Study on “Inter Class Testing”

Naila Firdaus1 Mr.Sauod Sarwar2
1 M.Tech scholar, 2 Senior Professor, H.O.D(Computer Science)
1 Al-Falah School of Engineering and Technology, Dhauj, Faridabad, India

Abstract— A software undergoes many levels of testing from unit testing to integration to system testing. As information system are becoming more complex, object oriented paradigm is gaining popularity because of its benefits in analysis, design and coding. Conventional testing cannot be applied for testing classes because of problems involved in testing class, abstract classes, inheritance, dynamic binding, message passing, polymorphism etc. Object oriented testing methods are similar to those of conventional testing methods. Three or four levels of object oriented testing are used, depending on the choice of what constitutes a unit. If individual functions are chosen as units, we have four levels of testing namely:

1. Operation testing: It is testing of individual function i.e. the function works according to the requirement. Like for instance an ‘addition’ function does addition of two numbers and no other mathematical operation.
2. Class testing: It is testing of interaction among previously testing methods/operations. Like, the above tested ‘addition’ function works intact even after introduction of a ‘subtraction’ function in the class.
3. Interclass testing: It is a major part of out and is concerned with testing interaction among previously tested classes. Like, a class which has the mathematical operation functions when integrated with a class that contains operation for calculations of areas of various shapes, performs its mathematical operations correctly.
4. System testing: It is conducted at port level entry and is concerned with testing of the entire system as a whole before being delivered to the customer.

Of the three main levels of software testing interclass testing is the least understood both in object oriented and traditional testing terms. Interclass testing is referred as Integration testing in general terms.

Keywords: Unit Testing, System Testing, Message Passing, Polymorphism, Class Testing, Testing Strategies, Thread Integration, Configuration Integration, and Hybrid Integration.

I. INTRODUCTION

Object oriented integration testing is testing of interaction between object oriented components. Integration testing assumes that unit testing has been done on components and 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.

An interaction between objects takes place when an object calls another object by sending a message. These two objects should be instances of different classes, else it is referred as unit testing.

Integration testing occurs after unit testing. One may chose class as a core unit or the function as a core unit in unit testing. Both unit choices have implications for object oriented integration testing.

If the function is the choice, two levels of integration testing are required:
1. One to integrate functions into a full class, and
2. One to integrate the class with other classes.

II. TESTING STRATEGIES

OO does not have a hierarchical control structure so conventional top-down and bottom-up integration tests have little meaning.

I have studied four different strategies followed for Interclass Testing which are as follows:
1. Client/Supplier: Here the structure of class/supplier classes can be used to guide integration.
2. Thread Based: The thread of messages activated to service an event decides the order of integration.
3. Configuration based: The configuration of different hardwares and softwares used in the system decide the order of integration.
4. Hybrid Strategy: A mix of top-down, bottom up or pair-wise, neighbour and path based techniques can be used.

III. STRATEGIES

A. CLIENT/SUPPLIER INTEGRATION:

Some term used to distinguish objects in this strategy are as follows:

ACTOR: An object that changes state or uses other objects without receiving a message from another application object. An actor is never a recipient of a message from another object.

AGENT: An object that sends or accepts abject messages. It causes other objects to be created and used.

SERVER: An object that accepts messages from other objects but never sends any.

This integration is done by the users as follows:

We first integrate all the servers. Next we integrate agents. There may be several agent builds.

Finally we integrate all the actors. It is demonstrated as follows:

Fig. 1: Build-1 Servers
**B. THREAD INTEGRATION:**
 Threaded integration is an incremental technique. Each processing function is called a thread. A collection of related thread is called a build. Builds may serve as a basis for test managements. The addition of new threads for the product undergoing integration proceeds incrementally in a planned fashion. System verification diagrams are used for threading the requirements.

In order to deliver the software product within specified deadlines, testing and development occurs parallel. As such to facilitate the decomposition of design into manageable components the concept of thread as the elementary building block for integration is chosen. In this context, a functional thread is defined as a logical execution sequence through a series of interfacing software components resulting from or ending in the receipt of a message, event or operator interaction.

Project leads decompose requirements in the early stage of the project into groups of threads. Developers are responsible for implementing the functionality of the thread and documenting a scenario under which an integrator can verify the same. Following implementation and unit test, the developer releases the corresponding software components to the testing team where the thread is tested. After being verified with basic functionality, the developer marks the thread ready for more extensive testing. Eventually all the tested threads corresponding to different software components are integrated into the system.

**C. CONFIGURATION INTEGRATION:**
 A process of testing the system under development on machines which have various combination of hardware and software. In many situations, the number of possible configuration is too large.

These days there are various OS, memory sizes, hard drive Types available in market. If you target 10 OS, 8 memory sizes, 6 harddrives, 7 CPU’s, there are alone 10*8*7*6=33670 different hardware configurations to be tested for a single product. Adding software components such as web browser, antiviruses, makes the situation even worse.

As it is testing is exhaustive and large number of configuration is a burden more, a crucial planning effort is required. For instance one may prioritize testing different configuration based on size of userbase or risk associated with an undiscovered bug in a particular application.

**D. HYBRID INTEGRATION**
 It involves a mix of the following strategies:

1) **Incremental Integration and Regression Testing:**
 It uses top down or bottom up approach depending upon the module chosen. If the ‘Root’ module is chosen a top-down approach is followed. If a ‘Leaf’ node is chosen a bottom up approach is followed

There exists a regression test set or ‘RTS’ to hold the test case. As each module is further added, use test to check that:
- New module works alone
- New module performs its function when connected to establish working module.
- Adding new module did not break anything else

The following explains the above:
An important aspect of this testing is that we always have a system that works. We just keep adding extra functionality. It is relatively easy to find and fix problems. If everything works until we add module X and then module A stops working, the problem is likely related to limited number of cases.

2) Pair-Wise Integration Approach:
It is similar to top down and bottom up approach, but it uses real code instead of stubs and drivers. It reduces effort since no need to write code for stubs and drivers. Stub minimization and driver minimization is a major issue in testing. Developing such strategy for testing is a future scope of Interclass testing.

3) Neighborhood Integration Approach:
A typical top-down/bottom-up approach follows breadth first search in contrast to neighborhood approach which follows a depth first approach.
In this approach a module is tested and then its dependent module and then there’s dependents until a stage is reached where no further dependents are present, before moving on to the next module.

IV. PROBLEMS WITH IT
Testing object oriented software is much more complex than traditional software. The productivity gained from object oriented technology is almost lost during testing.
A complex inheritance structure creates complexity and also cost too much. To minimize the cost you should test the software in a smart way. For example you have tested the interaction between ‘Object1’ and ‘Object2’. Assume that you have a third object ‘Object3’ that inherits from ‘Object1’. When you test the ‘Object3’ you need not test the ‘Object1’ again. You should only try and test the new and refined methods. After the testing of ‘Object3’, the ‘Object2’ should be able to interact with it.
Object preserve states but the state control i.e. the acceptable sequence of events, is disturbed over an entire program. This leads to state control errors.
In case of transitive nature of objects, problem with cyclic dependencies arise in the software.
As we know testing is an exhaustive process. Moreover, when there is no top or bottom of an object structure it is difficult to test such an object. One may wonder where should you start the test or where to end it? Even if we get a start point for testing, due to complex interaction between objects, test cases generated are large. The problem is what you should test and how much should you test. Of course one may choose an alternative of selecting a few test cases. The question with this strategy is coverage. How do you know if you have selected the right set of test cases? How is it rectified that no error is induced in part of the software that is not tested?
Also there are interface errors as object oriented program have many small components and therefore many interfaces. Encapsulation is not a source of error but may be an obstacle to testing.

V. CONCLUSION
This kind of testing provides that the strategies followed can be repeated n number of times until an untested object is no longer present.

A particular advantage is that the dependencies of the individual classes and objects on one another can be found with a source code analysis of the object-oriented program and that these dependencies can be used in order to test those objects that are dependent only on objects.
One future scope of this kind of testing I think is regarding the stub and drivers which is unavoidable. Stub minimization and driver minimization if possible can prove a boon to Interclass testing.

REFERENCES