Prediction of Software Defects Using Object Oriented Programming

[1]Dr Mukesh Singla, [2]Parul

[1]Professor & Dean, [2]Research Scholar
[1]Faculty of Engineering, Baba Mast Nath University
Department of Computer Science and Engineering, Rohtak, India

***Abstract*: Development without any defect is unsubstantial. Timely detection of software defects favors the proper resource utilization saving time, effort and money. With the increasing size and complexity of software, demand for accurate and efficient prediction models is increasing. To remain competitive in the dynamic world of software development, organizations must optimize the usage of their limited resources to deliver quality products on time and within budget. This requires prevention of fault introduction and quick discovery and repair of residual faults. In this paper a new approach for predicting and classification of faults in object-oriented software systems is introduced.**

***Index Terms*: Complexity of software, quality products, residual fault etc.**

I. INTRODUCTION

Software systems are complex creations. They perform many functions; they are built to achieve many different, and often conflicting, objectives. They comprise many components; many of their components are custom made and complex themselves. Many participants from different disciplines take part in the development of these components. The development process and the software life cycle often spans many years. Finally, complex systems are difficult to understand completely by any single person. Many systems are so hard to understand, even during their development phase, that they are never finished: these are called vaporware.
Engineering is a problem-solving activity. Engineers search for an appropriate solution, often by trial and error, evaluating alternatives empirically, with limited resources and incomplete knowledge. In its simplest form, the engineering method includes five steps:
1. Formulate the problem.
2. Analyze the problem.
3. Search for solutions.
4. Decide on the appropriate solution.
5. Specify the solution.

II. OBJECT DESIGN:

During object design, developers define solution domain objects to bridge the gap between the analysis model and the hardware/software platform defined during system design. This includes precisely describing object and subsystem interfaces, selecting off-the-shelf components, restructuring the object model to attain design goals such as extensibility or understandability, and optimizing the object model for performance. The result of the object design activity is a detailed object model annotated with constraints and precise descriptions for each element. Proposals of measures and metrics for cohesion abound in the literature as software cohesion metrics proved to be useful in different tasks [1], including the assessment of design quality [2], [3], productivity, design, and reuse effort, prediction of software quality, fault prediction , modularization of software, and identification of reusable of components [4].

Testing is the process of analyzing a system or system component to detect the differences between specified (required) and observed (existing) behavior. Unfortunately, it is impossible to completely test a nontrivial system. First, testing is not decidable. Second, testing must be performed under time and budget constraints. As a result, systems are often deployed without being completely tested, leading to faults discovered by end users. Most activities of the development process are constructive: during analysis, design, and implementation, objects and relationships are identified, refined, and mapped onto a computer environment. Testing requires a different thinking, in that developers try to detect faults in the system, that is, differences between the reality of the system and the requirements. Many developers find this difficult to do. One reason is the way we use the word “success” during testing. Many project managers call a test case “successful” if it does not find a fault; that is, they use the second definition of testing during development. However, because “successful” denotes an achievement, and “unsuccessful” means something undesirable, these words should not be used in this fashion during testing. One of the goals of the OO analysis and design is to create a system where classes have high cohesion and there is low coupling among them.[5]

There are many techniques for increasing the reliability of a software system:
• **Fault avoidance** techniques try to detect faults statically, that is, without relying on the execution of any of the system models, in particular the code model. Fault avoidance tries to prevent the insertion of faults into the system before it is released. Fault avoidance includes development methodologies, configuration management, and verification.

• **Fault detection** techniques, such as debugging and testing, are uncontrolled and controlled experiments, respectively, used during the development process to identify erroneous states and find the underlying faults before releasing the system. Fault detection techniques assist in finding faults in systems, but do not try to recover from the failures caused by them. In general, fault detection techniques are applied during development, but in some cases they are also used after the release of the system. The blackboxes in an airplane to log the last few minutes of a flight is an example of a fault detection technique.

• **Fault tolerance** techniques assume that a system can be released with faults and that system failures can be dealt with by recovering from them at runtime. For example, modular redundant systems assign more than one component with the same task, then compare the results from the redundant components. The space shuttle has five onboard computers running two different pieces of software to accomplish the same task

An inspection is similar to a walkthrough, but the presentation of the component is formal. In fact, in a code inspection, the developer is not allowed to present the artifacts (models, code, and documentation). This is done by the review team, which is responsible for checking the interface and code of the component against the requirements. It also checks the algorithms for efficiency with respect to the nonfunctional requirements. Finally, it checks comments about the code and compares them with the code itself to find inaccurate and incomplete comments. The developer is only present in case the review needs clarifications about the definition and use of data structures or algorithms. Code reviews have proven to be effective at detecting faults. In some experiments, up to 85 percent of all identified faults were found in code reviews [11], [12].

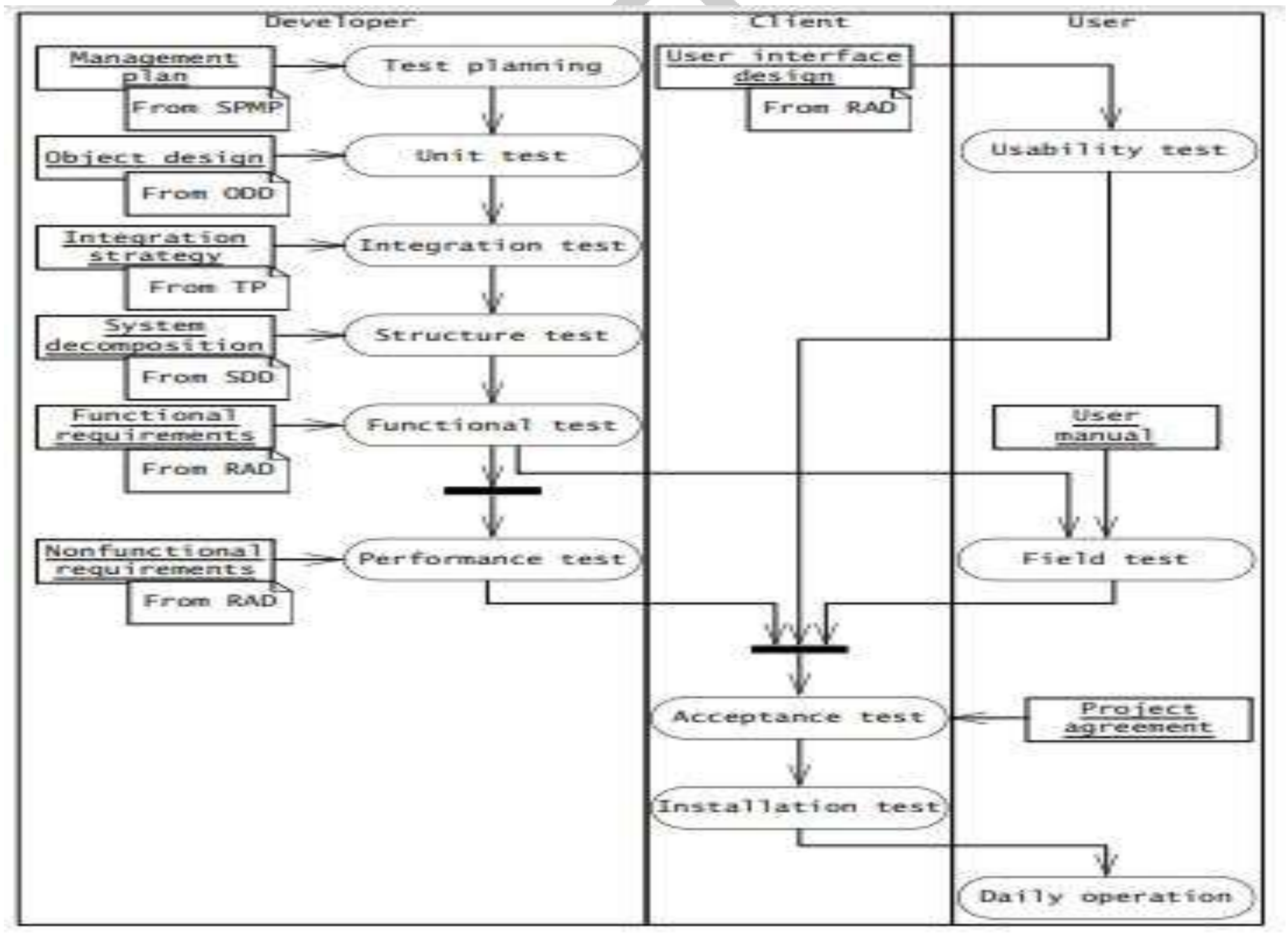

Figure: Testing process and related aspects

III RESEARCH METHODOLOGY

Object-oriented design has become a dominant method in software industry and many design metrics of object oriented programs have been proposed for quality prediction, but there is no well-accepted statement on how significant those metrics are. The object oriented metrics will be adopted to identify a limited set of measureable attributes that have a significant impact on prediction of Faults and quality attributes. The techniques involved will be statistical analysis. The statistical techniques will be used to reveal the relationship between metrics and dependent variables.

IV THE PROPOSED PLAN OF WORK

The starting of the work would be devoted on the introduction of various existing object-oriented design metrics. The second step would focus on study of software defects using complexity metrics in object-oriented design. The third step would focus on study of faults in object-oriented design. The fourth step would focus on impact faults on the quality of object-oriented software. Lastly the summary and conclusions and scope for further research would be discussed.

V TEST DRIVERS AND STUBS

Executing test cases on single components or combinations of components requires the tested component to be isolated from the rest of the system. Test drivers and test stubs are used to substitute for missing parts of the system. A test driver simulates the part of the system that calls the component under test. A test driver passes the test inputs identified in the test case analysis to the component and displays the results. A test stub simulates a component that is called by the tested component. The test stub must provide the same API as the method of the simulated component and must return a value compliant with the return result type of the method's type signature. Note that the interface of all components must be base lined. If the interface of a component changes, the corresponding test drivers and stubs must change as well. The implementation of test stubs is a nontrivial task. It is not sufficient to write a test stub that simply prints a message stating that the test stub was called. In most situations, when component a calls component b, a is expecting b to perform some work, which is then returned as a set of result parameters. if the test stub does not simulate this behavior, a will fail, not because of a fault in a, but because the test stub does not simulate b correctly. Even providing a return value is not always sufficient. For example, if a test stub always returns the same value, it might not return the value expected by the calling component in a particular scenario. This can produce confusing results and even lead to the failure of the calling component, even though it is correctly implemented. Often, there is a trade-off between implementing accurate test stubs and substituting the test stubs by the actual component. for many components, drivers and stubs are often written after the component is completed, and for components that are behind schedule, stubs are often not written at all.

VI COMPONENT INSPECTION

Inspections find faults in a component by reviewing its source code in a formal meeting. Inspections can be conducted before or after the unit test. The first structured inspection process was Michael Fagan's inspection method. The inspection is conducted by a team of developers, including the author of the component, a moderator who facilitates the process, and one or more reviewers who find faults in the component.

Fagan's inspection method consists of five steps:

- **Overview** The author of the component briefly presents the purpose and scope of the component and the goals of the inspection.
- **Preparation** The reviewers become familiar with the implementation of the component.
- **Inspection meeting** A reader paraphrases the source code of the component, and the inspection team raises issues with the component. A moderator keeps the meeting on track.
- **Rework** The author revises the component.
- **Follow-up** The moderator checks the quality of the rework and may determine the component that needs to be reinspected.

VII CK METRICS

The Chidamber and Kemerer have proposed six class-based design metrics for object-oriented systems [6][7].

- **Coupling Between Objects (CBO).** The CBO metric counts the number of other classes to which a class is coupled. It counts the number of reference types that are used in attribute declarations, formal parameters, return types, throws declarations, local variables, and types from which attribute and method selections are made. Primitive types, types from the java.lang package, and supertypes are not counted. High values of CBO metrics mean that the class is highly coupled. The developers and testers perceive that the maintainability and testability of highly coupled classes is difficult. which makes the process of maintaining and uncovering faults prerelease and postrelease difficult as well. The viewpoint are: If small values of CBO then improve modularity and promote encapsulation, indicates independence in the class making easier its reuse, makes easier to maintain and to test a class.
- **Lack of Cohesion of Methods (LCOM).** The LCOM metric is the number of pairs of methods in the class using no attributes in common (referred to as P), minus the number of pairs of methods that do (referred to as Q). The LCOM is set to zero if this difference is negative. After considering each pair of methods: LCOM = (P>Q) ? (P-Q) : 0. The LCOM metric measures the coherence among local methods in a class. The class that does one thing (i.e., cohesive class) is easier to reuse and maintain than the class that does many different things (i.e., the class provides many different services). The viewpoints are: If a great value of LCOM then increases complexity does not promotes encapsulation and implies classes should probably be split into two or more subclasses and helps to identified low-quality design.
- **Weighted Methods Complexity (WMC).** The WMC metric is the sum of the complexity of all methods for a class. Normally, many metrics tools calculate the WMC metric as simply the number of methods in a class. This is equivalent to saying all functions have equal complexity. However, in this research, the tool we used (i.e., Borland Together) calculates the WMC metric by summing the McCabe cyclomatic complexity of all the methods in the class. Therefore, high values of the WMC metric mean high complexities as well. The viewpoints are: WMC is a predictor of how much time and effort is required to develop and to maintain the class, the larger Number of Method (NOM) the greater the impact on children. Classes with large NOM are likely to be more application specific, limiting the possibility of reuse and making the effort expended one shot investment.
- **Depth of Inheritance Hierarchy (DIT).** The DIT measures the length of the inheritance chain from the root of the inheritance tree to the measured class. The DIT metric is an indicator of the number of ancestors of a class. It may require developers and testers to understand all ancestors to comprehend all specializations of the class, which is necessary to maintain or uncover pre and postrelease faults. The viewpoints are: If the greater values of DIT then the greater the Number of Methods (NOM) it is likely to inherit, making more complex to predict its behavior, the greater the potential reuse of inherited methods. Small values of DIT in most of the system's classes may be an indicator that designers are forsaking reusability for simplicity of understanding.

- **Number of Child Classes (NOC).** The NOC metric counts the number of descendents of a class. The number of children represents the number of specializations and uses of a class. Therefore, understanding all children classes is important to understand the parent. The high number of children increases the burden on developers and testers in comprehending, maintaining, and uncovering pre and postrelease faults. The viewpoints are: If the greater is the NOC then the greater is the reuse, the greater is the probability of improper abstraction of the parent class, the greater the requirements of methods testing in that class. Small values of NOC, may be and indicator of lack of communication between different class designers.

VIII MOOD METRICS

Abreu et at. Defined MOOD (Metrics for Object Oriented Design) metrics[8,9,10]. MOOD refers to a basic structural mechanism of the object-oriented paradigm as encapsulation (MHF, AHF) inheritance (MIF, AIF), polymorphism (POF), and message passing (COF). We will discuss MOOD metrics in the context of encapsulation, inheritance, polymorphism, and coupling. These are discussed below:

a. Encapsulation

The Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) were proposed together as measure of encapsulation

- **Method Hiding Factor (MHF).** This metric is the ratio of hidden (private or protected) methods to total methods. As such, MHF is proposed as a measure of encapsulation. If the value of MHF is high (100%), it means all methods are private which indicates very little functionality. Thus it is not possible to reuse methods with high MHF. MHF with low (0%) value indicate all methods are public that means most of the methods are unprotected.
- **Attribute Hiding Factor (AHF).** This metric is the ratio of hidden (private or protected) attributes to total attributes. AHF is also proposed as a measure of encapsulation. If the value of AHF is high (100%), it means all attributes are private. AHF with low (0%) value indicates all attributes are public.

IX CONCLUSION

It is hence proven that early fault identification can decline the cost of test and related efforts. It increases reliability and enhance quality in the software system, Fault identification and prediction is totally depends upon fault proneness of the data and data set being used in software. There is one more important thing that is identifying the leading faults and removing them automatically removes the previous dependent faults. This kind of elimination requires testing of the complete software system.

REFERENCES

[1] D. Darcy and C. Kemerer, “OO Metrics in Practice,” IEEE Software,vol. 22, no. 6, pp. 17-19, Nov./Dec. 2005.
[2] J. Bansiya and C.G. Davis, “A Hierarchical Model for ObjectOriented Design Quality Assessment,” IEEE Trans. Software Eng., vol. 28, no. 1, pp. 4-17, Jan. 2002.
[3] L.C. Briand, J. Wu¨ st, J.W. Daly, and V.D. Porter, “Exploring the Relationship between Design Measures and Software Quality in Object-Oriented Systems,” J. System and Software, vol. 51, no. 3, pp. 245-273, May 2000.
[4] J.K. Lee, S.J. Jung, S.D. Kim, W.H. Jang, and D.H. Ham, “Component Identification Method with Coupling and Cohesion,” Proc. Eighth Asia-Pacific Software Eng. Conf., pp. 79-86, Dec. 2001.
[5] Prakasa Rao Dasari, Vasanthakumari G; International Journal of Modern Engineering Research (IJMER), ISSN 2249-6645, Vol I, Issue 1, pp 113-119.
[6] Shatnawi R., “ A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems”, IEEE Transactions on Software Engineering, Vol. 36, No.2, pp. 223-224 March/April 2010M.
[7] Camargo Cruz Ana Erika, “Chidamber & Kemrer Suite of Metrics”, Japan Advanced Institute of Science and Technology School of Information, May 2008.
[8] Abreu F.B. and R.Carapuca.“Object-Oriented Software Engineering: “Measuring and Controlling the Development Process”. Proceedings of the 4th International Conference on Software Quality, McLean,Virginia, USA,October ,1994.
[9] Abreu, B.F. and W.L. Melo (1996): “Evaluating the impact of Object-Oriented Design on Software Quality”, Proceedings of METRICS ’96, IEEE, 1996.pp. 90-99
[10] Abreu F.B.(1995):. ECOOP’95 Quantitive Methods Workshop“Design Metrics for Object-Oriented Software Systems”1995
[11] M. E. Fagan, “Design and code inspections to reduce errors in program development,” IBM System Journal, Vol. 15, No. 3, pp. 182–211, 1976.
[12] T. C. Jones, “Programmer quality and programmer productivity,” IBM Technical Report TR–02.764, 1977.