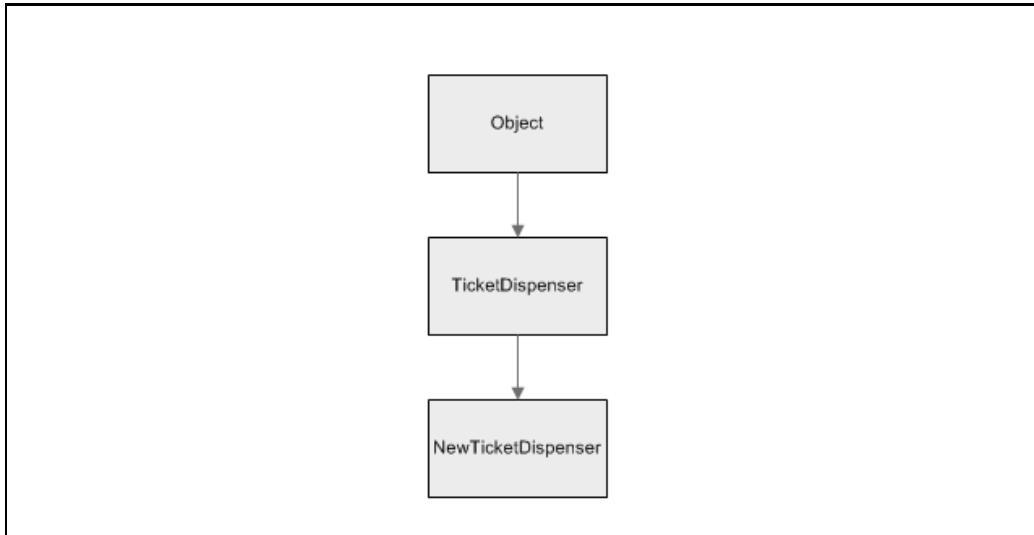
    public boolean printTicket () {
        if (returnTicketsAvailable ()) {
            numberOfTicketsDispensed += 2;
            return true;
        } else {
            return false;
        }
    }
}
```

There are a some of other more detailed aspects of inheritance that merit interest these being composite and multiple inheritance and these are are discussed in the next two sections.

2.4.2 Composite Inheritance

Composite inheritance is when there is some chaining of inheritance. This can be when you have one class which is the super class, and its subclass is itself a superclass of another class. An good example of this is when you are using the Java programming language, in which every class which itself does not inherit any classes always inherits from the object class. So in the last example the TicketDispenser class implicitly inherits from the object

class even though it is not defined. So if this is taken in the context of our NewTicketDispenser you get this kind of relationship:



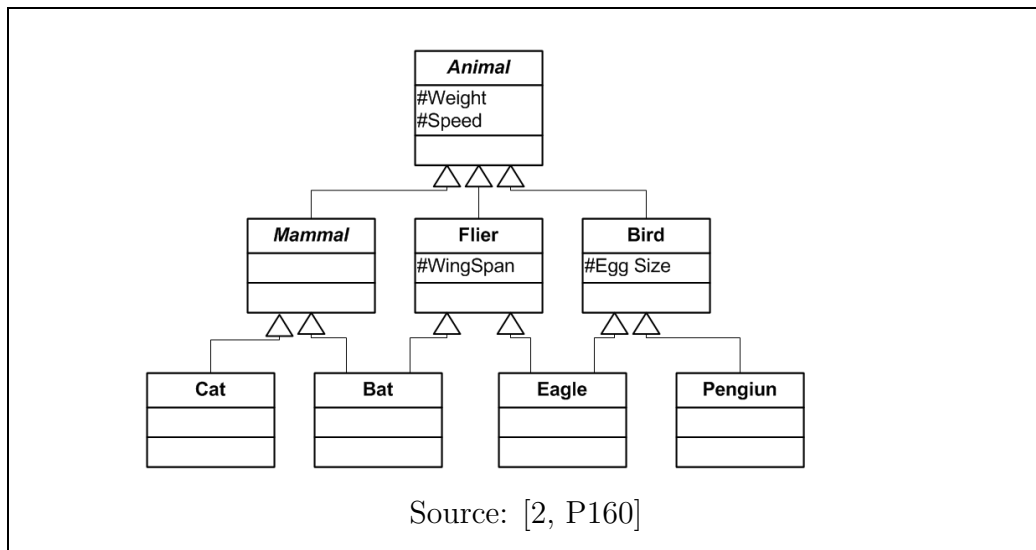
This will have implications later when we start to deal with testing issues, but is an important principle that is often used when designing and implementing systems in the object orientated paradigm.

2.4.3 Multiple Inheritance

In object orientated languages they can sometimes support what is called "multiple inheritance", this is simply where a class is able to inherit methods and variables from more than one superclass. The opposing case to this is when a language only allows a class to inherit from one class and one class alone, and this is most often called "single inheritance".

An example of multiple inheritance could be if you are modelling animals, you could end up with this kind of arrangement:

INSERT DIGRAM FROM P160 FROM Watt04

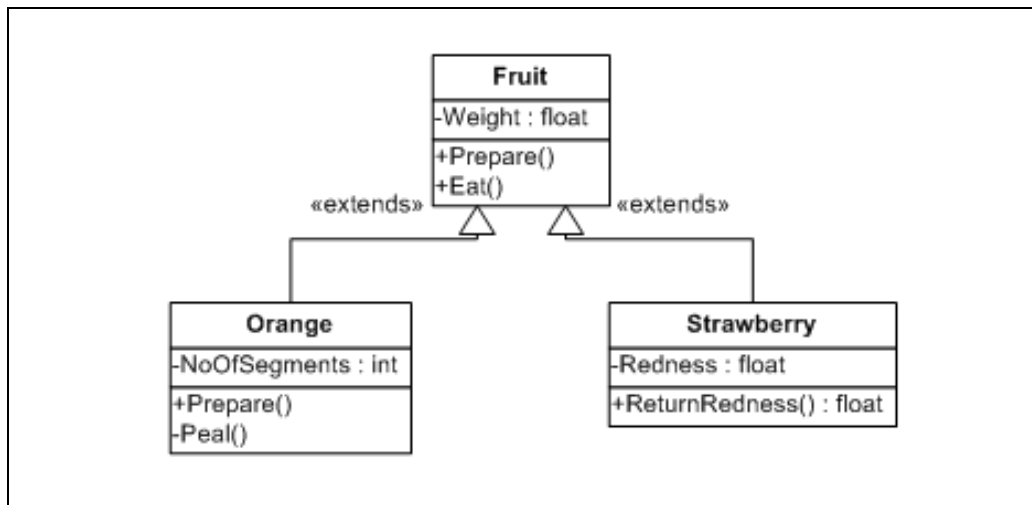


As you can see in this diagram this can result in some interesting structures, where you can see Eagle for instance, which is inheriting from two different classes which both themselves inherit from the class Animal.

There are a number of complications that can affect a class that inherits from more than one class, one of the most obvious of these is what happens when both classes that you inherit from have a method with the same name. This type of problem is often resolved in different ways depending on the language used, but in Python for instance any ambiguous methods/variables are resolved by searching the various superclasses and finding the order in which they are declared [2, P424].

2.5 Polymorphism

In the object orientated paradigm polymorphism allows you to reference and call methods on a set of objects not by their class, but by their superclass. To assist in explaining this further here is an example based on fruit:



In this example you can see that there is a Fruit super class, that defines that any Fruit object is able to be prepared in some generic way (such as washing it). The sub class Orange then overrides the extended method prepare because it has to be peeled before you can eat it, where as Strawberry can just use the extended generic fruit method.

So if in this example you wish to create some collection containing different types of fruit, and then you would like to iterate through each one and prepare them. Polymorphism rather than requiring checking which type of fruit it is before you call the prepare correct prepare method, you can just reference them as fruit, since they all have the fruit class as the super class. This is because you are guaranteed to have the prepare method due to them all inheriting from the fruit class.

Polymorphism generally takes two forms in a language, the first being static polymorphism where the binding of these dynamic objects is made during compilation, with Ada95 being an example language using this method. This is apposed to dynamic polymorphism where the binding of the methods is performed at run time instead, with Java being an example of this style of polymorphism. The first type of polymorphism has advantages because the binding is occurring at compile time if there are any errors these are found before the program is executed where in the second problems might not be found until run time which may cause faults.

3 Testing Levels for Object Orientated Systems

3.1 Introducing Testing Levels

To enable a discussion about performing testing with object orientated systems, you first have to have an understanding about the different levels of testing that are required. This is the topic that this section will discuss.

The first distinction that we need to decide when dealing with object orientated languages opposed to a traditional languages is how we define the smallest unit to be tested at the lowest level. In a traditional testing environment a single function is normally the smallest unit that is tested, which is normally been programmed and often tested by one person. It is also normally the smallest component of a system that is able to be compiled and executed on its own.

There are two options in selecting units for testing in an object orientated language, the first of these being an individual method within a class. The second option for an unit is simply taking an entire class as a unit.

If we first look at taking a single method as a unit it can quickly be seen that there are a number of issue that would have to be addressed. Firstly you can see that that it in itself might not be compilable, and even if it is, then it is possible that it might simply not run correctly. This could be the the case if the method concerned uses variables which are stored within the instantiated class and as such are not part of the method. Therefore if you wish to take a method as a unit you immediately are forced to break the concept of the individual unit, because when when performing testing you would need to have use variables external to the method.

There are of course benefits to testing at much lower levels, since it is known that if you can find a error in a program earlier in the development you can lower the costs to correct it[1, P350], but it similarly might be costly to try testing at such a low level when there can be so many other factors external to a method that can affect its operation.

Conversely if you take a class as a singular unit you can run into the problem that the units may not be small any more. In following good style guidelines classes should generally not be overly sized, but often this rule will be broken when dealing with large and complex software. This can then complicate the testing process, due to no longer attempting to test a small

simple section of code but rather a larger more complex one, where inherently it is harder to find errors.

This issue of selecting what constitutes a unit has results in two slightly different approaches, where by you can either have three or four different levels of testing. So if you take methods as a unit you have four levels with the "method level" being the extra level, and with class units you simply change the nature of the class level and don't make use of the method level.

3.2 Method Level

In this level the individual methods are tested to ensure they on their own are working as they are intended to by the specification. Although the intended testing is just for one method, it may be that other methods and variables may have to be included in the testing, to facilitate the testing of the method currently under test. This might be the case for instance if you are testing a method that calls one or more other methods within the same class, or indeed makes use of any class wide variables. The methods that are included in the testing could only be methods that have already been tested, otherwise the tests on the current method under test would not be valid. This is because if any of these methods are used as data sources by the method then if the data provided is incorrect it could cause the test to show that the method is not working correct despite the fact it may work perfectly with the correct data.

This then brings up the need for so called stub methods/variables, these are simply replacements for the real methods/variables which are often static in nature and simply return values that are known to be true by the tester. This then negates any problems with the testing of a method which calls untested methods, but of course then puts more of the burden on a detailed specification and the accuracy of the stub methods created by the tester.

3.3 Class Level

In this level the class is tested as a whole, with each of the methods being tested to ensure when a class is instantiated that the methods work correctly with the given inputs. This level is also referred to when dealing with the four level model as the intraclass testing level, where you ensure that when you interact with the different methods in the class the remaining methods behave in the correct way.

Again similarly to the method level at class level it is very possible that you might have to include other classes during the testing of a class, simply because a class is dependant upon another class, for instance if it inherits features from another. This again can require the need for stub classes as mentioned in the method level (Section 3.2).

3.4 Integration Level

In this level the focus of the testing is in finding errors in classes that cause faults during communication between different classes, and to ensure that there is a minimum chance of these faults when you move onto the system level of testing. This is an important level of testing, which is required to ensure when classes are composed together with other classes that they work as intended.

This is significant phase of testing when you start to reuse classes in ways that might not be initially expected. If the interface that has been created does not behave in the way in which it should, then when it comes to reuse the class in a later software package then the unexpected behaviour could be problematic. This is ever more an issue in particular when using object orientated systems, one of the benefits of which is supposed to be easier reuse. It can also be considered that integration testing is in some way similar to class testing, where class testing is the intraclass testing, and integration testing is interclass testing.

There are a number of different strategies for performing integration testing, a few of these include:

- Big Bang Integration - Immediately testing with all the classes together, which is rather a risky strategy and can make it difficult to find the cause of faults; but it can prove to provide quick results to whether a system is working or not if there are time constraints.
- Bottom Up Integration - Testing the lowest classes in the hierarchical tree and working up using drivers.
- Top Down Integration - Testing the highest classes in the hierarchical tree and working down using stubs.

3.5 System Level

At this final level the program is tested as a whole, and is mainly based on the specification given for the application. This therefore results in system level testing often mainly being based on functional testing techniques.

One of the main aims is to find errors that can only be found when dealing with the application as a whole. Another aim of system testing is to ensure that the application that has finally been developed does actually have all the functionality required as per the specification from the view point of a potential end user, which essentially comes down to the question whether the application is finished and can be handed over to the end user.

It can be noted that due to the high level nature of system level testing it is generally the case that the testing schemes used for traditional languages can be equally applied to those object orientated languages with good effect[1, P718]. In addition to this testing at this level is very dependent on an accurate specification of the system that is meant to have been created, this can then be used as a basis to ensure that the feature set has been implemented correctly, and that there are no underlying errors.

4 Testing Object Orientated Systems

4.1 Overview

There are a variety of effects that using languages that make use of the object orientated paradigm have in regards to testing, ranging from complications with encapsulation to the results of using inheritance and polymorphism. In this section some of these issues will be discussed.

4.2 Encapsulation

Encapsulation as stated earlier provides classes a way to control access to its methods and variables. This ability itself does not create a significant amount of bugs apart from attempting to access components that have been set to private or protected.

The main issue with encapsulation is that it can make structural testing of a class more complicated to carry out. In order to perform accurate testing you need to be able to query the state of an object at every stage of execution, and to achieve this you would require access to all of the object variables. This can be an issue because as we have seen if variables have been set to private or protected that is entirely possible that you might not have access to be able to retrieve the variables in order to be able to see the state.

The obvious solution to this is to then simply make all of the fields public, but the issue here is that although you can then directly query the state of an object at all times you have intrinsically changed the system under test. By changing the source code of the object to perform testing it is likely that some problems might not be found in the finally program that has the correct encapsulation applied. The most obvious of these is one of the methods that are being tested accesses a private variable that has been set to public for the purpose of the test.

Another option could well be to use a series of accessor methods, but again here you are changing the code which is contained within the unit under test. So if when you come to remove these accessor methods you accidentally remove an accessor method that is supposed to exist you may have created an error that would not have originally existed.

Now if you consider the example ticket machine that has been discussed previously (See listing 1), and you now have a new class that monitors the number of tickets available by directly accessing the "numberOfTickets" and

”numberOfTicketsDispensed” variables. If you remember both of these variables have been set to private/protected which means under normal conditions when trying to access them directly an error would be returned. If we now proceed with testing by making all the variables public in order to query the state of the object during testing, this system appears to be working. Of course in reality when the variables are made private again for the final completion of the program it will not work, this of course is an obstacle that needs to be overcome.

4.3 Inheritance

Earlier in chapter 2.4 the basic principles of inheritance for object orientated languages were discussed, further to this this chapter this one will continue to discuss some of the testing issues which are related to this aspect of object orientated languages.

One complication with the concept of inheritance is the trouble in understanding the actual feature set a class provides in a large system. It is entirely possible for you to have a class with only one method in it, but if it inherits from one or more classes, which themselves inherit a large number of features. This can make it hard to distinguish the precise methods and variables that the class you are testing is supposed to have, in particular when you allow multiple inheritance. This can be all the more difficult when you start taking into account overriding of methods, which can then obfuscate what a method actual does. You can of course use the specification to define what the class is supposed to have as it features, but this in itself only helps with functional testing, rather than facilitating the equally important structural testing.

An important issue that needs to be addressed when considering testing a class which extends from another is what testing is required for the various methods that are inherited/overridden. Here are a few key issues that need to be addressed:

- Do you need to retest the inherited methods, and if you do can you reuse the tests from the superclass?
- Can tests created for the superclass be reused to test methods in the superclass that have been overridden?

If you first consider the class which is being extended has already been exhaustively tested, and is known to work as intended. The issue of needing to retest the inherited methods is straightforward, and the answer is yes you do need to test them again. This is because although the actual code being executed is the same, there could be changes to methods which the method under test calls within the class that might change the expected behaviour. These changes then require for the tests to be run again, to ensure that it will work as intended. Although the elements of reuse are somewhat damaged in this inheritance system there is some light at the end of the tunnel, that being you are to use the tests designed for the superclass at least as a basis to test the method in the subclass.

Due to the method being overridden it is likely to be a specialisation of the superclasses method and this will mean that potentially some of the input conditions could have been changed although the type of the inputs and outputs will remain the same. This means that a limited amount of the tests created for the superclass may be relevant but more than likely a new set of tests will be required.

One last thought is that the reusability of the test cases is reliant on the correct use of inheritance. By this it is meant that inheritance is used in a way to represent or model a relationship, for example a class *Media*, with subclasses *DVD* and *Video*. In the example you can see that a *Media* class in some way represents both *DVD*'s and *Videos* and shares properties. In this case reusability of test cases is easier, but conversely if you simply use inheritance because it is easier than structuring the program in the correct manner then it will become more difficult if at all possible to reuse the test cases[1, P500].

However there are some benefits if you do focus on a specification based testing (functional testing) when you come to testing a subclass, you may be able to reuse a test suites [1, P75] reducing some of the effort in testing the sub class as you may only need to create test suits for the new methods.

A number of testing axioms exists that clarify some of the issues surrounding inheritances effect on testing which are antiextensionality, antidecomposition, and anticomposition[1, P505].

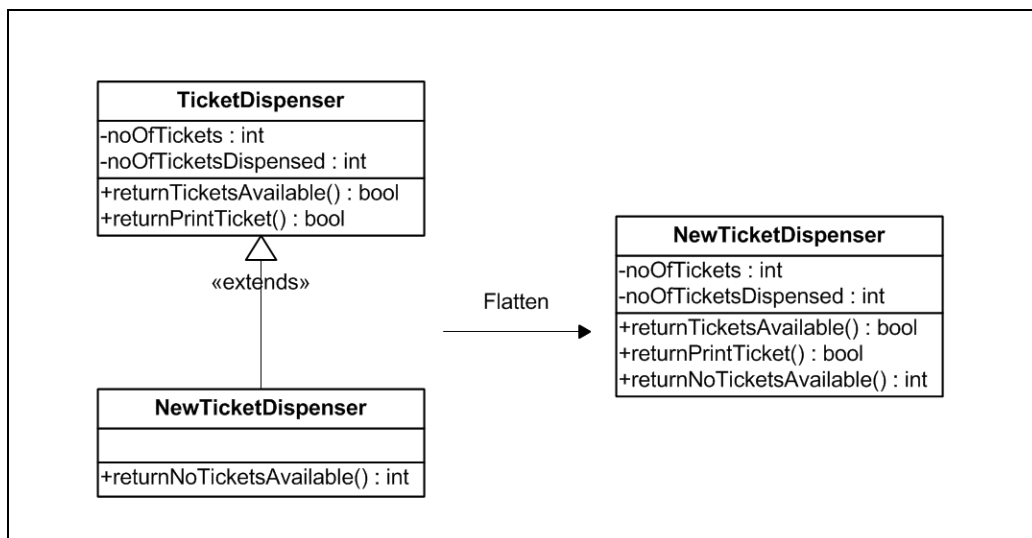
Antiextensionality means that if you have a test suite for a method in a subclass which provides good coverage it does not necessarily provide equally good coverage in another subclass. Therefore this means that if you have a set of subclasses that in order to perform high quality testing on each of them you will need to create modified test cases for each.

The antidecomposition axiom means that if good test coverage is achieved on one class in a system that does not necessarily mean that any classes that the tested class calls have been equally well covered. Therefore if you then wish to provide good levels of testing coverage you then need to create separate test suits for each class individually.

Finally the last axiom anticomposition simply means that if you have good coverage on each individual method within a class, this itself does not mean that you have tested well the class as a whole. Implicitly from this you can infer that you need to create test suits that cover not just individual methods but rather the class as a whole also, and of course this is reflected by the testing levels, namely Method testing, and Class testing.

4.3.1 Class Flattening

One solution to make testing of systems with inheritance easier is to perform what is called class flattening. This simply put flattens the class heriachy that you would normal see in a class diagram:



As can be seen in the ticket machine example that we have above if we were at this point to flatten the **NewTicketDispenser** class in order to perform testing it would result in all the methods and variables that are protected or public being transfered into the **NewTicketDispenser** class:

Listing 4: A Flattened NewTicketDispenser

```
public class NewTicketDispenser {  
  
    private int    numberOfTickets;  
    private int    numberOfTicketsDispensed;  
  
    public NewTicketDispenser(int numberOfTickets) {  
        this.numberOfTickets = numberOfTickets;  
        numberOfTicketsDispensed = 0;  
    }  
    public boolean returnTicketsAvailable() {  
        if (numberOfTicketsDispensed <  
            numberOfTickets) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
    public boolean printTicket() {  
        if (returnTicketsAvailable()) {  
            numberOfTicketsDispensed++;  
            return true;  
        } else {  
            return false;  
        }  
    }  
    public int returnNoOfTicketsAvailable() {  
        return numberOfTickets –  
            numberOfTicketsDispensed;  
    }  
}
```

Firstly it should be noted that in this demonstration of class flattening that the Java Object class has been left out simply for ease of demonstration, but it is unnecessary for this simple example.

This process then results in an explicit class which can then more easily be tested, due to all its features being present in the class rather than being spread throughout the class hierarchy. It forces the tester to then consider during the flattening process any ambiguities as a result of multiple inheritance, providing that there is an understanding of the way in which the target programming language handles these.

4.4 Polymorphism

One of the first issues that you have to consider when dealing with testing a system making use of polymorphism is that it creates complex relationships between classes that must be properly understood before you have a chance to create test suites that can test a system effectively.

A safe approach has been devised that gives testers a clear and concise way to perform a good level of testing on polymorphic classes[1, P659] when dealing with what could be potentially a huge number of tests required without some kind of culling. This approach requires that every class in the system under test has at least these three testing goals achieved during integration testing:

1. Each method in class must have complete branch coverage, therefore for each branch both possible the true and false options are taken.
2. If the class is polymorphic in nature then every method that may be bound dynamically must be tested at least once.
3. All methods called by the class under test on other polymorphic objects must be tested at least once, and each possible binding for polymorphic object must be tried.

If all these above points have been met then you would be able to say that the classes in question have been tested effectively.

5 Summary

5.1 Conclusions

In conclusion there are a range of different testing related issues that need to be considered before embarking on testing a system that has been created. The simple suggestion that because you are using an object orientated language it does not automatically mean that once you have tested a class once, that when you come to reuse it that it will not need to be tested. It is in fact rather more complicated than that, although through object orientated languages you can make it conceptually easier to reuse code, which in turn can reduce the number of errors[1, P69] careful testing strategies still need to be created.

5.2 Futher Work

There is a large variety of follow up work that can continue on from this document. One of the clearest steps onwards is to look further at the practicalities of testing at each of the different testing levels, as well as looking at the implementations of the object orinetation paradigm and how this effects the testing process in each case. Another angle that could be taken is looking at various automation efforts for each of these levels, in particular at how by using UML test case generation can be automated in some cases.

One possible avenue for further study could be to continue with simply looking in more detail at the benefits and pitfalls of the different types of integration testing (for example big band or top down), or looking at how it can be easier with the help of UML to build test suits at the system level.

Again an even deeper level you could try some practical examples of testing a system and actually try the different methods to see how practical they actually are, moving slightly away from the theoretics of testing.

References

- [1] Robert V. Binder, *Testing Object Orientated Systems*, 6th Printing, Addison-Wesley, 2005
- [2] David Watt, *Programming Languages and Design Concepts*, Wiley, 2004
- [3] Ian Sommerville, *Software Engineering*, Eighth Edition, Addison Wesley, 2007
- [4] Paul C. Jorgensen, *Software Testing: A Craftsman's Approach*, Second Edition, CRC Press, 2005