A New Method for Estimating the Testability of Polymorphism in Class Hierarchy

Jin-Cherng Lin
Department of Computer Science and Engineering, Tatung University, Taipei 10451, Taiwan
E-mail: jclin@ttu.edu.tw

Yun-Liang Huang
Department of Computer Science and Engineering, Tatung University, Taipei 10451,Taiwan
E-mail: huang.strecker@inventec.com

Abstract-*Software testing is one of the most common ways for assuring quality of software system, and software testability be also recognized as new part of the software quality at the same time. Conventional software testing methods are divided into two categories: static testing and dynamic testing. Most of testing methodologies fall into the category of dynamic test, due to the face that more information can be derived during programs execution. On the contrary, in object-oriented software test, the feature of inheritance and polymorphism produce new testing obstacle during dynamic testing. Our research in this paper will propose a tester perspective to assess the polymorphism in design stage and this way also provide useful information for developer to probe into the fault that is hidden from test in early stage. In addition, from our polymorphism RATO model, we recognized a new factor of testability in design stage that is referred to as reachability.*

We think that the main contribution of this paper is providing an alternative tester perspective for developer and let developer can estimate the polymorphic behavior in class inheritance hierarchy that may cause of test obstacle at design phase. At the same time, this is first literature to propose the polymorphism RATO model to model polymorphic behavior, and use it as basis of analytical unit to decompose inheritance class hierarchy. Besides, we overcome the interdiction of lack of testing information during design stage in the past. The metric of testability on polymorphism, generated by the polymorphism RATO model, can reveal defects that hide at design stage in class hierarchy, it can provide most early information for developer on modify, redesign the system and find another way to fix these defect as well.

Keywords: Descendant path, Dynamic binding, Inheritance, Polymorphism RATO model, Reachability, Testability

1. Introduction

The software testing is one of the most common ways for assuring quality of software system. Conventional software testing methods are divided into two categories: static testing and dynamic testing. In static testing the program is analyzed without executing it, while in dynamic testing the program is executed. Most of testing methodologies fall into the category of dynamic testing, due to the face that more information can be derived during programs execution. Functional testing and structural testing are two major approaches in dynamic testing. They are also called black-box testing and white-box testing. In functional testing, tests are constructed based upon the program's functional properties, ignoring its internal structure. In structural testing, the internal control flow structure or data dependencies are used to develop the testing methodology to conduct the testing [6].

As for static test method, most of researches are through analyzed the program or system structures to establish its related complexity or other software metric (e.g. Testability, Dependencies, Coverage and so on) in order to evaluate whether system or programs can hide the fault from test[14][15]. Through software metrics, we can evaluate the accuracy of design at develop stage or to be used as basis to establish the adequacy test plan at test stage.

Testing software is a difficult process, in general, and sufficient resources are seldom available for testing. From quality standpoint, testing is done throughout a development effort and is not just an activity tacked on at the end of a development stage to see how well the developer did. We see testing as part of the process that puts quality into a software system. As a result, we address the testing of all development products even before any code is written since it is costly to redesign a system during implementation or maintenance in order to overcome a lack of testability. This notion is discussed in [9] as well.

1.1 Potential test issues in object-oriented software

Most of people seem to believe that testing object-oriented software is not much different from testing procedure-oriented software. While many of the general approaches and techniques for testing are the same or can be adapted from traditional testing approaches and techniques, but our experiences and current research papers indicated; in fact, they are different and presented a lot of new challenges and test obstacles. The polymorphism is a key feature of object-oriented programming which showed a new technical challenge to tester.

The power of polymorphism brings the expressiveness of programming languages. It also brings a lot of new anomalies and fault types. We refer to all of these problems as a new obstacle in object-oriented test. Unfortunately, the techniques that are used to eliminate faults in procedure-oriented programs are not as applicable to those found in object-oriented programs [10].

We know the method calls with polymorphism, because of the dynamic binding characteristic, the program code of the actual execution cannot be predict. It will dynamic be decided at the run time. We called this polymorphism headache is yo-yo path phenomenon [1] and illustrates its actual call path with figure 1. Therefore, it is very difficult to observe and track executed path in test processes. Besides, tester also hard to understand polymorphic behavior meaning for its source code even though have documented. Consequently, Tester cannot design adequacy test case to expose hidden errors that will result in low readability. Hence, tester can not design the test case of the adequacy to expose hidden errors. These will lead to low quality of software. Finally, the inherent complexity of the relationships found in object-oriented program also affects testing. There are other potential faults to be discussed in the past literatures [8], [10] and [13]; we summarized it as follows;

- The interactive complexity between components.
- The data coupling with intra-class and inter-class.
- The object dependency between classes.

In other words, from the test perspective, the object-oriented programming with polymorphism is hard to produce an adequacy test case because you don't know which type of object will be executed in runtime and you can't analyze the testability of program code in static test as well because you don't know which fragment of code will be executed. An object-oriented program with polymorphism, from test viewpoint, it is very difficult to produce a test procedure no matter when it is on software test or measurement of testability. Therefore, that's why we say object-oriented software have potential testing issues.

1.2 Goal of this research

This paper will present a model for the appearance and realization of object-oriented faults in polymorphism and discusses specific categories of inheritance and polymorphic faults. The model and categories can be used to support empirical investigations of object-oriented testing techniques, to inspire further research into object-oriented testing and analysis, and to help improve design and development of object-oriented software. Finally, we will base on this model, and then we will provide a technique to measure polymorphism testability during design stage. The situation of object-oriented design can't be assess can be overcome through our research. This is what all object-oriented software measures can not do it before. This research addresses the metric of polymorphism for object-oriented program measure in design stage.

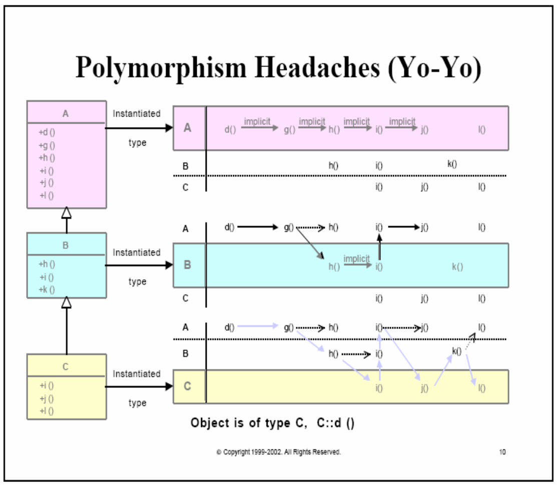

Figure 1. Yo-Yo path headaches

2. Polymorphism RATO Model

We will show a simple type and class hierarchy diagram in figure 2 to illustrate the mechanics of polymorphism RATO (reference attachment to object) model. Although the model is called a class diagram, we thought of it as a type diagram from type-oriented perspective, and therefore each of rectangles in the figure will represents a type. In this example, each of polymorphic method call can be modeled by RATO model (see figure 3.). To understand these, consider the class hierarchy diagram shown in figure 2. We use type declaration and definition to create class hierarchy and then all of polymorphic behavior can be mapped to polymorphism RATO model in figure 3 [13].

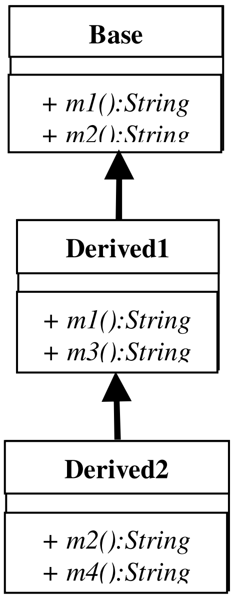

Figure 2. Simple type and class diagram

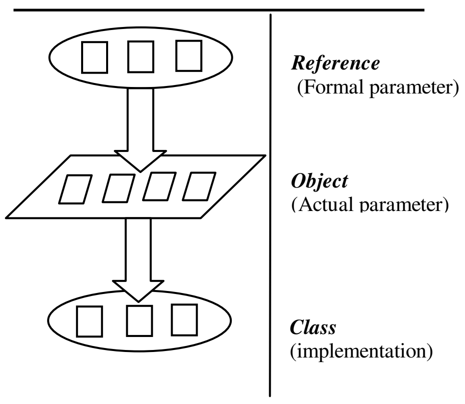

Figure 3. Polymorphism RATO model

Follow the polymorphism RATO model, if a statement base on figure 1 and declared as below;
Derived2 derived2 = new Derived2()；
This statement declares an explicitly classed reference variable, derived2, and attaches that reference variable to a newly created Derived2 class object. The top panel in figure 2 depicts the Derived2 reference variable as a set of portholes, through which the underlying Derived2 object can be viewed. There is one hole for each Derived2 method. The actual Derived2 object maps each Derived2 method to appropriate implementation code. What is problem here? For example, the Derived2 object maps overrides the m1() implementation in class Base. As far as the reference variable derived2 is concerned, that code is reachless. That is meaning that some implementation code may be inaccessible under test when polymorphism occurred. There is polymorphism declaration base on figure 2 class diagram. If the statement declaration is
Base base= derived2；
There is absolutely no change to the underlying Derived2 object or any of the method mappings, though methods m3() and m4() are no longer accessible through the Base reference. These also mean that some implementation code may be inaccessible under test when polymorphism occurred. This polymorphism RATO model illustrates the shielding effect on polymorphism. From tester perspective, we thought these fragment codes are easily hiding fault from test and hard to produce adequate test case that will affect the testability and quality of software system. We also define a concept of descendant path base on polymorphism RATO model. Using the concept of descendant path, we can enumerate all of polymorphism possibility and decompose a class hierarchy to analyze polymorphic behavior, in which we recognize descendant path as primary building block of polymorphism in class inheritance hierarchy. From polymorphism RATO model, we know that sometimes methods in object will hide from different reference variable attachment. This shielding effect never discussed in any literatures of polymorphism measurement and it will affect tester to produce the test case. Basically, this effect will decrease software testability. We recognize this phenomenon as new factor of DFT (Design for Testability) for polymorphism issue. This new factor is addressed as reachability. In the past researches, polymorphic complexity was measure by number of method overridden. In this paper, we use this new factor and descendant path's complexity to improve the testability of polymorphism measurement. It differs greatly from controllability and observability that are key factors in testability measurement previously.

3. Polymorphism measures in class hierarchy

From the polymorphism RATO model, we understand and use the concept of the descendent-path as basis for observing all polymorphic behavior on the class hierarchy diagram. What is descendant path? We define the descendant path as follows:

Definition: Descendant path (DP). In the class hierarchy, a descendant path is the set of classes crossed by a path going from the root class to a leaf class.

We also propose an algorithm here for finding descendant path in class inheritance hierarchy. This algorithm use breadth-first searching (BFS) algorithm to traverse every class in hierarchy from root node to leaf node.

Algorithm for finding descendant path
Input: a class hierarchy
Output: descendant path
Step 1: build a class inheritance hierarchy data structure and initiated each Θ_{dp}. Such as,
$\Theta_{dp}=\{\text{node} \mid \exists! \text{ node} \in \text{Root class node}\}$
Step 2: use BFS algorithm to traverse all root class nodes, from root to leaf node, in the process of traverses, descendant node will be add into its descendant path(Θ_{dp}).
Step 3: Unite every descendant path (Θ_{dp}) on leaf node will get final descendant path set.
$\Theta_{dp} = \bigcup_{i=1}^{nbLeaf} \theta_{dp}(i)$; suppose that the total number of leaf node is *nbLeaf*.

Using the notion of descendant path as building block, we can decompose class hierarchy and assess its testability in design stage. The mathematical formula is given by:

Definition: The complexity of descendant path with respect to inheritance. Let *DP* be descendant path and *h* be the height of *DP*, the complexity for *DP* is:

$$CP(DP) = h \times (h-1)/2 \qquad \text{Eq. 1}$$

Definition: Polymorphic metric for each descendant path. Let *P* is a path involved in the polymorphism. $A_1,\ldots,A_{nbAttach}$ are each of polymorphic behavior in polymorphism RATO Model. The polymorphic metric for each descendant path is given by:

$$PolyMetric(P) = \sum_{i=1}^{nbAttach} PoPM(A_i) + PoHM(A_i) \qquad \text{Eq. 2}$$

Next, let A is a polymorphism reference attachment to object combination. The probability of polymorphic method on each polymorphic behavior is given by:

$$PoPM(A) = \frac{NOVM}{NOM} \qquad \text{Eq. 3}$$

Where NOM denotes number of object method and NOVM denotes number of overridden method. Moreover, the probability of hidden method on each polymorphic behavior is given by:

$$PoHM(A) = \frac{NOM - NRM}{NOM} \qquad \text{Eq. 4}$$

Where NOM denotes number of object method and NRM denotes number of reference method.

Definition: Testability of polymorphism in inheritance hierarchy. Let *IH* be an inheritance hierarchy and $P_1,\ldots, P_{nbDP}$ are the path involved in the polymorphism. The testability of polymorphism in *IH* is given by:

$$ToP(IH) = \sum_{i=1}^{nbDP} \frac{1}{PolyMetric(P_i) \times CP(DP_i)} \qquad \text{Eq. 5}$$

4. Conclusion

Most of people know that the purpose of the software test is used to improve software quality. However, the method and concepts of the traditional programming language test have no longer applied on test of the object-oriented system. From the research thesis of the object-oriented system that have announced, we understand that test of the object-oriented system polymorphism is the most difficult process, and furthermore its related research for announcing was few in the past. Generally speaking, it is easy to obtain more testing information during the period of dynamic test. On the contrary, polymorphism testing broke this rule. The tester has a new challenge in polymorphism test that is due to its dynamic binding characteristic. Moreover, the polymorphic behavior also confuses developer, that is to say, polymorphism is easy to hide the fault from testing and hard to produce an appropriate test case. Consequently, the polymorphism becomes an obstacle in test process and affects the software quality in final. For resolving these problems, in this paper we propose a another kind of developer's standpoint, through the polymorphism RATO model accurate understanding polymorphic behavior, moving the testability estimated of test stage to design stage. Thus, the developer can evaluate system testability on design stage. Furthermore, developer can use these information thinking whether developer redesign the system or improve software testability by program skill during implementation. We believe that there are no other ways to fully cover the test of polymorphism in dynamic testing or static testing. We also believe that our research point out a new direction of polymorphism test.

We think main contribution of this paper is to provide alternative tester perspective to developer and let developer can estimate the polymorphic behavior in class inheritance hierarchy that may cause of test obstacle at design phase. At the same time, this is first literature to propose the polymorphism RATO model to modeled polymorphic behavior, and use it as basis of analytical unit to decompose inheritance class hierarchy. Besides, we overcome the interdiction of lack of testing information during design stage in the past. The metric of ToP(IH)(Eq.5), generated by the polymorphism RATO model, can reveal defect that hide at design stage in class hierarchy, it can provide most early information to developer on modify or redesign the system or find another way to fix these defect as well. Nevertheless, notion of descendant path is unique technique to be used to analyze object interaction in object-oriented test. Our paper makes other contribution is the polymorphism RATO model reveal mechanism of polymorphism in which developer can truly understand the polymorphic behavior. Developer can get some hints of polymorphism to resolve design issues by way of polymorphism RATO model. They also can use these kinds of information to increase system testability by concept of abstract class and control a number of methods overridden to avoid its complexity to grow up too fast. Carefully using the concept of abstract/virtual class in hierarchy design will dramatically reduce complexity of class hierarchy. In addition, the polymorphism RATO model also provides a lot of information to create the guideline for developer that can assist developer to avoid violated design rules of natural. Finally, we recognize a new testability factor in DFT that is never find in the past, reachability, it differs from exist factors of testability on testability measurement or previously definition. Most of factors in DFT are qualitative because it is not easily to do quantitative analysis but we make it possible. In the paper, we used different perspective mining the implicitly factor that is embedded in object-oriented design and propose a unique notion to model and decompose the class hierarchy and then accumulated related data to generate a quantitative metric. The quantitative metric can help developer significant reduced the defect of design to be implemented into the software system. Not everything discussed above all can be doing effectively in the past. Well, the developer was not afraid of polymorphism design because of they can use this research to get most early design information to assess its defect in object-oriented

5. Future work

In addition to above-mentioned matter, we will try to find other barrier in the way of object-oriented test. Otherwise, design pattern and template is also a new testing obstacle in object-oriented programming. In addition, the concept of coupling relation between subroutine in testing procedure-oriented program or the dependencies relation between software components also adapt to estimate the relationship between objects in object-oriented software. Using control flow structure and data dependencies between objects' interaction that maybe is a way to find other metrics to assess polymorphism in design stage. Furthermore, we will keep research and announce our studies in this field.

References

[1] Roger T. Alexander, A. Jefferson Offutt, "Criteria for Testing Polymorphic Relationships", 11th International Symposium on Software Reliability Engineering (ISSRE '00), pages 15-23, San Jose CA, October 2000.

[2] Roger T. Alexander, A. Jefferson Offutt, "Analysis techniques for testing polymorphic relationships." In Thirtieth international conference on Technology of Object-Oriented Languages and Systems (TOOLS30), pages 104-114, Santa Barbara, CA, 1999.

[3] James Bach, "Attributes of Software Testability", VeriTest.com Tester's Network, http://www.veritest.com/.

[4] Benoit Baudry, Yves Le Traon, and Gerson Sunyé, "Testability Analysis of a UML Class Diagram", Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS.02), 4-7 June 2002, Page(s): 54-63

[4] Robert V. Binder, "Design for Testability in Object-Oriented Systems", COMMUNICATIONS OF THE ACM September 1994/Vol.37, No.9

[6] Chi-Ming Chung, Ming-Chi Lee, "Object-Oriented Programming Testing Methodology", Software Engineering and Knowledge Engineering, 1992 Proceeding., Fourth International Conference on, 15-20 June 1992., Page(s):378-385

[7] Chi-Ming Chung, Ming-Chi Lee, "Inheritance-Based Object-oriented Software Metrics", TENCON '92. Technology Enabling Tomorrow: Computers, Communications and Automation towards the 21st Century 1992 IEEE Region 10 International Conference, 11-13 Nov. 1992, Page(s):628-632 vol. 2

[8] B.H. Liskov and J.M. Wing, "A Behavioral notion of subtyping", ACM Transactions on Programming Languages and Systems. Vol. 16, No. 6, November 1994. Pages 1811-1841.

[9] John D. McGregor, David A. Sykes, "A Practical Guide to Testing Object-Oriented Software", Addison-Wesley, 2001.

[10]Jeff Offutt, Roger Alexander, "A Fault Model for Subtype Inheritance and Polymorphism", The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01), pages 84-95, Hong Kong, PRC, November 2001.

[11]William E. Perry, "Effective Methods for Software Testing-Second Edition", Published by Wiley & Sons, Inc.

[12]Bret Pettichord, "Design for Testability", Pacific Northwest Software Quality Conference, Portland, Oregon, October 2002

[13]Wm. Paul Rogers, "Reveal the magic behind subtype polymorphism ", http://www.javaworld.com/javaworld/jw-04-2001/jw-0413-polymorphp.html, April 2001

[14]Jeffery Voas, Larry Morell, Keith Miller, "Predicting Where Faults Can Hide from Testing", IEEE Software. March 1991, pp.41-48.

[15]Jeffery M. VOAS, Keith W. MILLER, "Software Testability: The New Verification", IEEE Software, Vol. 12, No.3: May 1995 pp. 17-28.