A Comparison of Software Testing Using the Object-Oriented Paradigm and Traditional Testing

Jamie S. Gordon
jamie.s.gordon@unf.edu

Robert F. Roggio
broggio@unf.edu

School of Computing
University of North Florida
Jacksonville, FL 32224 United States

Abstract

Software testing is an important part of any software development. With the emphasis on developing systems using modern object oriented technologies, a critically-sensitive issue arises in the area of testing. While traditional testing is reasonably well understood, object oriented testing presents a host of new challenges. This paper focuses on what differentiates the two in test cases, testing levels, and OO features affecting testing.

Keywords: Object-oriented testing, traditional testing, testing levels

1. INTRODUCTION

Object-oriented testing is based not only on both the input and output of an object's methods, but also how that input and output may influence the object's state. Many of the positive features touted by object-oriented languages can map directly into increases in testing complexity. While the many beneficial features of the object-oriented paradigm are important, the increases in program complexity (sometimes in unintended and unseen ways) often negatively impacts testing in terms of effort and time.

Traditional testing involves the viewing of input and output of a program in a procedural manner. Both types of testing still involve tried and true testing types. In fact, many of the differences show up in white-box testing because the two types of programming can often solve the same problems using the same input and output.

This paper seeks to determine how testing is different in an object-oriented paradigm versus that of a traditional (procedural) program.

2. LITERATURE REVIEW

Research done on object oriented testing has changed over the years. Many early papers written on the subject lamented the inability of researchers to address the differences between object-oriented programs and procedural testing. As Turner and Robson pointed out, "the vast majority of research conducted into the testing of object-oriented programs fails to address the difference between the object-oriented and more traditional programming techniques," (Turner & Robson, 1993).

At around that same time, Hayes wrote a paper identifying some aspects of object-oriented programs and how that may affect systems (Hayes, 1994). The paper also described a

testing methodology that the author believed should be recommended for OO-programs. Hayes was successful at identifying the problems involved with inheritance, but did not discuss many other issues in much depth, such as polymorphism and dynamic binding. This was a problem with many early papers on the subject, which primarily focused on objects' states and inheritance. It became increasingly evident that many more features of object-oriented programming need to be considered for testing by looking at more recent papers, such as that of Jain, "Testing Polymorphism in Object-Oriented Programming," which described how advanced polymorphism makes it difficult to understand all the possible interactions among classes (Jain, 2008).

Gu, et al., wrote a paper detailing three processes to select test data and for evaluating the coverage of those tests. They were the flow-graph-based approach, graph-based class testing, and the ASTOOT approach, which used algebraic specification to determine test cases. These methods derive test data, for example the flow-graph-based process uses the flow of control from method to method to model test data while the graph-based process models transitions between different states of an object. These focused on program flow and state changes and ignored other features of OO as well. The authors also discussed how the testing of object-oriented programs must differ from traditional testing methods (Gu, et al., 1994).

By 1996, there was already enough literature for Johnson to report on the different testing levels and techniques proposed by researchers (Johnson, 1996). However, there was not much of a consensus at that point for a standardized system of testing. For example, there was a disagreement on unit-testing, in that some authors disagree that it should be involved in object-oriented testing at all.

As time went on, the differences between object-oriented and traditional (procedural) testing became more evident. For example Khatri, et al., described many features – encapsulation, inheritance, polymorphism, etc. – of object-oriented programming and how they made it more difficult to decide how test should be done (see *Object-Oriented Features that Affect Testing* below) (Khatri, et al., 2011). However, that paper did not describe in detail how testing should be done. Bhadauria described the same features, but also gave a sequence of testing levels and what kinds of tests should be run in each (Bhadauria, 2011). Some authors described design metrics that may help programmers determine beforehand how difficult to test their design may be (Badri, 2012). This sort of empirical view of object-oriented testing is another useful area of study. Authors have discussed both new and older metrics for measuring testability, and which are the most valuable to object-oriented programming (Yeresime, et al., 2012).

3. BACKGROUND: OBJECTIVES OF TRADITIONAL AND OBJECT-ORIENTED TESTING

There are many people with vested interests in the testing process, including programmers, testers, program managers, and end-users. These people are some of the stakeholders in the system, those that are impacted by the system or influenced by its behavior. Individuals or groups of individuals acting in these roles are those who depend on testing to show the systems performs as intended. The major objective in testing is to discover as many faults, errors, and defects as possible with minimum effort and cost (Khatri, Chillar, and Sangwan, 2012). According to Johnson (Johnson, 1996), "Testing is the process of executing a program with the intent to yield measurable errors." Testing is not about showing that there are no errors – effective testing comes from creating effective test cases that can coerce out errors and failures (Naik & Tripathy, 2013). This can help designers find 'defects' (term attributed to design) and programmers find 'faults' (term normally attributed to programming). Given this backdrop, however, what constitutes an effective test is quite different when contrasting traditional (procedural) testing and object-oriented testing (Dechang, Zhong, & Ali, 1994)

4. TEST ADEQUACY AXIOMS

Elaine Weyuker defined eleven axioms to determine the adequacy of a test set (Hayes, 1994). Some are less interesting as they apply equally to both testing paradigms, such as the applicability axiom (Every program has an adequate test set), the monotonicity axiom (It is possible to create a set of test cases that is larger than is necessary), the renaming axiom (If P is simply a renaming of Q, and T is adequate for Q, then T is adequate for P), or the non-exhaustive applicability (Program P is

adequately tested by T, where T is a non-exhaustive test set). Below is a list of the axioms to consider when discussing the difference between traditional and object-oriented testing. (Table 1)

Axiom	Description	Traditional	Object-Oriented
Complexity	For all n, there is a program that is adequately tested by a test set of size n, but not by a test set of size n-1	There is a minimum set of inputs that must be tested	There is a minimum set of inputs and object states that must be tested
Anti-extensionality	There are programs P and Q that compute the same functions (semantically similar), where T is adequate for P but not for Q	It cannot be assumed that the same test cases can be used for different programs that accomplish the same things	It cannot be assumed that the same test cases can be used for functionally similar programs, this can be extended to mean that just because one state is correct for one program, that does not mean that is correct for a similar program
General Multiple Change	There are programs P and Q that are syntactically similar, where T is adequate for P but not for Q	Syntax does not tell you what needs to be tested	The syntax of two programs does not determine the test sets, this also means that if two programs use the same classes, the test cases should be different because the messages sent between them may be different
Anti-decomposition	There is a program P and component C where T is adequate for P and T' is the subset of T that can be used for Q, but T' is not adequate for Q.	A component of a program (say a method) can be adequately tested for use within one program, but not necessarily on its own.	"When a new subclass is added (or an existing subclass is modified) all the methods inherited from each of its ancestor super classes must be retested."
Anti-composition	There exist programs P and Q and a test set T where T is adequate for P and the subset of T that can be used for Q is adequate for Q, but T is not valid for P;Q (the composition of P and Q).	Two programs (or methods) can be adequately tested on their own, but once combined or used in another class; they may no longer be adequately tested.	"If only one module of a program is changed, it seems intuitive that testing should be able to be limited to just the modified unit. However, [this] states that every dependent unit must be retested as well."

Table 1 Test Adequacy Axioms (Hayes, 1995)

5. TEST CASES

Traditional Test Cases

Test cases are often based on the traditional model of processing. The traditional Von-Neumann model of processing is in Figure 1.

Figure 1 Von-Neumann Model of Processing (Labiche, Tosse, Waeselynck, & Durand, 2000)

This model works well for the procedural paradigm where the input dictates what the output of a program is. In accordance with this model, a test case ignores the processing aspect and focuses on input and output. One may thus view a test case as an ordered pair: <input, expected output> (Naik & Tripathy, 2008). This can be done because the processing is only dependent on the input into the application, as there is no program 'state' to consider, necessarily.

The expected output of a system would normally be described as either values produced by the program or messages to the user based on the input (Naik & Tripathy, 2008). The rationale behind this assertion is justifiable, as the output is program-generated and defined structurally rather than behaviorally (Johnson, 1996). This also implies test cases may be derived from static analysis for dynamic testing (discussed ahead).

Object-Oriented Test Cases

Test cases in the object-oriented paradigm are more complex. The traditional testing model is insufficient, because objects in a program have their own states which may well be impacted by the processing of input parameters (Turner & Robson, 1993). In addition, these state changes may not at all be evident from the output of a program. For example, consider a program that has objects of this class:

Student
-name: String -grades: int[]…
+addGrade(int grade): void +sortedGrades(): int[]…

Figure 2 Student Class Examples

In this example, only the important methods are listed. Suppose a test case is developed for sortedGrades(), where sortedGrades() is supposed to sort the grades array and then return the sorted values. Consider the following test case:

<add grades: (100, 50, 75),
expected output of sortedGrades(): 50, 75, 100>

Figure 3 Test Case with Output Results

These tests might pass with the traditional test model. However, without examining the state of the Student object, it is unknown whether the grades array has actually been altered, or if the sortedGrades() method simply returns a sorted array of integers without actually altering the grades array. The method sortedGrades() is designed to return the grades[ ] array as a sorted list without affecting the grades[ ] array itself. The reason for this is so that the user may specify in the interface that they want grades in ascending order by percentage. However, they may also want the grades in the order that they were entered, so it is important to preserve the original state. This means that not only should the expected output of the method be tested, but the expected state of the class should also be included in the test case. (Figure 4)

<add grades: (100, 50,75),
expected output of sortedGrades(): 50, 75, 100},
expected state of grades[]: {50, 75, 100}>

Figure 4 Test Case with Output and State

Due to this trait of the object-oriented model, the Von-Neumann model needs to be changed to accommodate the state of the objects involved in processing. Robson and Turner suggest the following adaption (Figure 5):

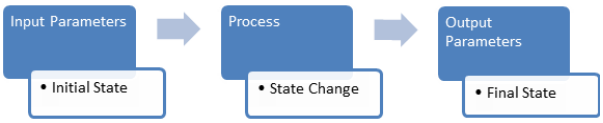

Figure 5 Von Neumann Model with Added State Changes (Turner and Robson, 1993)

Indeed, Dechang, et al., 1994 agree that an effective test case involves both the changing class state and the sequence of operations. The object-oriented paradigm is based on objects as instances of classes; therefore programming is inherently state-based. Not only that, but an object's internal values are not the only thing to consider when developing test cases. The associations between objects through method calls, inheritance, polymorphism, etc. make object-oriented test case generation much more complex (Johnson, 1996) (discussed ahead). For now, it is important to note that there is no strict input-process-output correspondence in object-oriented programming. For more advanced testing, where a method chain is involved for example, it is recommended that a few more items are inserted into the test case: (1) a list of messages and operations that either will or may be executed by the test, (2) any exceptions that may or are expected to occur, and (3) any environmental setup external to the program (Bhaudaria, et al., 2012). This is in addition to any supplementary information deemed necessary.

6. TESTING LEVELS

There are many between object-oriented testing levels and traditional testing levels. While

object-oriented programming provides functionality not afforded by procedural programming paradigm – such as data encapsulation and reuse of objects – the ease of writing object-oriented programs does not translate to testing. In fact, many researchers have observed that testing programs written in an object-oriented language increases the effort required for adequate testing (Jain, 2008). In this section, four levels of testing are described. Typically, in traditional testing, there is unit and system testing. With object-oriented testing it is necessary to include two new levels, class testing and integration testing.

Unit Testing

Unit testing can be used in both object-oriented and traditional testing. Methods and routines are tested independently of each other in unit testing (Johnson, 1996). The defects or faults of other classes and functions should not impact unit testing (Roggio & Gordon, 2013). In traditional testing, tests can be made by defining inputs and observing to see if the output of a method or set of methods matches the expected outputs of the function. These functions or methods need to be independent units, units which do not call other methods or use common global data (Hayes, 1994). However, in object-oriented unit testing, the method cannot interact with other classes or be dependent on its class's methods. Testing individual methods is significantly more difficult. In fact, some authors state that unit testing cannot be deduced from one object's operations because (when isolated) one may not see the object's relations to other methods, the class's state, and other classes (Labiche, et al., 2000). Instead, it is suggested that unit testing be combined with integration testing (Hayes, 1993).

To actually accomplish unit testing on individual methods, several additional items must be tracked. The first is any attributes of the class that may be changed by calling the method. The second is that other methods in the class called by a particular method are determined to be correct. The third is that objects of other classes used by the method must be first tested and determined to be correct. This means that the testing levels are not in a linear order, and have to be determined from a different method. There are many different ways of determining levels such as the flow-graph-based and graph-based techniques mentioned earlier (Dechang, et al, 1994).

Another way of dealing with dependencies when trying to unit test is simulating the dependent classes. This is an extension of a testing technique known as writing drivers or stubs. Traditionally, drivers and stubs were written as "dummy" methods for dependent methods. In object-oriented testing, this is extended to entire classes. A driver is written when a class is dependent on another for data to process. A driver is usually used with a lower layer in a hierarchical development model. A stub is a method written that is handed data to process when the module that processes data has not been written yet, or when the module that has been written has not been tested. Stubs are often written when testing higher classes in a hierarchical design.

Class Testing

This version of testing involves testing methods as they relate to and interact with one another. Of course, because this is "class testing," it is only involved in object-oriented testing (Johnson, 1996). Some authors consider this to be object-oriented testing's version of unit testing (Johnson, 1996) (Labiche, et al., 2000). The reasoning behind this is that testing a class's methods in isolation, without any relation to other methods, is not actually useful for any nontrivial task. Methods are meant to interact. In any event, the purpose of class testing is to test how a single class's methods interact with one another. Again, this means that any classes referred to by an object's methods need to be tested thoroughly beforehand, or the dependent classes need to be simulated in some way.

Cluster Testing

Cluster testing involves extending class testing to verify that a group (cluster) of cooperating classes interacts correctly. According to Johnson**,** (Johnson, 1996), a cluster of classes is a group of classes that are dependent and cooperate with one another directly. Traditional testing does not appear to have a clear comparison. In order to do this, the cooperating classes must have previously been tested individually, through class and unit testing if possible. (See next paragraph)

Integration & System Testing

In traditional testing, integration testing tests methods together. This will include methods that are dependent on other methods or dependent on common global data (Hayes, 1994). For object-oriented programming, integration testing is an extension to cluster

testing. In integration testing, the testing is extended to the system as a whole. The clusters are combined into the total system, which is then tested as a whole, with all the dependencies intact. Another, more specialized, case of integration testing is system testing which is running the whole system based on normal customer usage scenarios as close to the customer's environment as possible (Johnson, 1996).

7. OBJECT-ORIENTED FEATURES THAT AFFECT TESTING

There are many positive features of object-oriented programming, and although they make the paradigm very effective, these features make it more difficult to test. The seven factors described below have been mentioned by different authors as factors that affect the amount of effort needed for adequate testing (Khatri, Chillar, & Sangwan, 2011) (Badri & Toure, 2012) (Jain, 2008) (Yeresime, Jayadeep, & Ku, 2008). The seven factors are encapsulation, inheritance, polymorphism, cohesion, coupling, dynamic binding, and abstraction. As described below, each contributes to greater difficulty in designing tests (other than cohesion that eases it) over traditional testing. The feature will be described and then its effect on testing over traditional testing will be given.

Encapsulation

Encapsulation is used to restrict access to some of an object's attributes and methods. When a program is written procedurally, then this is not as much of an issue because programs are typically full units whose private or protected methods are not modified by outside programs. In object-oriented programs, it can become difficult to observe object interactions with encapsulation, especially when variables and methods are not visible outside a class (Khatri, et al., 2011). This restricted visibility means that it might be more difficult to be aware of an object's state, which is important because private fields and methods can be affected, such as with getters or setters. Testing is therefore more difficult when the state of the object is important for a class test case and strong encapsulation is used (Bhadauria, Kothari, & Prasad, 2011). Of course, it is also important for class and cluster testing because a class's or other classes' methods may influence an object's state. If part of that state cannot be observed, then it will be difficult to not only design test cases, but also to observe testing results.

If it is the case that an object is strongly encapsulated, it is important to find a way to verify that private fields are correct if they are modified by other classes (Jain, 2008). The ability to control a test's input may also be difficult because the initial state cannot be determined, either (Badri & Toure, 2012). This might mean creating new methods to display a class's state, which may or may not go against the class goals as designed (Badauria, et al., 2011). Perhaps the attribute was designed to be invisible to all objects due to security concerns.

Inheritance

It does not make much sense to talk about inheritance in the case of a procedural program. The only near comparison is the reuse of methods or structures, but this should not be as complex as in object-oriented programs. On the other hand, inheritance is a method of sharing attributes and behavior from pre-existing classes to other subclasses. When one class is a subclass of another, it does not guarantee that all the inherited methods are still correct if they have been verified in the superclass (Khatri, et al., 2011). The superclass being well tested will not mean that all the classes that inherit it will be correct. Any new methods or attributes that have been added to the subclass may affect properties inherited by the subclass. Yeresime et al (Yeresime, et al., 2012) describe an empirical measure of inheritance, Depth of Inheritance Tree or DIT. DIT refers to the maximum length of a path from a class to the root class in an inheritance tree (Badri & Toure, 2012). The deeper a class in the tree, the higher the number of methods that can be inherited; this makes its behavior more complex, more difficult to predict, hence more difficult to design effective test cases (Yeresime, et al., 2012).

These issues result from invisible dependencies between parent and child classes. A child cannot then be tested without its parent class because errors in behavior might easily propagate down the inheritance tree (Badauria, et al., 2011). Another issue may arise when an inherited method is changed in the subclass, but the subclass has an untouched, inherited method that uses the changed method. The overridden method and untouched inherited method need to be tested. In the example ahead, getNum() of

Child may be called, but it will return a value that would be unexpected by examining Parent. (See Figure 6)

```
public class Parent{
   int num;
   public void setNum(int n){
         num = n;
   }
   public void getNum(){
         return mult();
   }
  public int mult(){
         return num * 2;
   }
}// end class

public class Child extends Parent{
   public int mult(){
         return num * 4;
   }
}// end class
```

Figure 6 Inheritance Example

Yeresime et al also define a metric for measuring a different kind of complexity, Number of Children (NOC) (Yeresime, et al., 2012). NOC is the number of immediate sub-classes in a class hierarchy and is meant to be a measure of the influence a class may have over the system as a whole (Badri & Toure, 2012). This would be used as part of cluster and system testing to determine how much emphasis should be put into testing that particular class.

These two metrics, DIT and NOC, are taken from the Chidamber and Kemerer metric suite. They can be used to determine the overhead involved in testing. The advice given by Yeresime is that if DIT is greater than six, then design complexity is high and testing overhead can be large. If the NOC is similarly high, then the design of abstract classes is diluted. The abstraction of classes is not utilized or the designed abstract classes are too general. Yeresime et al, also state that these metrics are untrustworthy for determining faults themselves and cannot be used to measure fault-proneness (Yeresime, Et al., 2012). It is difficult to assign inheritance a precisely measurable metric at this point, as inheritance comes in many forms, and inheritance trees can become very complex. It is therefore important– while still using inheritance effectively – to try to keep inheritance as simple as possible.

Polymorphism

Procedural languages do not have a very good comparison to polymorphism. Polymorphism allows attributes of an object to take multiple forms or data types. In addition, an operation may return more than one type of data or may accept more than one type of data for parameters (Khatri, et al., 2011).

Polymorphism is crucial to object-oriented programming and helps make it versatile and reusable (Bhadauria, et al., 2011). But all the different forms an object may take should be tested. A class or group of classes should be designed well enough so that the overhead required to test is low. For example, a class such as shown in the next figure is not advisable:

```
public class Foo{
        Object o1;
        Object o2;
 }// end class
```

Figure 7 Object can Morph into any other Class.

The reason for this cautionary note is that *o1* and *o2* can take almost any form because Object is the superclass of all objects in the Java language. This would make testing a Herculean task as *o1* and *o2* could become almost any data type. Polymorphism should still be used, but the attributes of a class should be more limited and well-defined, in regards to both design and testing. As stated by Hayes (1993), "testing should be used to ensure that data abstraction and value restriction are implemented properly." In Figure 8, Shape is a class that has three subclasses, Triangle, Square, and Circle. There are still nine possible combinations that *shape1* and *shape2* can take, but this is much more manageable than the first example. The testing of attributes should be done in unit and class testing. Other testing concerns include methods with return values that are polymorphic as well as parameters that are polymorphic. This would more readily be accommodated in cluster or system testing.

```
public class Bar{
        Shape shape1;
        Shape shape2;
  …
 }// end class
```

Figure 8 Well Defined, Limited Attributes

According to Jain (Jain, 2008), the first major type of polymorphism is called "ad hoc

polymorphism." This type of polymorphism is considered completely syntactical, that is, entities are polymorphic only because they share a name and do not have to be behaviorally linked. This can best be explained by examining its first subtype, overloading. Overloading refers to separate methods which share a name. These methods may have completely different parameters and method bodies. A group of overloaded methods can be treated as completely separate methods from one another during testing without any extra effort. The second type of ad hoc polymorphism, coercion, is a conversion from one class or data type to another during code execution. This is also fairly easy to test because the conversion type can be determined statically from code. For example, when 0.50 * someInt + 6.9 is executed, the integer someInt will be converted to a float or double to coincide with the data type of 0.50.

A second major type of polymorphism is called universal polymorphism. This is considered to be "true" polymorphism, and refers to an object being able to become many different data types (Jain, 2008). This is an umbrella for two subtypes, inclusive and parametric polymorphism. Inclusive polymorphism is polymorphism where a subclass can be used in place of a superclass (see *Dynamic Binding*). In parametric polymorphism a method or object can be written in a generic manner through parameters which are given a class value when the object is instantiated. In this way parameterized types are not "written in stone," which implies they are not dependent on any one class. An object or function can be used in different contexts without any conversion or run-time testing needed in this type.

These types of polymorphism should be taken into account for testing. Indeed, understanding all the interactions that can result from the polymorphic nature of some objects can be very difficult (but necessary) to keep in mind when developing test cases (Jain, 2008). Ad hoc polymorphism is less testing intense because the tests can be derived statically. Universal polymorphism, as the name might imply, is more difficult to test because the forms an entity can take may be wide-ranging.

Cohesion

Cohesion is a measure of the degree to which the methods of a class create a single, well-defined class (Khatri, et al., 2011). In procedural programming, cohesion refers how well a module of code (typically a file) belongs together as a single unit. Most of the rest of this discussion talks about how a class is cohesive in terms of instance variables. Procedural programs do not have instance variables, but instead information is passed between methods as parameters. Therefore, cohesion in the procedural realm is concerned with methods dealing with similar parameters and functionality. In OO, if a class is cohesive (its methods contribute to the class as a single unit) then the class is reusable, more reliable, and more easily understood. Cohesion is related to coupling; if there is high cohesion, there is low coupling and vice-versa (Khatri, et al., 2011). High cohesion means that the methods within a class are similar in the variables used and the tasks they perform. This means test data are easier to create and more easily understood. Low cohesion means that there are many different types of data that need to be defined for a specific class (Yeresime, et al., 2013). This complexity in design leads to higher costs of testing, and renders testing itself more error-prone.

Another defined metric for this testing factor is the Lack of Cohesion in Methods (LCOM). LCOM is defined as the mathematical difference between the number of methods whose instance variables are completely dissimilar, and the number of methods whose instance variables are shared (Yeresime, et al., 2012). See Figure 9.

Consider the three sets of instance variables for a class with three methods: 1: {a, b, c, d, e}l 2:{a, b, e}, and 3: {x, y. z}

Figure 9 Method Cohesion

Methods 1 and 2 have shared instance variables, and, therefore, have cohesion. However, 1 and 3 and 2 and 3 have no shared instance variables, and therefore no cohesion. In this case, the LCOM would be one (2 non-cohesive method pairs – 1 cohesive method pair). If LCOM is high, it means that a class is not cohesive (and might be a candidate for refactoring into two classes. At the testing stage, a class will need to have different testing sets for the different methods rather than one testing set for the entire class. This leads to confusion and overall complexity of the testing process.

LCOM is found in the same suite as DIT and NOC (the Chidamber and Kemerer metric su*ite*) (Yeresime, et al., 2012). The authors Badri and

Fadel (Badri & Toure, 2012) found that LCOM and lines of code (LOC) were the most predictive testing metrics over DIT, NOC, (both previously discussed) and CBO which is described below.

Coupling

Coupling is a measure of the dependency between modules. Strong coupling is undesirable for many reasons, chiefly of which is that it prevents the change of components independently of the whole. (Also, many feel strong coupling cohesion is the antithesis of highly-valued high cohesion, which arguably results in low coupling) – Strong coupling means that all (or many) of the methods coupled together need to be understood as a set, instead of each class operating as its own unit (Khatri, et al., 2011). Strong coupling negatively contributes to testing, because it implies that unit testing cannot be done effectively. In good design, coupling is kept to a minimum especially for a large or complex system where coupling could result in cluster and system testing absorbing the majority of testing resources (Badauria, et al., 2011).

An additional Chidamber and Kemerer metric is Coupling Between Objects (CBO) (Yeresime, et al., 2012). CBO is a count of the number of classes to which a class is coupled (Badri & Toure, 2013), and represents still another measure of complexity. High CBO leads to less reliability, and the higher interoperability between classes causes unit testing to be difficult (Yeresime, et al., 2012). However, some interoperability is generally required for object-oriented programming as objects need to be able to communicate in some way. This implies the necessity of cluster testing.

Other metrics for software complexity are efferent coupling (Ce) and afferent coupling (Ca). These come from the R. C. Martins metric suite (Yeresime, et al., 2012). Efferent coupling occurs between packages and is the measure of all the classes external to a package that are used within the package (See Figure 10). In contrast, afferent coupling between packages counts all the classes external to a package that are dependent on the classes within a package. (See Figure 11) In conjunction, these two help measure the stability of a package as a whole (Yeresime, et al., 2012), where stability is measured $I = \frac{Ce}{Ce+Ca}$ (Scale 0 to 1 with 0 absolute stability; 1 absolutely unstable.) Stability, in a sense then, is a measure of how well a package can adapt to change. In a testing sense, stability can imply how changes in one particular package might impact other classes. If this impact is high due to high instability, then much regression testing must be done in other packages as part of cluster or system testing.

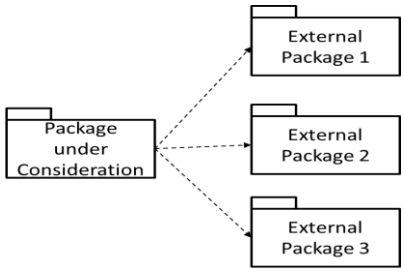

Figure 10 Efferent coupling (Ce)

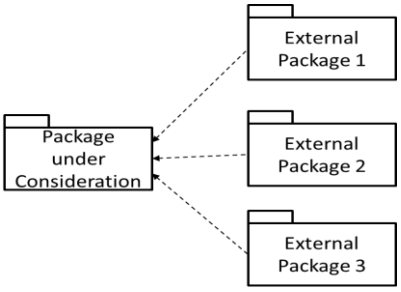

Figure 11 Afferent coupling (Ca)

Dynamic Binding

Dynamic binding is a result of either inclusive polymorphism or type parameterization polymorphism in some languages. For example, in Java the return type of a function or even the types of some fields can be decided at run-time rather than compile time: (Figure 12) Dynamic binding introduces concerns when deciding how to design test cases, because the exact data type of attributes cannot be known statically (Khatri, et al., 2011) .

```
public class Foo <E> {
   E field;
   public E getField(){
      return field;
   }
   public void setField (E field){
      this.field = field;
   }
}
```

Figure 12 Dynamic Binding is Determined at Runtime

The class in Figure 12 may be instantiated in many ways and with many data types through parameterization. Thus when designing test cases in parameter polymorphism, it may only be necessary to test based on how other classes in the program will instantiate Foo. Those data types that are used for parameterization in other classes must be considered for testing cases, but others need not be. This would be a part of cluster or system testing. Unit testing or class testing would be difficult to accomplish because it would not be known (without looking at the system as a whole) how Foo might be instantiated. The behaviors and properties of E might be incredibly varied. Class and unit testing should therefore probably not be done in this case of dynamic binding, due to its complexity as a unit. With the whole system, it can most likely be determined which data types E might take.

Inclusive polymorphism**,** in contrast, may be a simpler form of dynamic binding to test (Jain, 2008). This is because it is often known what classes inherit a superclass. In this way, it can be known what types an object may be bound to at run-time. All of these dynamic bindings must be included in a test case.

Abstraction

An abstract class is a type of class that cannot be instantiated. An interface is used as a template for other classes. If the class is just abstract and not an interface, it provides useful methods as well as variable fields (Khatri, et al., 2011). However, these defined methods cannot be tested directly and analysis is done from their subclasses, because one may not instantiate an abstract class or an interface in most languages (Badauria, et al., 2011). … This can lead to major overhead. If an abstract class is inherited by more than one class, how many of those child classes should be tested? Even if the child classes have not overridden the inherited methods, all would need to be tested because the abstract class itself cannot be tested, directly.

An R.C. Martins metric described by Yeresime et al is abstractness (Yeresime, et al., 2012). Abstractness is a ratio of the number of abstract classes/interfaces to the total number of classes in a package. This measure is used with the instability measure (see *Coupling*) to create a line graph (Yeresime, et al., 2012). (See Figure 13) These points along the "main sequence" line are considered to be balanced between abstraction and stability. These are well designed and more easily tested. This means that more abstract and unstable, the more difficult to test.

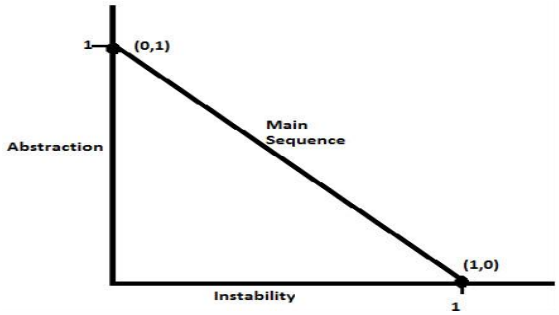

Figure 13 A-I Graph (Yeresime, et al., 2012)

8. CONCLUSION

Object-oriented testing is based not only on both the input and output of an object's methods, but also how that input and output may influence the object's state. . While the many beneficial features of the object-oriented paradigm are important, the increases in program complexity (sometimes in unintended and unseen ways) often negatively impacts testing in terms of effort and time. Some of these features, like cohesion help lower the amount of testing required, but others cause testing efforts to rise.

Traditional testing involves the viewing of input and output of a program in a procedural manner. Test cases tend to be one dimensional. In object-oriented testing, however, test cases are two dimensional, because changes in an object's state must be considered. Traditional testing involves both unit and system testing, while object-oriented testing requires class testing (for how the methods of a single object work together) and cluster testing (for how coupled objects change each other's' states). Thus, it is important to note that verification testing (the testing done by the developers) has been truly changed by the object-oriented paradigm, while validation (that done by the end-user) has not.

Moving forward, it will continue to be important to define more and better ways for testing object-oriented programs. Some already exist, but they are wide-ranging and there has been no major consensus as to what the best way to test is or what factors are most important in testing. Most focus on the fact that order to test object-oriented modules is not as definite as in traditional programs, where the order of tests

follows a procedural path. In object oriented testing, an object may send a message to another object at any time.

9. REFERENCES

Badri, Mourad, and Fadel Toure, "Empirical Analysis Of Object-Oriented Design Metrics For Predicting Unit Testing Effort Of Classes," Journal Of Software Engineering & Applications 5.7 (2012): 513-526.

Bhadauria, Sarita Singh, Abhay Kothari, and Lalji Prasad, "A Full Featured Component (Object-Oriented) Based Architecture Testing Tool" International Journal Of Computer Science Issues (IJCSI) 8.4 (2011): 618-627.

Gu, Dechang, Yin Zhong, and Sarwar Ali. "On Testing of Classes in Object-Oriented Programs" Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research

Hayes, Jane Huffman. "Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach," Notes in Computer Science, Vol. 858 (1994): 205-220.

Jain, Ajeet K. "Testing Polymorphism in Object-Oriented Programming." ICFAI Journal of Computer Sciences 2.4 (2008): pages 43-53.

Johnson, Jr., Morris S. A Survey of Testing Techniques for Object-Oriented Systems, Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative research (CASCON '96)

Khatri, Mrs. Sujata, Chhillar Dr. R. S., and Sangwan Mrs. Arti "Analysis Of Factors Affecting Testing In Object-oriented Systems," International Journal On Computer Science And Engineering 3 (2011): 1191.

Labiche, Y., Thevenod-Fosse, P., Waeselynck, H., and Durand, M.-H, "Testing Levels for Object-Oriented Software," Proceedings of the 22nd International Conference on Software Engineering, pages 136-145

Naik, Kshirasagar and Priyadarshi Tripathy Software Testing And Quality Assurance: Theory And Practice. John Wiley & Sons, 2008 pages 7-27

Turner, C.D. and Robson, D.J. "The State-Based Testing of Object-Oriented Programs," Software Maintenance, 1993 CSM-93, Proceedings., Conference on Software Maintenance pages 302-310, 27-30 Sep1993

Yeresime, Suresh, Pati Jayadeep, and Rath Santanu Ku "Effectiveness of Software Metrics For Object-Oriented System," Procedia Technology, Volume 6, 2nd International Conference on Communication, Computing &Security [ICCCS-2012] 420-42