Testing Object-Oriented Software Systems: A Survey of Steps and Challenges

Clarence J M Tauro
Christ University
Bangalore, India

N Ganesan
Director (MCA)
RICM, Bangalore

Anupam Ghosh
Christ University
Bangalore, India

Nirupam Ghosh
Christ University
Bangalore, India

ABSTRACT
Object-Oriented programming is a combination of different levels consists of abstraction, class level cluster level and system level. In this article, we are going to discuss about the different testing aspects for object oriented programs. Idea is to test different testing aspects of Object-oriented Software Systems. The challenge is to cover testing with minimum effort to get maximum output.

Keywords
Object-oriented software system, Unit-testing, Model based testing, integration testing, system testing, test automation.

1. INRODUCTION
Previously computer program was nothing but just a list of commands called functions where all data getting stored in one single location which creates problem in testing the program properly.

Object-oriented testing and traditional testing are similar in few aspects. Like we use unit testing, we perform integration testing to make sure all subsystems work correctly, we perform system testing to make sure software meets the specified requirements.

Object-oriented programming language has features like inheritance, polymorphism which is completely new and brought technical challenges for tester while testing. This paper will tell how to test encapsulation, polymorphism testing along with Unit-testing, Model based testing, integration testing.

2. NEED OF TESTING
Testing is a merry-go-round process which includes a good amount of time along with cost, for all. But the reality is quite opposite, without testing it is not possible to deliver projects successfully, as during software development, developers make many mistakes throughout the different phase of development and testing helps in correcting those mistakes. In other way, testing encompasses all phases of development-in every phase; the work products of that phase are tested. So in every phase of development there is testing activity. For example, in the requirement engineering stage, the SRS (System Requirement Specification) document is written and tested to check whether it captures all the user requirements or not. The same is applicable for object oriented testing as object-oriented programming increases software reusability, extensibility, interoperability, and reliability and at the same time it is necessary to realize these benefits by uncovering as many programming errors as possible.

3. WHAT TESTING IS AND ISN’T
Testing comprises the efforts to find defects. Testing does not include efforts associated with tracking down bugs and fixing them. In other words, testing does not include the debugging or repair of bugs. Testing is a procedure of finding faults, defects in the software. While debugging is to rectify the faults, defects find during testing in the software.

4. PROBLEM AND CHALLENGES
The object-oriented paradigm has set of testing and maintenance problems. The inheritance, aggregation and association relationships among the object classes make an OO program difficult to test. The encapsulation and information hiding features result in chains of member function invocations that often involve objects of more than one class. The problems for software testing are:

1. It is difficult to understand the code and prepare the test cases.
2. It is not cost-effective to construct test stubs for member functions since most of them consist of one to two statements. Rather, one would just use them provided that they have been tested.
3. It is necessary to determine and limit the required regression tests when a function or a class is changed.
4. It requires a fresh look into the traditional coverage criteria and to extend them to include not just coverage of individual functions, but also invocation sequence, object stated and state sequences, and data definition and use path across functions and objects.[18]

5. OO TESTING
The fundamental unit of object-oriented program is class testing. The code for a class can be tested effectively by review or by executing test cases. [1] For each class, decision is taken whether to test it independently as a unit or in some way as a component of a larger part of the system. Initially we want to make sure that the requirements set forth in the specification are meeting exactly by the code for a class. The amount of attention given to testing a class to make sure that it does nothing more than what it is specified for depends on the risk associated with the class supplying extra behaviors. Any incomplete coverage of code after a wide range of test cases have been run against the class could be an indicator

that the class contains extra, undocumented behaviors. Or it could merely suggest that the implementation must be tested using more test cases.
The decisions are based on following factors:

- The role of the class in the system, especially the degree of risk associated with it.
- The complexity of the class measured in terms of the number of states, operations, and associations with other classes.
- The amount of effort associated with developing a test driver for the class.

A test driver that creates instances of the class and sets up a suitable environment around those instances to run a test case is generally used to test Classes. The driver sends one or more messages to an instance as specified by a test case, then checks the outcome of those messages based on a reply value, changes to the instance, and/or one or more of the parameters to the message.

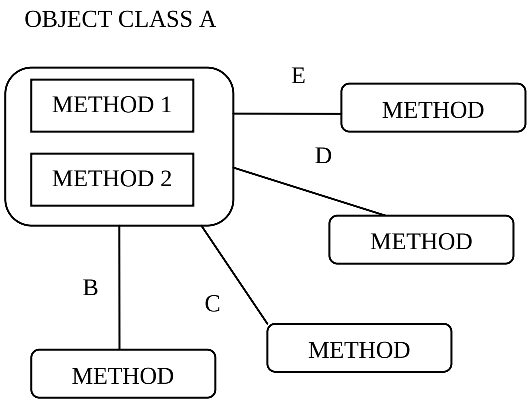

Figure1. Object Class

Figure1 illustrates an example of an Object Class "A" which has two methods. This Object Class has a dependency association with all other Classes as shown. If a change was made to the unit "Object Class A" (METHOD 2), in theory all test cases involving the unit should be rerun, at all levels. The dependency diagrams required for testing are less complex and association is easier to manage.

5.1 Encapsulation

In object oriented programming, all the operations that are performed on some data are modeled and stored within a single structure called a class. The behavior and interface (which is defined by the class' public methods) of a class are defined by the methods that operate on its instance data. In a conventional paradigm, the modeling of these two aspects is done separately. Encapsulation is about risk management, reducing our maintenance burden, and limiting our exposure to vulnerabilities —especially those caused by bypassed/forgotten sanity checks or initialization procedures, or various issues that may arise due to the simple fact of the code changing in different ways over time. Technically, encapsulation is hiding internal details behind an opaque barrier so as to force external entities to interact through publicly available access points. It makes us consider exactly how access is restricted. It also makes us consider what exactly a detail that needs to be protected is, and what exactly should be exposed to the outside world. Encapsulation minimizes the ripple effect of making a change and therefore generally minimizes the amount of regression testing required at the UNIT level.

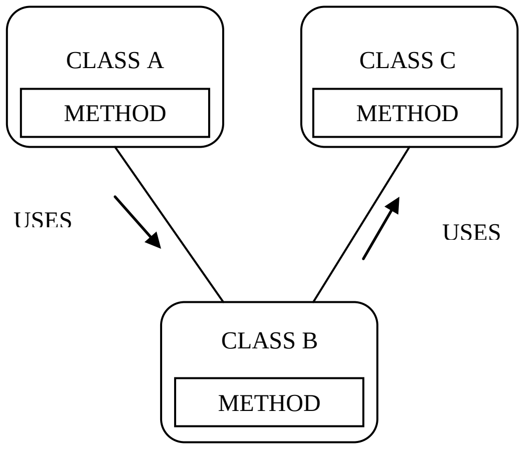

Figure2. Encapsulation

The order of unit testing can save a lot of time and effort. For example, using the dependencies shown in Figure2, if A is tested first, we require a stub for B, if B is then tested, we require a stub for C. If the testing order is reversed, C, B, A, then the tester wouldn't have to create any stubs, he/she could simply use the actual class, as a stub. [15]

5.2 Polymorphism

In the object oriented paradigm, it is possible to define a single generic interface for multiple methods with the same name that perform the same or similar operations. This helps in reducing the complexity by using or reusing the same interface to specify a general class of action.
As an example, a method used to draw graphics. If three dimensions are passed, the method draw creates a triangle, if four dimensions a rectangle, five pentagons and so on. In this case there are three different methods with the same name but they accept different amounts of variables in the method call. Generally the selection of the variant is determined at run-time (dynamic binding) by the compiler based on, for example, the type or number of arguments passed.
Few questions arise that need to be considered before testing:

1. Do we only need to one variant?
2. Do we test all variants?
3. If all, do we need to test all at all levels?

There isn't one set answer for these questions. The answers to these questions will depend on the testers, the companies policies etc. In a perfect world we would test everything. However, in reality it is generally impossible to test everything in large scale projects.
The same test cases (Driver and Stubs) could be used to test each variant at the UNIT level (test reuse). Also because of software reuse, it may not be necessary to test all the variants at the INTEGRATION level if all are fully tested at the UNIT level. Whether each variant will be tested at the system level would depend on the requirements specification.

5.3 Inheritence

Inheritance does not introduce new classes of faults, but it provides an opportunity for optimization by re-using test executions. [1] During analysis and design, inheritance relationships between classes can be recognized in the following two general ways:

1. As a specialization of some class that has already been identified
2. As a generalization of one or more classes that have already been identified[15]

Inheritance relationships can be identified at just about any time during an iterative, incremental development effort. In particular, the specialization relationship can be applied even fairly late in an effort without a large impact on most other program components. This flexibility is one of the big advantages to using inheritance and one of the strengths of object-oriented technologies.

Implementing classes is more straightforward when done from the top of the hierarchy down. In the same way, testing classes in an inheritance hierarchy is generally more straightforward when approached from the top down. In testing first at the top of a hierarchy, we can address the common interface and code and then the test driver code for each subclass. Implementing inheritance hierarchies from the bottom up can require significant refactoring of common code into a new super class.
The chances of dealing with different inheritance structures and the added possibility of potentially dealing with multiple forms of inheritance can add another level of complexity to the testing process. [15]

These issues raise a number of questions:

1. Do we completely test all BASE classes and their subclasses and at what levels should we test?
2. Do we completely test all BASE classes and only the changes or modifications in the subclasses, and if so at which levels?
3. In which order do we test the hierarchy, top down or bottom up?

To answer number 3 first, the generally accepted practice is to test top down, starting with. The levels to test at (unit, integration) are discussed in Table 1.

Scenario	Unit	Integration
None		X?
New	X	X?
Redefined	X	X?
Virtual Completed	X	X?
Virtual not Completed		

Table1. Inherited Testing

The Table1 summarizes the recommended testing at the unit and integration level for each method of inheritance.

1. None- There is no need to perform unit testing if this was done at the BASE class level. However there may be a requirement to perform integration testing if the inherited attributes (methods and /or data) are used in a new scenario.
2. New- In this case the new attributes would need to be tested at the unit level, since this is the first level at which these attributes are introduced. They would be integration tested only if there are used by another class in a scenario.
3. Redefined- In this case unit testing must be performed again since the structure of the inherited attribute has been changed. Integration testing is conducted if the attributes are used at that level in a scenario.
4. Virtual Completed. Unit testing must be performed, and integration testing if used at that level of inheritance.
5. Virtual not completed- No testing is required.[15]

6. MODEL BASED TESTING

With increasing complexity of software program, it is essential that people involve in design and development of software should communicate closely. Uniform Modeling Language (UML) gives us a standard way to create a system's blueprint, covering business process and system classes and also concrete things like classes written in a specific programming language, database schema and reusable components. The use case model is a model design based on the user's understanding/view of the system.
The modeling diagrams used in UML:

1. Use case diagrams: The use case model captures the requirements of a system. Use cases are a means of communicating with users and other stakeholders what the system is intended to do.
2. Class diagrams: Class diagrams depict a static view of the model, or part of the model, describing what behavior and attribute it has rather than detailing the process for achieving operations. Class diagrams are most useful in illustrating relationships between classes and interfaces.
3. Object diagrams: It is a special case of a class diagram. Object diagrams use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes at some point in time. They are useful in understanding class diagrams. They don't show anything architecturally different to class diagrams, but reflect multiplicity and roles.
4. Sequence diagrams: A sequence diagram is a form of interaction diagram which shows objects as lifelines running down the page, with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline.
5. Collaboration diagrams: It describes interaction among classes and associations. These interactions are modeled as exchanges of message between classes through their association.
6. State Machine diagrams: A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.
7. Activity diagrams: An activity diagram is used to display the sequence of activities. Activity diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity.
8. Component diagrams: Component diagrams illustrate the pieces of software, embedded controllers, etc., that will make up a system. A component diagram has a higher level of abstraction than a Class Diagram - usually a component is implemented by one or more classes (or objects) at runtime.
9. Deployment diagrams: A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

The vast majority of the work examining model based testing of OO systems focuses on the use of either class or state diagrams. Class diagrams provide information about the public interface of classes, method signatures, and important relationships between classes. State diagrams provide information about the behavior of a class (or the BASE class and working our way down. This enables a tester to reuse test cases, and potentially test results in certain situations set of classes.)

There are many phases in the testing process, including unit, function, system, regression, and solution testing. The following table illustrates the differences between these phases, as well as the potential UML diagram for use in the phase. [17]

Test type	Coverage Criteria	Fault Model	UML Diagram
Unit	Code	correctness, error handling pre / post conditions, invariant	class and state diagram
Function	Functional	functional and API behavior, integration issues	interaction and class diagrams
System	Operational Scenarios	workload, contention, synchron. , recovery	use case, activity, and interaction diagrams
Regression	Functional	Unexpected behavior from new / changed function	interaction and class diagrams
Solution	Inter-System communication	Interop. Problems	use case and deployment diagrams

Table2. Diagram use in different testing phases

The problem lies with UML is that during testing process it is not decided that which diagrams might be useful in various phases. Some issues need to be taken care before effectively applied UML in testing process. One issue is that it is tempting to think that the models which are derived during design and implementation can be used by tester as well. But this is not feasible reason

1. The model that derived from development process lack details in features need to develop test cases.
2. Tester gain valuable insight by building or modifying the models in the testing process.

7. INTEGRATION TESTING

Software test strategy provides the basic strategy and guidelines to test engineers to perform software testing activities in a rational way. Software integration strategy usually refers to an integration sequence (or order) to integrate different parts (or components) together.

A test model is needed to support the definition of software integration test strategies. [7]

Typical test models:

- Control flow graph
- Object-oriented class diagram
- Scenario-based model
- Component-based integration model
- Architecture-based integration model

.When Unit testing is completed then only we can perform Integration testing.

One of the biggest problems in integration testing is to determine how long to spend to test this phase as it takes almost the whole testing phase.

Driver and Stub- Driver are programs which simulate the behaviors of software components [2] (or modules) that are the control modules of a under test module. Stubs are programs which simulate the behaviors of software components (or modules) that are the dependent modules of a under test module.

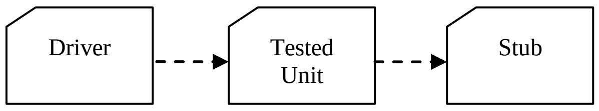

Figure3. Driver and Stub

Strategies of Integration testing for object-oriented software:-

1. On Top Down testing process the main control module is used

   as a test driver, and the stubs are substituted for all modules directly subordinate to the main control module, the subordinate stubs are replaced one at a time with actual modules. The tests are conducted as each module is integrated. On completion of each set of tests, another stub is replaced with the real module.

A Top Down approach is obviously a White Box method as in depth knowledge of lower layers of the programs functionality is required for the generation of the stub files.

2. Bottom UP Testing works in reverse of really Top Down testing. In this process the low-level modules are combined into clusters that perform a specific software sub-function. The driver is written to coordinate test case input and output. The test cluster is tested. Drivers are removed and clusters are combined moving upward in the program structure. The advantages with this are that once a layer has been completely tested, it’s less likely that any Bug found has occurred in that layer.

Some other integration testing strategies are:

1. Execution based integration test – Tracing the execution of an interaction. This testing strategy finds control flows that cannot be executed.
2. 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.
3. Function based integration testing – Tests the correct provision of functionality through the component’s collaboration. This testing strategy focuses on detecting mismatches between the interpretations of the interaction between components.

8. SYSTEM TESTING

The last phase of testing is System testing which comes just before the product is delivered to customer. This level of testing ensures that the program matches the final specification that was drawn at beginning of the project. [9]The most important thing in this phase is that it not concerned on how system works rather it is more concerned with the result produced. For this reason, system testing is considered to be ‘Black Box testing’. System Testing should take place in the setup which is accurately reacts like the system in which the product will be deployed. By this way any error occur can be easily caught and fixed before final product release.

It seems that system tests for object oriented systems would be no different to system tests for non-object oriented systems. Specifications for object oriented software can be very different from non-object oriented software. For example, object-oriented systems can be modeled in UML and use `Use Case' and `Class' diagrams. These diagrams are then used to produce the test cases.

9. REGRESSION TESTING

Regression testing is performed similar to traditional systems to make sure previous functionality still works after new functionality is added. [3] On changing a class which has been tested previously implies that the unit tests should be rerun. The test scenario may have to be adapted based on the changes done to support proper testing. In addition, the integration test should be redone for that suite of classes.

10. TEST AUTOMATION

Not but the least automated testing always play vital role in delivering product with good quality at proper time. Test automation is software that automates any aspect of testing of an application system with a capability to generate test inputs and expected results. It reduces the repetitive works that we do manually and also provide us the result as ‘pass’ or ‘fail’.

The appropriate extent of automated testing depends on our testing goals, budget, software process, kind of application under development, and particulars of the development and target environment.

Automated tests ensure a low defect rate and continuous progress, whereas manual tests would very rapidly lead to exhausted testers. To summarize the characteristics of tests we are aiming at: [19]

- Tests run the system – in contrast to static analyses.
- Tests are automatic to prevent project members to get bored with tests (or alternatively to prevent a system that isn’t tested enough)
- Automated tests build the test data, run the test and examine the result automatically.
- Success resp. failures of the test are automatically observed during the test run.
- A test suite also defines a system that is running together with the tested production system. The purpose of this extended system is to run the tests in an automated form.
- A test is exemplar. A test uses particular values for the input data, the test data.
- A test is repeatable and determined. For the same setup the same results are produced. [19]

11. CONCLUSION

The conclusion of this short expose is that it throws some light on different testing aspects of object-oriented software system. And to full fill the challenge ‘to cover testing with minimum effort to get maximum output’, we need automate the test since only an automated test can cover the code in combination. In testing object-oriented systems, the test is moved up to a higher level of abstraction, where test automation is absolutely necessary.

12. ACKNOWLEDGMENT

We are heartily thankful to Prof. Jebrial, Prof. Joy Paulose, whose encouragement, guidance and support enabled us to develop an understanding of the subject.

13. REFERENCES

[1] Mauro Pezz`e, Michal Young,” Testing Object Oriented Software”, Proceedings of the 26th International Conference on Software Engineering (ICSE’04), IEEE, 2004

[2] Priti Bansal, Sangeeta Sabharwal, and Pameeta Sidhu, “An Investigation of Strategies for Finding TestOrder During Integration Testing of Object Oriented Applications”, International Conference on Methods and Models in Computer Science, IEEE, 2009

[3] Suganya G,Neduncheliyan S,”A Study of Object Oriented testing techniques: Survey and challenges”,IEEE Conferences,2010

[4] John D. McGregor and David A. Sykes, ”A Practical Guide To Testing Object –Oriented Software”Mar, 2001.

[5] Perry D., Kaiser G.: “Adequate Testing and Object–oriented Programming”, Journal of 00-Programming, Vol. 2, No. 5,Jan. 1990,

[6] Gareth Thomas, ”Object Orientated Integration Testing”, December 14, 2006;

[7] Jerry Gao Ph.D., “Software Integration Testing”, Jan 1999

[8] Jilles van Gurp, ”Object Oriented testing”, December 9,1998

[9] Dafydd Vaughan, “System Testing with Object-Oriented Programs”, Jan 12,2007

[10] Binder, R. “Testing Object-Oriented Systems. Models, Patterns, and Tools”, Addison-Wesley, 1999.

[11] Myers, G. “The Art of Software Testing, John Wiley & Sons”, New York, 1979.

[12] Jilles van Gurp, “Object Oriented Testing Report”, 1998

[13] Grady Booch, Ivar Jacobson and James Rumbaugh,”The Unified Software Development “,Process.1999

[14] GradyBooch, RobertA.Maksimchuk , MichaelW.Engel, BobbiJ.Young, Jim Conallen, Kelli A. Houston, “Object-Oriented Analysis and Design with Applications (3rd Edition)”, 2007

[15] Maj Nicko Petchiny, ”Object Oriented Testing”,April 1998

[16] Jitendra S. Kushwah, Mahendra S. Yadav, Testing for Object Oriented Software” Indian Journal of Computer Science and Engineering(IJCSE),Feb 2001

[17] Clay E.Williams, ”Software Testing and the UML”, International Symposium on Software Reliability Engineering(ISSRE99),Bocs,Baton,1999

[18] D.Kung, J.Gao, P.Hsia, F.Wen,” Change Impact Identification in Object Oriented Software Maintenance”Software maintenance,Proceeding, International Conference,IEEE,1994

[19] Bernhard Rumpe,” Model-based Testing of Object-Oriented Systems”, International Symposium, FMCO 2002