ANALYSIS OF THE CLASS DEPENDENCY MODEL FOR OBJECT-ORIENTED FAULTS

Pranshu Gupta and David A. Gustafson
Department of Computing and Information Sciences, Manhattan, Kansas, USA

ABSTRACT

A test order should be an integral part of all OO testing criteria. It defines the order in which classes in a program should be tested, and accordingly defines the order of test executions. The object relation diagrams (ORD), test dependency graphs (TDG) and class dependency model (CDM) creates a test order based on the inheritance and association relationships between classes and methods of a program. In this paper, we apply the class dependency model to object-oriented (OO) programs seeded with a set of standard object-oriented faults. We analyze where the OO faults are concentrated in the hierarchy of the testing order of classes created using the CDM. Based on this analysis, we show that the approach for defining the test order should be different for various categories of OO faults (such as inheritance, method overloading, polymorphism, etc) as these object-oriented program properties cause different behaviors in the program.

KEYWORDS**:** *Software testing, CDM, ORD, TDG*

I. INTRODUCTION

Various diagrams have been defined to show the relationships between classes and used for defining test orders for OO programs. There are diagrams such as object relation diagrams (ORD) [1, 2, 4, 11, 12 and 13], test dependency graphs (TDG) [1, 2, 4, 9, 10 and 12] and class dependency model (CDM) [1, 4] that help in defining a test order.

The ORD model shows the inheritance relationship and the association (A) relationship between classes but does not provide the details of the association relationship [1]. The lack of detail about the interaction between classes leads to a potential weakness that can give rise to more stubs and thus more testing effort. An example of ORD model is shown in figure 1. On the other hand, the CDM diagram takes into account two types of OO relationships: Inheritance relationship (I) and Use relationship (U) both at the class level and the method level as shown in figure 2a and 2b. The CDM is generated from the ORD by adding the details of the association relationship between classes [1]. Thus, CDM shows the inheritance relationship as well as the association relationship including the detail of the association relationship (e.g. a use relationship between methods of different classes) as shown in figure 2b. CDM model shown in figure 2a can be generated using the object relation diagram (ORD) as well as by the static analysis of the source code [1]. The model is further enhanced to the method level as shown in figure 2b. CDM has been used to generate test orders for the OO systems [1].

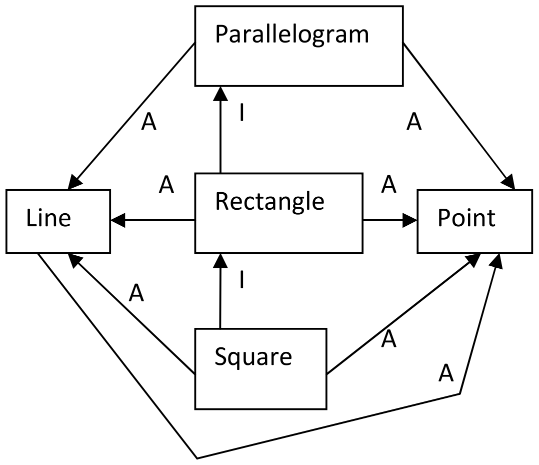

Figure.1. Example of ORD model

The TDG is also an extension of ORD and includes relationships at the method level. Literature research shows that TDG does not distinguish between different relationships between classes and methods and thus is not a good strategy for defining test orders [4].

A test order based on the ORD model tests the base classes preceding the derived classes. The classes are categorized based on inheritance and association between classes. Based on this model, the test order tests the classes at a higher level in the inheritance tree before the classes at the lower level [4]. Thus, both ORD model and the CMD model test the base class group before testing the derived class group. But in these models, further ordering between these class groups is based on different type of relationships between classes. ORD is based on the association relationship and CDM is based on the use relationship. When using the source code for creating test orders, they may or may not be the same.

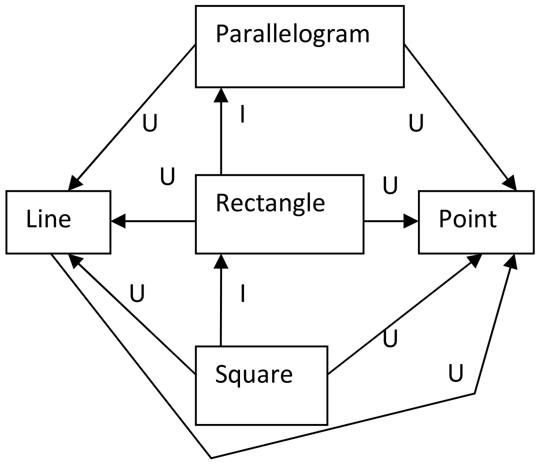

Figure.2a. Example of CDM model

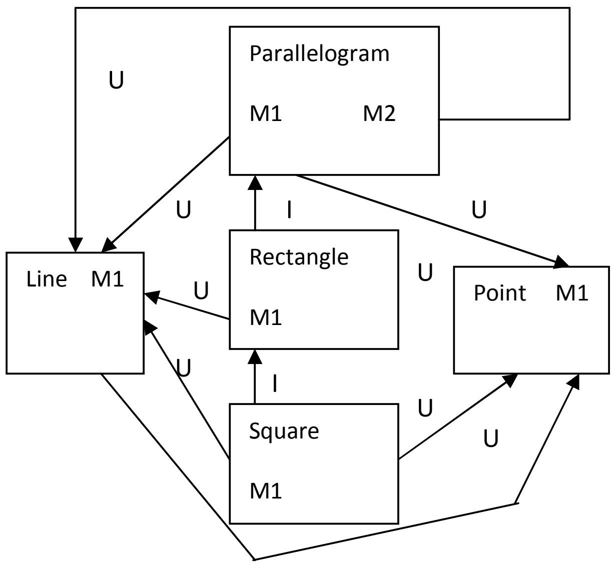

Figure.2b. Example of CDM model at method level

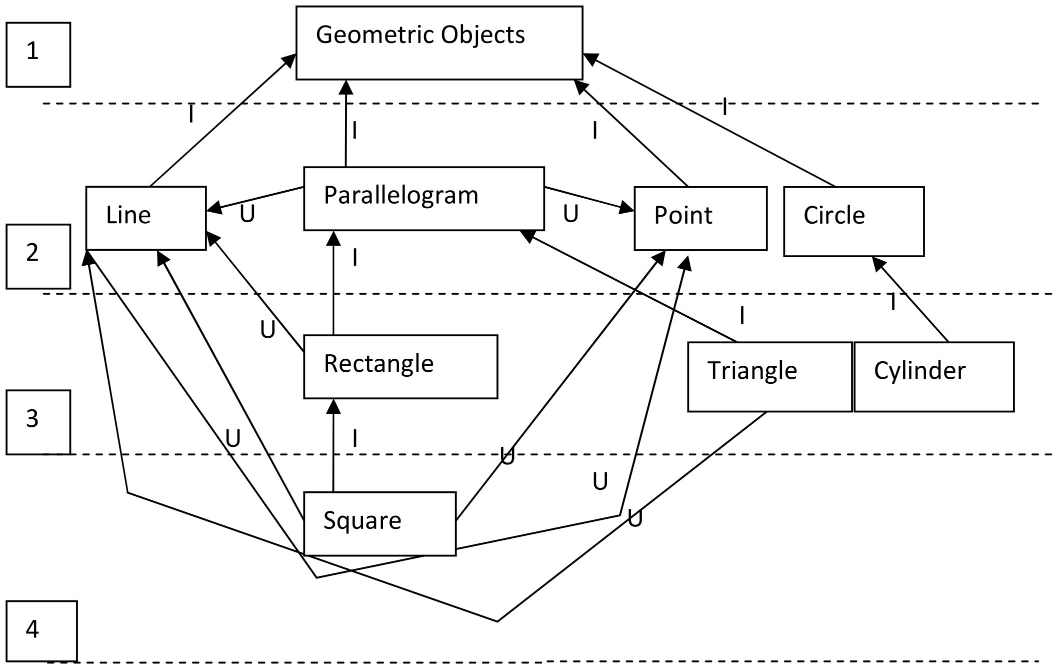

Figure.3. Class Dependency Model [GEO]

The CDM model uses the class relationships such as inheritance as well as the use relationship dependencies (a method in a class using a method from another class) to create a test order. The test order created mostly tests the base classes first, but ordering is based on both inheritance and the use relationship dependencies thus both relationships together decide in which order the base classes or the inherited classes should be tested. This model distributes the classes into various levels in a hierarchy based on the inheritance relationship, further divides them using the interaction between methods of these classes. For example in figure 3, *Line*, *Parallelogram*, *Point* and *Circle* classes are at the same inheritance level. In this case the use relationship dependencies among classes will help in deciding which of these base classes should be tested first. The use relationship is shown in Figure 3 using the "U" abbreviation in the diagram. The "U" in the diagram represents a relationship between two methods of the connected classes. The specific methods of the class are not shown in the diagram here but are considered in calculating the test order using this method dependency between classes. Figure 2b shows a partial diagram that depicts the dependencies between two methods of different classes.

The TDG model can create different test orders based on how the graph edges are analyzed for method-to-class relationships, method-to-method and class-to-class relationships, separating method relationships from class relationships. As mentioned earlier, both ORD and TDG do not provide detailed information about the relationships between classes and methods and thus are not considered good strategies for creating test orders. In this paper, we use CDM model to create a class test order and analyze whether it is beneficial for finding OO faults.

The strategies defined to create a test order using the above mentioned models focus on minimizing the stubs. Badri compares the CDM and ORD strategies for minimizing stubs [1]. In this paper, we are not contradicting this result but analyzing if the test order created by these strategies will work for finding the faults specific to OO programs.

A set of OO faults can be generated using the OO mutation operators. A number of mutation operators have been defined for OO programs [5, 6, 7 and 8]. Derezinska mentions the mutation operators that are specific to OO programs, thus we use these mutation operators to induce faults in the OO programs. These faults create a standard set of OO faults that can be induced in OO programs. The mutation operators used in this paper are categorized based on OO properties: Method Overloading, Polymorphism and Inheritance. The 4 operators used for Inheritance are: IHI (Hiding variable insertion), IHD (Hiding variable deletion), IOD (Overriding method deletion) and IPC (Explicit call of a parent's constructor deletion). The method overloading operators used are OAO (Argument order change), OAN (Argument number change), OMR (Overloading method contents change), OMD (Overloading method deletion) and the polymorphism operators used are PNC (*new* method call with child class type), PMD (Member variable declaration with parent class type) and PPD (Parameter variable declaration with child class type).

The next section describes the related work followed by section III that lays out the experiment that was setup to analyze the test order for the sample program. Section IV shows the results from the experiment and provides a discussion. Finally, section V discusses the conclusions of this paper.

II. RELATED WORK

There are a number of diagrams that help in defining the test order for programs. Badri et al discusses various class integration strategies and suggests a new test strategy based on a new class dependency model that takes classes as well as method interactions into account for defining a test order [1]. It also discusses the earlier diagrams that define test order– TDG and ORD. Primarily, it shows how an ORD diagram is converted to a CDM diagram and further how a detailed CDM diagram is used to create a test order. Bansal et al focuses on the test order generation strategies specifically for C++ programs [2]. Jeron et al discusses one of the earlier diagrams, that is the TDG and how UML was used to create this model which was further used to create a test order for the program. It discusses the various types of interactions between classes and methods [10]. Later, Wang et al proposed an effective approach for defining a test order specific to Java programs. It extends the TDG model by representing inter-class dependencies as well as inter-class coupling information [16]. Milanova et al describes how to construct precise ORD diagrams. It actually extended the existing ORD model that shows more details about interclass dependence [3]. Abdurazik et al discusses the issues with creating class integration and test order and proposes a new algorithm in the paper that computes the test order [15]. Jiang et al implements a tool for generating test order for object-oriented systems [17]. The main idea behind all these diagrams and algorithms is to model a strategy that can be used to define an effective test order for a program.

III. EXPERIMENT

Using the class dependency model (CDM), a test order is created for the sample program. The first step is to create the class dependency model using the non-mutated sample program, which is created using the object relation diagram. A partial ORD for the sample program is shown in figure 1. The

ORD model is then converted to the CDM diagram. The CDM for the sample program GEO is shown in figure 3. GEO constructs geometrical objects such as *Circle*, *Square*, *Rectangle*, etc and calculates the area and perimeter of these objects. It has 9 classes and has approx 1100 LOC. It is a small program but does take into account the various aspects of OO programming such as inheritance, polymorphism, etc [14].

The sample program is then analyzed for object-oriented faults. The OO mutation operators are categorized based on the OO properties. We use operators from 3 categories to seed faults in the sample program: Inheritance, Method overloading and Polymorphism [5, 6, 7, and 8]. Each class has multiple instances of each type of fault and for this experiment each instance of the fault is counted [14]. Figures 5, 6 and 7 shows the total instances of faults induced in each class in each category. It is visible from these charts that most faults are concentrated in the inherited classes.

A test order is then created based on the CDM model [1]. Each class is assigned major level numbers based on the inheritance relationship. The base classes are assigned major level number 1 and the number increases for each class at a lower level in the inheritance tree. Figure 3 shows the major level number assigned to each class at each inheritance level. The classes in each major level number are further prioritized based on the minor level numbers; the classes with lower minor level numbers are tested first. The minor level numbers are assigned to each class using the number of methods in a class *using* methods from a different class. The minimum number assigned is 1 if no methods are used from other classes. For example, the *Rectangle* class uses a method from class *Line* and thus a minor level number of 1 is assigned to class *Rectangle* as shown in figure 3 and 4. The pairs in figure 4 show the major level numbers and minor level numbers respectively, this pair decides the priority of testing the classes. The classes with least minor level number are tested in the beginning. In case of classes with same minor level numbers, the test order is defined based on the major level number; class with the lower major level number is tested earlier [1]. Figure 4 shows the test order created for sample program GEO. The test order is analyzed combined with the faults induced in the OO program. We analyze where the faults are present in the test order, i.e. which level of the test order contains the most faults.

1) GeometricObjects (1, 1)
2) Point (2, 1)
3) Circle (2, 1)
4) Rectangle (3, 1)
5) Cylinder (3, 1)
6) Line (2, 2)
7) Parallelogram (2, 3)
8) Triangle (3, 3)
9) Square (4, 3)

Figure.4. Test order for GEO

Figures 5, 6 and 7 show the total instances of the faults induced at each level in the test order. The graphs represent faults in the method overloading, inheritance and polymorphism category. Figure 5 shows the total instances of the faults in the category of method overloading for the test order. It shows that method overloading faults are more concentrated towards major level number 3 which represent classes lower in the inheritance tree. Figure 6 shows the total instances of the inheritance faults in GEO at each level in the test order. It shows that most faults are again concentrated in level 3 and some of these instances are near level 1. Figure 7 shows the total instances of faults based on the polymorphism property of the OO programs at each level. The next step in the experiment was to create test cases and verify how many of the induced faults at each level were killed. The test cases used in this experiment were created using the statement coverage criterion which is a widely accepted criterion for testing [14].

IV. Result & Analysis

The analysis was performed based on how many faults were killed at each level in the test order. Figure 8, 9 and 10 shows the test order vs. the faults killed at each level for method overloading, inheritance and polymorphism faults respectively. Figure 8 shows that the total instances of faults killed are grouped around level 3. Level 3 classes consists of 2 leaf nodes of the inheritance trees in GEO and 1 class that is included in the max length inheritance tree in GEO. The instances of faults killed are also closely scattered in level 3. Thus, when CDM test order is used to test the method overloading faults in the OO programs, the faults are killed later in the testing phase. Therefore, a different approach is required to create a test order.

Figure 9 shows that faults not killed are clustered in level 1 and level 3 and same levels have some of the faults killed. This result shows that a majority of inheritance faults will be killed later in the test order when CDM is used. Thus, a different strategy should be used to create a test order for this type of fault.

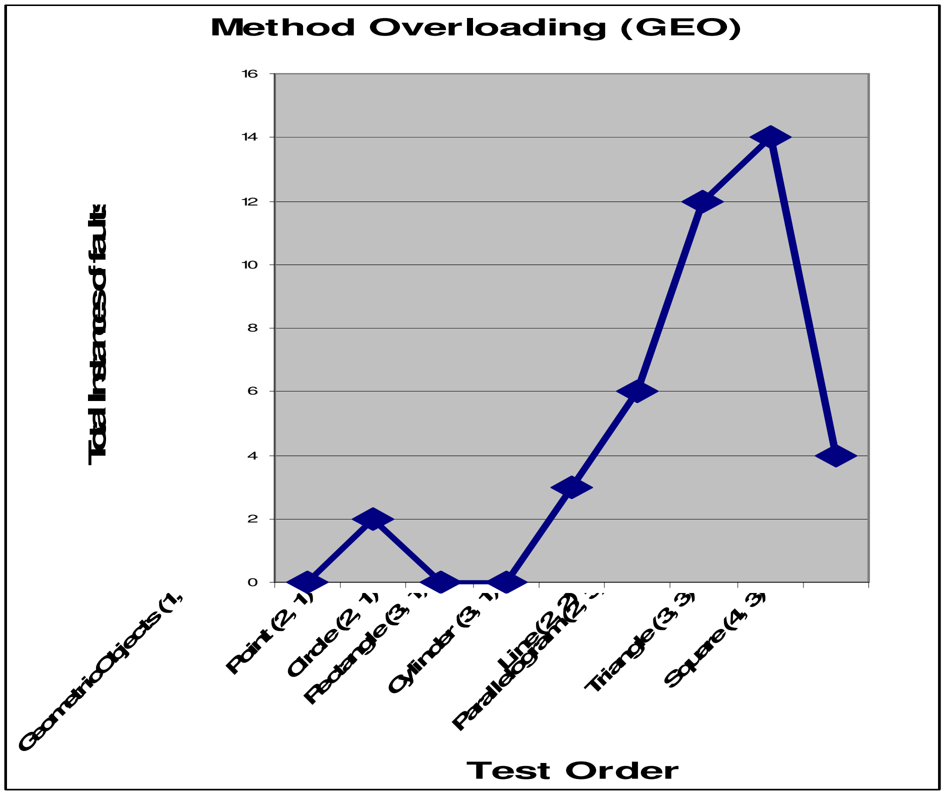

Figure.5. Test Order vs. faults for Method Overloading

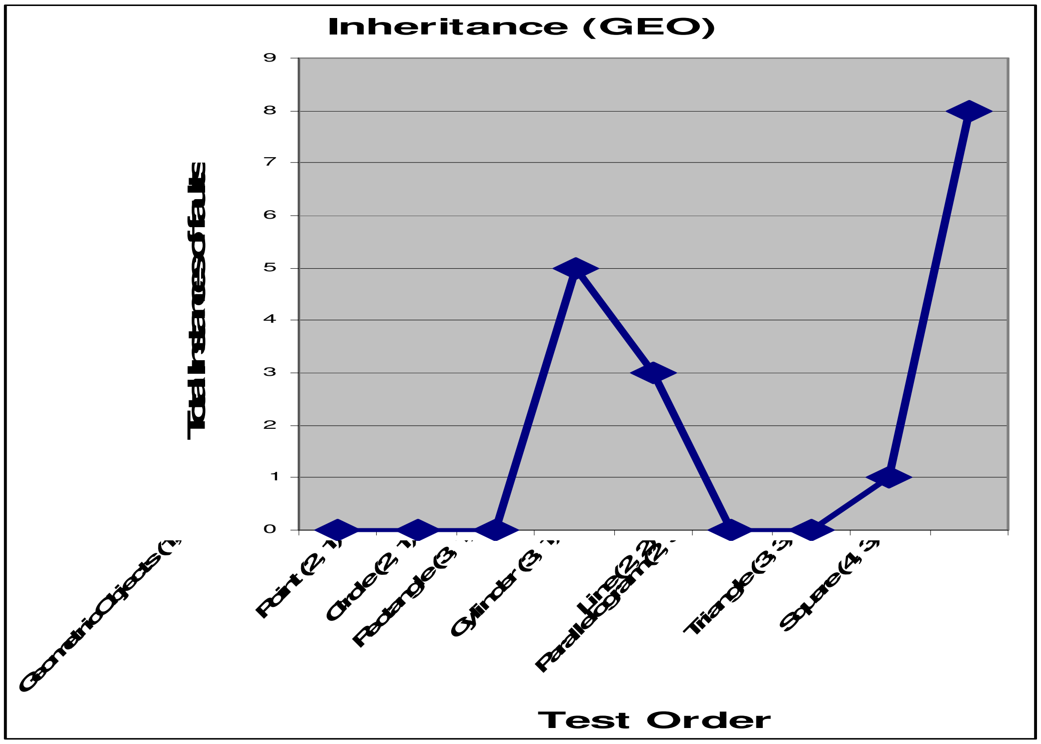

Figure.6. Test Order vs. faults for Inheritance

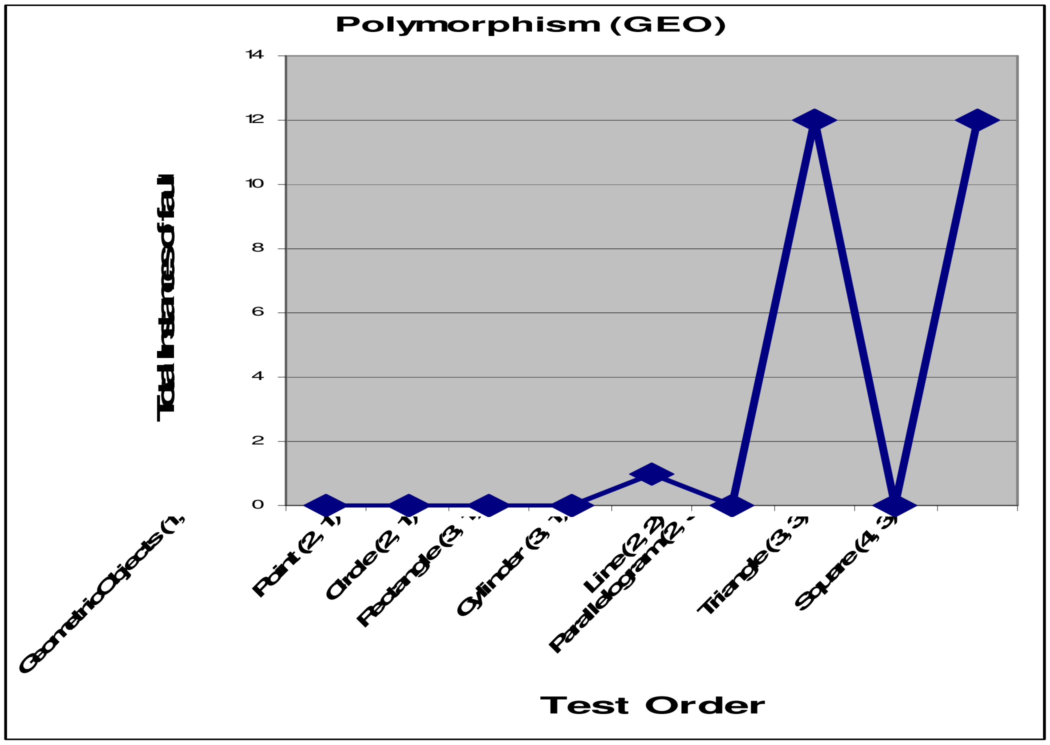

Figure.7. Test Order vs. faults for Polymorphism

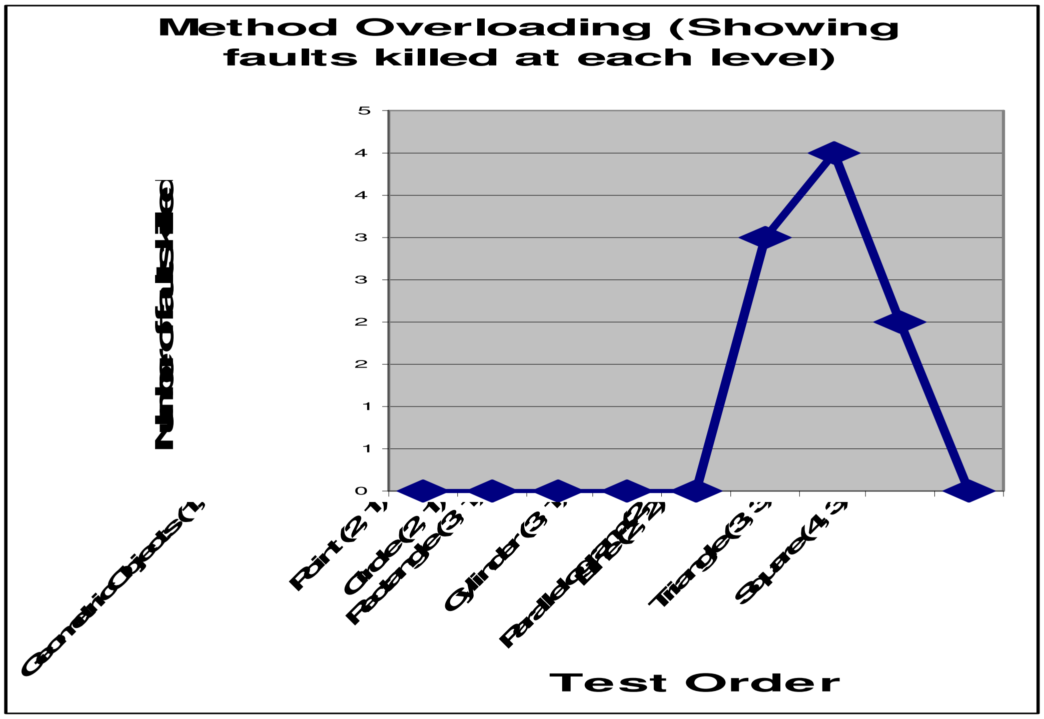

Figure.8. Test Order vs. faults killed for Method Overloading

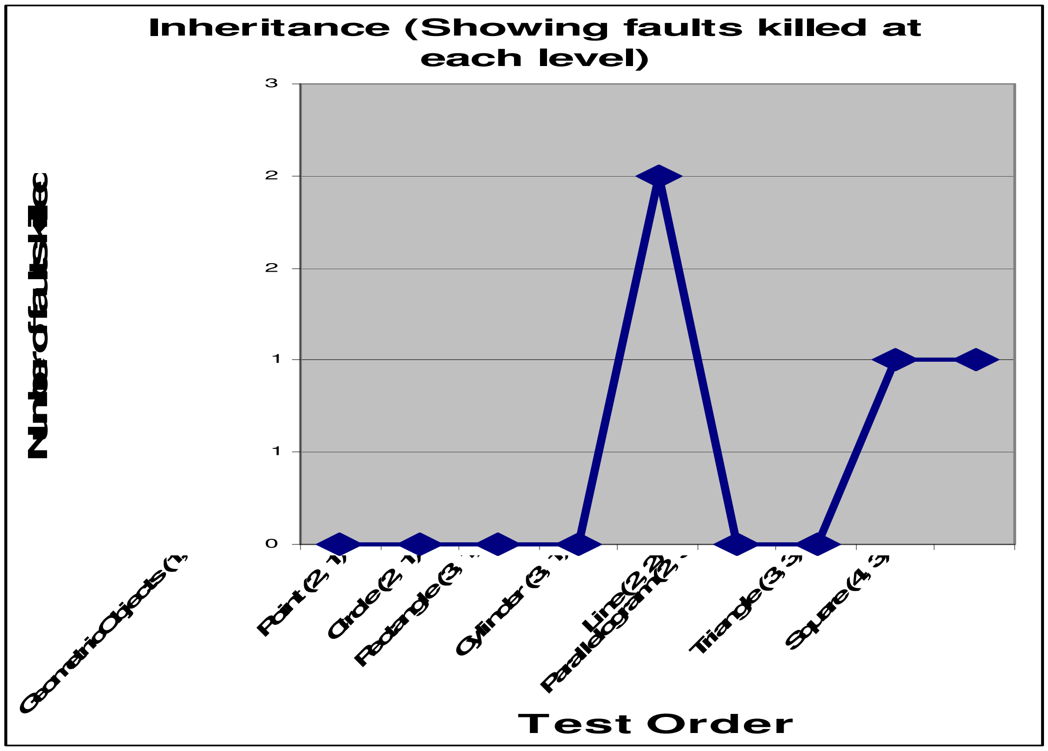

Figure.9. Test Order vs. faults killed faults for Inheritance

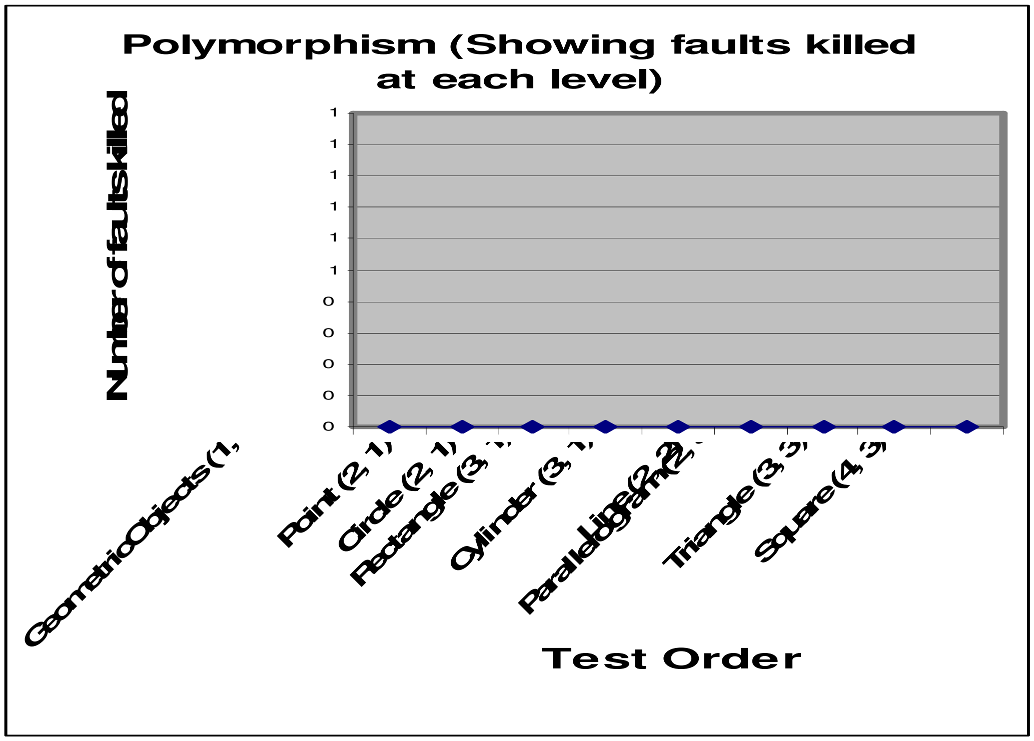

Figure.10. Test Order vs. faults killed for Polymorphism

Figure 10 shows that none of the polymorphism faults were killed at any level but the total number of instances for this category is concentrated in level 3 (shown in figure 7) of the test order. This shows that most of these faults are clustered in level 3 of the test order that contains 2 leaf nodes. Thus, a test order needs to be defined where leaf nodes have priority over parent nodes in the testing phase.

The method overloading faults are gathered along the various inheritance trees in the program and the leaf nodes of the inheritance trees. The polymorphism faults are concentrated in the leaf nodes of the inheritance trees and in the class that is included in the max length inheritance tree of the program. The inheritance faults are clustered in the leaf nodes and the parent of the leaf nodes in the inheritance trees of the OO program. Thus, we can see that for these types of OO faults the leaf nodes need to be tested first in order to kill more of these faults early in the testing phase. From the graphs in figure 8, 9 and 10, it shows that different types of faults are clustered at different levels in test order. This is the reason we cannot customize one testing plan for different types of object-oriented faults.

V. CONCLUSION

Most of the method overloading faults are located in the inherited classes and thus if killed earlier would reduce testing efforts when testing the base class before the inherited class. When testing a base class earlier than the inherited class, if there is a fault seeded in the inherited class it may take more time to analyze and find the fault as the class under test would be the base class. Therefore, once the faults are removed from the inherited class the faults to be analyzed would only be present in the base class.

For the inheritance faults, the test order needs to be defined along the various inheritance trees in the object-oriented program. This type of test order will help in testing the inheritance property along the different trees in the program. In the sample program GEO more faults are located along the inheritance tree, e.g. Parallelogram -> Rectangle -> Square. The graphs show that these types of faults are largely seeded in the inherited classes. Thus we need to test the classes along the inheritance tree taking into account that inherited classes need to be tested earlier than the base classes.

The polymorphism faults are one of the difficult faults to be killed. The results show that more of these faults are also seeded in the inherited classes. Also, the combination of methods and classes play an important role in killing these types of faults. Thus we need to define a test order which takes into account the combinations of various methods combined with classes and prioritizes them in an order where inherited classes are tested first.

The method overloading results show that leaf nodes of the inheritance tree need to be tested first (e.g. Triangle and Square). The Inheritance faults show that the inherited classes along the inheritance tree should be tested before the base classes. Finally, the polymorphism faults show that they are concentrated in the inherited classes as well and are one of the most difficult faults to be killed. Thus, in most cases we need to test the inherited classes before the base classes or use a combination of both. The results show that the test order created using CDM does not perform well with object-oriented faults, as more faults are killed later in the test order. We need to define a test order for each type of object-oriented property as each property causes the faults to behave differently and thus need to be tested differently.

REFERENCES

[1] Badri L, Badri M, Ble V.S, "A method level based approach for OO integration testing: an experimental study", *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks SNPD/SAWN.*, pp. 102- 109, 23-25 May 2005
[2] Bansal P, Sabharwal S, Sidhu P, "An investigation of strategies for finding test order during Integration testing of object Oriented applications,", *Proceeding of International Conference on Methods and Models in Computer Science, ICM2C*, pp.1-8, 14-15 December 2009
[3] Milanova A, Rountev A, Ryder,, B.G, "Constructing precise object relation diagrams,", *Proceedings of International Conference on Software Maintenance*, pp. 586- 595, 2002
[4] Briand L.C, Labiche Y, Yihong Wang, "An investigation of graph-based class integration test order strategies," *Software Engineering, IEEE Transactions on* , vol.29, no.7, pp. 594- 607, July 2003
[5] Ma Y. S, Harrold M. J, Kwon Y. R, "Evaluation of Mutation Testing for Object-Oriented Programs," in *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*, Shanghai, China, pp. 869–872, 20-28 May 2006.
[6] Derezinska A, "Advanced Mutation Operators Applicable in C# Programs,"
Warsaw University of Technology, Warszawa, Poland, Technique Report, 2005.
[7] Kim S, Clark J, McDermid J. "Class mutation: Mutation testing for object-oriented programs", *Proceedings of Net Objectdays Conference on Object-Oriented Software Systems*, October 2000.
[8] Yu-Seung Ma, Yong-Rae Kwon, Offutt, J, "Inter-class mutation operators for Java", *Proceedings of 13th International Symposium on Software Reliability Engineering, ISSRE,* pp. 352- 363, 2002
[9] Le Traon, Y, Jeron, T, Jezequel J. M, Morel P, "Efficient object-oriented integration and regression testing,", *IEEE Transactions on Reliability* , vol.49, no.1, pp.12-25, March 2000
[10] Jeron T, Jezequel J. M, Le Traon Y, Morel P, "Efficient strategies for integration and regression testing of OO systems," *Software Reliability Engineering,* pp.260-269, 1999
[11] Labiche Y, Thevenod-Fosse P, Waeselynck H, Durand M. H, "Testing levels for object-oriented software", *Proceedings of the 2000 International Conference on Software Engineering,* pp.136-145, 2000
[12] Hanh V. Le, Akif K, Traon Y. Le, Jezequel J.M, "Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies," *Proceedings 15th European Conference Object-Oriented Programming ECOOP*, pp. 381-401, June 2001.
[13] Briand L.C, Labiche Y, Yihong Wang, "Revisiting strategies for ordering class integration testing in the presence of dependency cycles," *Software Reliability Engineering, 2001. ISSRE 2001*, pp. 287- 296, November 2001
[14] Gupta P, Gustafson, D. A, "Object-oriented testing beyond statement coverage", *In the proceedings of ISCA 23rd international conference on CAINE 2010*, pp 150-155.
[15] Abdurazik A, Offutt J, "Coupling-based class integration and test order." *In Proceedings of the 2006 international workshop on Automation of software test (AST '06).* pp 50-56.
[16] Wang Z, Li B, Wang L, Li Q, "An Effective Approach for Automatic Generation of Class Integration Test Order," *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual* , pp.680-681, 18-22 July 2011
[17] Jiang S, Zhang, Y, Li H, Wang, Q, "An approach for inter-class integration test order determination based on coupling measures", *Chinese Journal of Computers*, pp 1062-1074. June 2011

AUTHORS

Pranshu Gupta is currently a doctoral student at Kansas State University, KS, USA in the department of Computing and Information Sciences. She has also completed a Masters degree at James Madison University, VA in Secure Software Engineering and a Bachelors degree in Electrical Engineering. Her research interests include Software Design, Software Testing and Data Warehousing.

Gustafson is a Professor at Kansas State University, KS, USA in the department of Computing and Information Sciences. He earned his PhD in Computer Science at the University of Wisconsin - Madison. His research areas include Software Testing, Software Measures, Robotics, and Computer Vision.