ORIENTAL JOURNAL OF
COMPUTER SCIENCE & TECHNOLOGY
An International Open Free Access, Peer Reviewed Research Journal
www.computerscijournal.org

ISSN: 0974-6471
December 2011,
Vol. 4, No. (2):
Pgs. 281-292

Design Architecture Class Diagram for a Comprehensive Testing Tool

SARITA SINGH BHADAURIA[1]*, ABHAY KOTHARI[2] and LALJI PRASAD[3]

[1]MITS /Department of Electronics,Gwalior (India).
[2]IIST/ Computer Engineering, Indore (India).
[3]TRUBA College of Engineering and Technology/ Computer Science and Engineering, Indore (India).
*Corresponding author: saritamits66@yahoo.co.in

(Received: October 04, 2011; Accepted: October 06, 2011)

ABSTRACT

Object-orientation involving class and object concepts and their properties play an important role in constructing any object-orientated system. In this research work, a comprehensive class diagram is provided that may help in designing a comprehensive software testing tool. A requirement specification for a comprehensive software testing tool is established that would involve studying the feature set offered by existing software testing tools, along with their limitations.The requirement set thus developed will be capable of overcoming the limitations of the limited feature sets of existing software tools and will also contribute to the design of a comprehensive architecture class diagram for a software testing tool that includes most of the features required for a software testing tool (most of the testing techniques came from procedural and object-oriented programming system development). In addition, because different user interfaces are provided by different tools, an effort has been made to use them in the present system that is being designed.

Key words: Software architecture, Class level architecture, Fault-based testing, and Scenario-based testing.

INTRODUCTION

Software testing is an important means of assessing software quality.

Program testing is a rapidly maturing area within software engineering that is receiving increasing attention from both computer science theoreticians and practitioners. Its general aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances[10].

Testing often consumes 40%-50% of development efforts, and it consumes more effort for systems that require higher levels of reliability. Testing is a significant portion of the software engineering process. With the development of fourth generation languages (4GL), which speeds up the implementation process, the proportion of time devoted to testing is decreasing. As the amount of maintenance and upgrade work for existing systems grows, a significant amount of testing will also be needed to verify systems after changes are made[24].

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system conceptual, specification and implementation. These perspectives become evident as the diagram is created and help solidify the design.

The remainder of the paper is organized as follows: Section 2 introduces object-oriented software engineering and presents the various stages of testing. Section 3 discusses the literature survey Section 4 presents various artifacts related to class testing. Section 5 presents the objectives of this research. Section 6 presents the preliminary class architecture, and section 7 presents the conclusion and future work.

Object Oriented Testing

An object is an entity composed of data and procedures. The procedures, referred to as methods, implement the operations on the object's data. Each object has a state, an identity, and a behavior. The definition of the type of object is a description of its capabilities. Object-oriented testing focuses on the states of the objects and their interactions. In an object-oriented testing system, classes play important roles, classes are the smallest testable units, and they provide an excellent structuring mechanism. They allow a system to be divided into well-defined units that can then be implemented separately. Second, classes support information-hiding. A class can export a purely procedural interface and the internal structure of the data may be hidden. This allows the structure to be changed without affecting users of the class, thus simplifying maintenance. Third, object-orientation encourages and supports software reuse. This may be achieved either through the simple reuse of a class in a library, or via inheritance, whereby a new class may be created as an extension of an existing one. The behavior of inherited methods can be changed because of methods that are called within methods must be tested per class.

Unlike conventional test case design, which is driven by an input-process-output view of software or the algorithmic detail of individual modules, object-oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class. Object-oriented software is developed incrementally with iterative and recursive cycles of planning, analysis, design, implementation and testing .Testing plays a special role here, because it is performed after each increment[19,20].

The Major Stages of Research and Development Trends in Object Oriented System Architecture and Testing (Literature survey)

Generally, we see three major stages in the research and development of testing techniques, each with a different trend. By trend, we mean how mainstream of research and development activities find the problems to solve and how they solve the problems. As described below, technology evolution involves testing technique technologies. The technique used for selecting test data has progressed from an ad hoc approach, through an implementation-based phase, and is now specification based. The literature survey includes the solution approaches of various research studies that dealt with problems related to testing methods and issues in the design of testing tools for various circumstances.

Literature Survey

[BBL97] A framework for probabilistic functional testing is proposed in this paper. The authors introduce the formulation of the testing activity, which guarantees a certain level of confidence into the correctness of the system under test. They also explain how one can generate appropriate distributions for data domains, including most common domains, such as intervals of integers, unions, Cartesian products, and inductively defined sets. A tool assisting test-case generation according to this theory is proposed. The method is illustrated on a small formal specification[5].

[Beizer90] This book gives a fairly comprehensive overview of software testing that emphasizes formal models for testing. The author provides a general overview of the testing process and the reasons and goals for testing. In the second chapter of this book, the author classifies the different types of bugs that could arise in program development. The notion of path testing, transaction flow graphs, data-flow testing, domain testing, and logic-based testing are introduced in detail. The author also introduces several attempts to quantify program complexity and a more abstract discussion involving paths, regular expressions and syntax testing. The implementation of software testing based on these strategies is also discussed[4].

[BG01] Testing becomes complicated with features, such as the absence of component source code, that are specific to component-based software. This paper proposes a technique combining both black-box and white-box strategies. A graphical representation of component software, called a component-based software flow graph (CBSFG), which visualizes information gathered from both specification and implementation, is described. It can then be used for test-case identification based on well-known structural techniques[7].

[BIMR97] In this paper the authors use formal architectural descriptions (CHAM) to model the behaviors of interest of the systems. A graph of all the possible behaviors of the system in terms of the interactions between its components is derived and further reduced. A suitable set of reduced graphs highlights the specific architectural properties of the system, and can be used for the generation of integration tests according to a coverage strategy, analogous to the control and data flow graphs in structural testing[3].

[GG75] This paper is the first published paper that attempted to provide a theoretical foundation for testing. The “fundamental theorem of testing” proposed by the authors characterizes the properties of a completely effective test selection strategy. The authors argue that a test selection strategy is completely effective if it is guaranteed to discover any error in a program. As an example, the effectiveness of branch and path testing in discovering errors is compared. The use of a decision table (a mixture of requirements and design-based functional testing) as an alternative method is also proposed[12].

[GH88] In this article, the evolution of software test engineering is traced by examining changes in the testing process model and the level of professionalism over the years. Two phase models, the demonstration and destruction models, and two life cycle models, the evolution and prevention models, are provided to characterize the growth of software testing with time. Based on the models, a prevention-oriented testing technology is introduced and analyzed in detail[11].

[HIM00] Unified Modeling Language (UML) is widely used for the design and implementation of distributed, component-based applications. In this paper, the issue of testing components by integrating test generation and test execution technology with commercial UML modeling tools such as Rational Rose is addressed. The authors present their approach to modeling components and interactions and describe how test cases are derived from these component models and then executed to verify their conformant behavior. The TnT environment of Siemens is used to evaluate the approach by examples[13].

[Howden76] The reliability of path testing provides an upper bound for the testing of a subset of a program’s paths, which is always the case in reality. This paper begins by showing the impossibility of constructing a test strategy that is guaranteed to discover all errors in a program. Three commonly occurring classes of errors, computations, domain, and sub case, are characterized. The reliability properties associated with these errors affect how path testing is defined[16].

[Howden80] The usual practice of functional testing is to identify functions that are implemented by a system or program from requirement specifications. In this paper, the necessity of design testing and requirement functions is discussed. The paper indicates how systematic design methods, such as structured design and the Jackson design, can be used to construct functional tests. Structured design can be used to identify the design functions that must be tested in the code, while the Jackson method can be used to identify the types of data that should be used to construct tests for those functions[17].

[Huang75] This paper introduces the basic notions of dynamic testing based on a detailed path analysis in which full knowledge of the contents of the source program being tested is used during the testing process. Instead of the common test criteria in which every statement in the program is executed at least once, the author suggested and demonstrated with an example that a better criterion is to require that every edge in the program diagram be exercised at least once. The process of manipulating a program by inserting probes along each segment in the program is suggested in this paper[14].

[JM94] Many models exist for estimating and predicting the reliability of software systems, most of which consider a software system as a black box and predict the reliability based on the failure data observed during testing. In this paper, a reliability model based on the software structure is proposed. The model uses the number of times a particular module is executed as the main input. A software system is modeled as a graph, and the reliability of a node is assumed to be a function of the number of times it gets executed during testing–the larger the number of times a node gets executed, the higher its reliability. The reliability of the software system is then computed through simulation by using the reliabilities of the individual nodes[18].

[Marciniak94] A book intended for software engineers, this book gives introductions, overviews, and technical outlines of the major areas in software engineering. A review of test generators is provided in which the major types of test case generators are given and their intended purpose and principles are discussed. A review of the testing process is given in which the entire process of testing is discussed from planning to execution to achieving to maintenance retesting. All of the common terms and ideas are discussed. A review of testing tools is provided in which the testing tool for each purpose is discussed and several state-of-the-art systems are described[23].

[Miller81] This article serves as one of the introductory sections of the book Tutorial: Software Testing and Validation Techniques. A cross section of program testing technology before and around the year 1980 is provided in this book, including the theoretical foundations of testing tools and techniques for static analysis and dynamic analysis effectiveness assessment management and planning and the research and development of software testing and validation. The article briefly summarizes each of the major sections and provides a general overview of the motivation forces, the philosophy and principles of testing, and the relationship between testing and software engineering[22].

[ROT89] This paper proposes one of the earliest approaches focusing on utilizing specifications in selecting test cases. In traditional specification-based functional testing, test cases are selected by hand based on a requirement specification, which means functional testing merely includes heuristic criteria. Structural testing has an advantage in that the applications can be automated and the satisfaction determined. The authors propose approaches to specification-based testing by extending a wide variety of implementation-based testing techniques to formal specification languages, and they demonstrate these approaches for the Anna and Larch specification languages[25].

[RR85] In this paper, a variety of software technologies are reviewed. The technology maturation process by which a piece of technology is created is described: first, an idea is formulated and preliminarily used; it is then developed and extended into a broader solution and finally enhanced to product-quality applications and marketed to the public. The time required for a piece of technology to mature is studied, and the actions that can accelerate the maturation process are addressed. This paper serves as a strong framework for technology maturation study[29].

[RW85] A family of test data selection criteria based on data flow analysis is defined in this paper. The authors contend that data flow criteria are superior to current path selection criteria in that when using the latter strategy, program errors can go undetected. The definition/use graph is introduced and compared with a program graph based on the same program. The interrelationships between these data flow criteria are also discussed[28].

[Shaw90] Software engineering is still in the process of becoming a true engineering discipline. This article studies the model for the evolution of an engineering discipline and applies it to software technology. Five basic steps are suggested for the software profession in creating a true engineering discipline: understanding the nature of expertise, recognizing different ways to obtain information, encouraging routine practice, expecting professional specializations, and improving the coupling between science and commercial practice. The significant shifts in software engineering research since the 1960s are also discussed in this article[31].

[WC80] Domain errors are in the subset of the program input domain and can be caused by incorrect predicates in branching statements or incorrect computations that affect variables in branching statements. In this paper, a set of constraints under which it is possible to reliably detect domain errors is introduced. The paper develops the idea of linearly bounded domains. The practical limitations of the approach are also discussed, the most severe of which is generating and then developing test points for all boundary segments of all domains of all program paths[33].

[Whit00] As a practical tutorial article, this paper answers questions from developers about how bugs escape testing. Undetected bugs come from executing untested code, differences in the order of executing, combinations of untested input values, and untested operating environments. A four-phase approach is described in answering the questions. By carefully modeling the software's environment, selecting test scenarios, running and evaluating test scenarios, and measuring testing progress, the author offers testers a structure for the problems they want to solve during each phase[32].

[Poston 2005]

Here we summarize their work

´ Integration of all the data across tools and repositories.

´ Integration of control across the tools.

´ Integration to provide a single graphical interface for the test tool set.

Limitation

It emphasizes only integration tools (usability and portability)[27].

[Rosenberg 2008] The approach to software metrics for object-oriented programs must be different from the standard metric sets. Some metrics, such as line of code and cyclomatic complexity, have become accepted as standard for traditional functional/procedural programs. However, for object-oriented scenarios, there are many proposed object-oriented metrics in the literature.

Limitation

This provides only a conceptual framework for measurement[26].

[Agrawal 2007] As per this paper the importance of software measurement is increasing which is leading to the development of new measurement techniques [2].

Limitation

a) In this research, object-oriented metrics does not provide any relationship between requirements and testing attributes.

b) In this research, object-oriented metrics cannot be evaluated for large data sets.

"Software quality is another focus of our research. Metrics fall into two categories: the productivity and the quality. Most of our object oriented metrics are quality related. We wish to achieve good maintainability, reusability, flexibility and portability in the architecture of the software testing tool under construction".

[Anderson 2005] They emphasize that the software industry has performed a significant amount of research on improving software quality using software tools and metrics that will improve the software quality and reduce the overall development time. Good-quality code will also be easier to write, understand, maintain and upgrade [1].

Limitation

a) In this research, object-oriented metrics does not provide any relationships between requirement testing attributes.
b) In this research, object-oriented metrics does not provide full-featured testing tools (only complexity and cohesion measures).
c) In this research, object-oriented metrics provides only a conceptual framework for measurement.

[Briand 1999] This paper shows that the relationships between most of the existing coupling and cohesion measures for object-oriented (OO) systems and the fault proneness of object-oriented system classes can be studied empirically [6].

Limitation

Only emphasizes cohesion and coupling metrics.

[Bitman 1997] This research defines a key problem in software development: changing software development complexity and the method to reduce complexity[8].

Limitation

It provides only a complexity measurement technique.

[Harrison 1998] Coupling is the degree of interdependence between two modules. In a good design coupling is kept to a minimum. Coupling should be low in a large and complex system. No coupling is highly desirable, but this is not possible in practice. The strengths and weaknesses of different types of coupling are discussed[15].

Limitation

Only cohesion and coupling metrics are emphasized.

[Chidambaram & Kemerer 1994]

The coupling between the object (CBO) metric of Chidambaram and Kemerer are evaluated for five object-oriented systems and compared with an alternative design metric called NAS that measures the number of associations between a class and its peers (Harrison R.S). The NAS metric is directly collectible from design documents, such as the object model[9].

Limitation

a) No relationship between requirements and testing attributes is provided.
b) A basic idea of the size and effort estimation is not provided.
c) Measuring the complexity of a class is subject to bias.

Artifacts of Class Testing

In this section, we refer to several of the attributes of object-oriented systems and discuss the different testing techniques suitable for object-oriented software systems. Attributes play an important role in making object-oriented software[24].

Encapsulation

Wrapping data and functions into a single unit is known as encapsulation. This restricts the visibility of object states and restricts the observability of intermediate test results. Fault discovery is more difficult in this case.

Inheritance

The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class, and the new one is called the derived class or the subclass. Inheritance results in invisible dependencies between super/sub-classes. Inheritance results in reduced code redundancy, which results in increased code dependencies. If the function is erroneous in the base class, it will also be inherited in the derived class. A subclass cannot be tested without its super-classes. Abstract classes cannot be tested at all.

Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means that one name represents multiple forms. Because of polymorphism, all possible bindings must be tested. All potential execution paths and potential errors must be tested. Testing begins by evaluating the OOA and OOD models. Object-oriented analysis models can be tested using the collected requirements and use cases. Object-oriented design can be tested by using the class and sequence diagrams. Structured walkthroughs and reviews should be conducted to ensure correctness, completeness and consistency.

Object–oriented programming is centered on concepts such as Object, Class, Message, Interfaces, Inheritance, and Polymorphism. Traditional testing techniques can be adopted in object-oriented environments by using the following techniques:

´ Function-based testing.
´ Class testing.
´ Integration testing.
´ Fault-based testing.
´ Scenario-based testing.

Function-based Testing

Like conventional (traditional) testing, function-based testing is based on product requirements and specifications.

Class Testing

Class testing is performed on the smallest testable unit in the encapsulated class. As part of a class hierarchy, each operation must be tested because its class hierarchy defines its context of use. New methods, inherited methods and redefined methods within the class must be tested. This testing is performed using the following approaches:

- Test each method (and constructor) within a class.
- Test the state behavior (attributes) of the class between methods.

Class testing is different from conventional testing in that conventional testing focuses on input-process-output, whereas class testing focuses on each method. In addition to testing methods within a class (either white box or black box). Test cases should be designed so that they are explicitly associated with the class and/or method to be tested. The purpose of the test should be clearly stated. Each test case should contain the following:

1. A list of messages and operations that will be exercised as a consequence of the test.
2. A list of exceptions that may occur as the object is tested.
3. A list of external conditions for setup (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
4. Supplementary information that will aid in understanding or implementing the test.

Some challenge in object-oriented class testing[21]
Encapsulation

Difficult to obtain a snapshot of a class without building extra methods that display the classes' state.

Inheritance and polymorphism

´ Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).
´ Other unaltered methods within the subclass may use the redefined method and need to be tested.

White box tests

Basis path, condition, data flow and loop tests can all apply to individual methods but do not test interactions between methods.

Class-level testing can be classified into the following parts
Random class testing

Identify methods applicable to a class. Define constraints on their use:

´ The class must always be initialized first. Identify a minimum test sequence.
´ Choose an operation sequence that defines the minimum life history of the class. Generate a variety of random (but valid) test sequences.
´ This exercises more complex class instance life histories.

Partitioned-based testing

This approach reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software for the following types of partitioned-based testing:

State-based partitioning

Tests are designed such that operations that cause state changes are tested separately from those that do not cause any changes in the state.

Attribute-based partitioning

For each class attribute, operations are classified according to those that use the attribute, modify the attribute and do not use or modify the attribute.

Category-based partitioning

Operations are categorized according to the function they perform:

i. Initialization.
ii. Computation
iii. Query
iv. Termination

Integration Testing

OO does not have a hierarchical control structure, and thus, conventional top-down and bottom-up integration tests have little meaning. Integration testing can be applied in three different incremental strategies:

´ Thread-based testing, which integrates classes required to respond to one input or event.

´ Use-based testing, which integrates classes required by one use case.

´ Cluster testing, which integrates classes required to demonstrate one collaboration.

Test cases should be designed so that they are explicitly associated with the class and/or

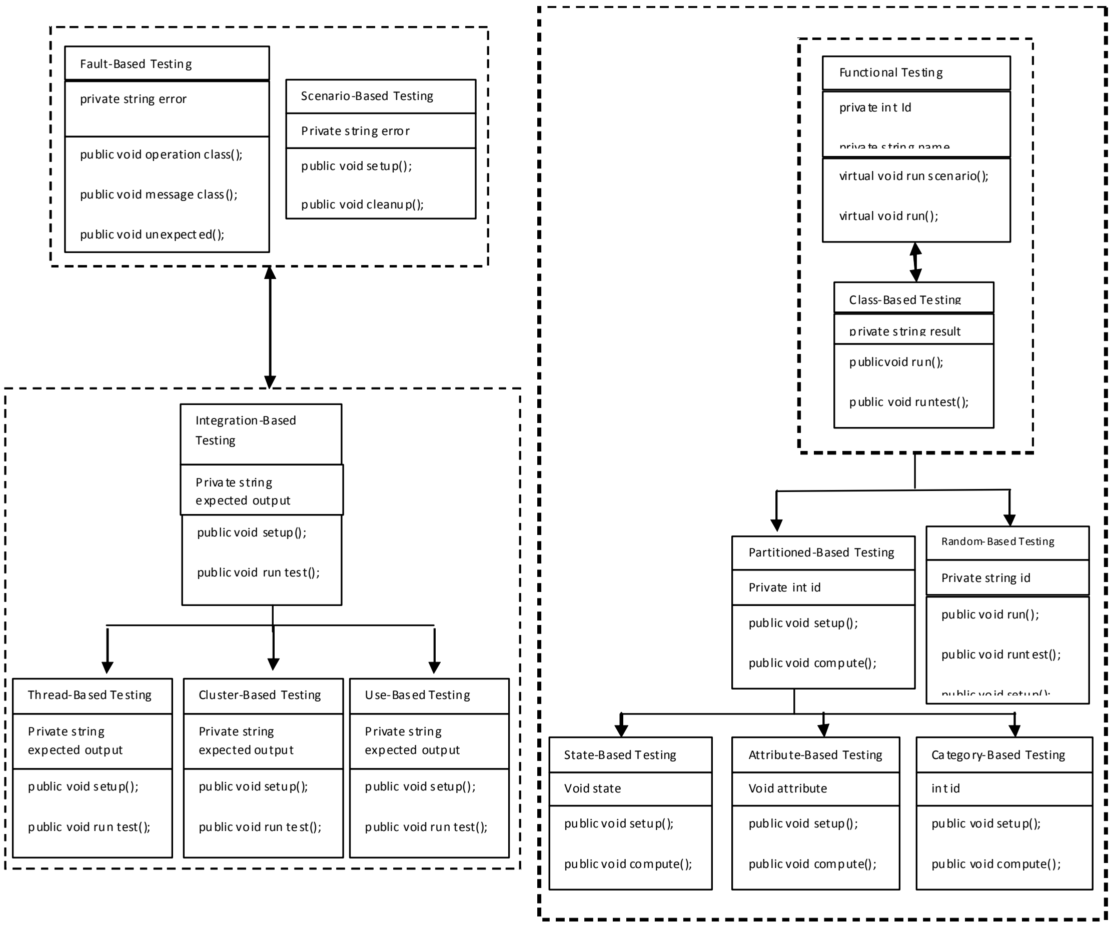

Fig. 1: Class Architecture for the Testing Tool

method to be tested. The purpose of the test should be clearly stated. Each test case should contain the following:

- A list of messages and operations that will be exercised as a consequence of the test.
- A list of exceptions that may occur as the object is tested.
- A list of external conditions for setup (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
- Supplementary information that will aid in understanding or implementing the test.

Fault–based Testing

Any product must conform to customer requirements. Hence, testing should begin with the analysis model itself to uncover errors. Fault–based testing is the method used to design tests that have a high probability of finding probable errors in the software [24]. Fault–based testing should begin with the analysis and design models. This type of testing can be based on specifications (e.g. user's manuals) or the code. It works best when based on both.

Scenario–based Testing

Scenario-based testing concentrates on what the customer does, not what the product does. It means capturing the tasks (use cases, if you will) that the customer has to perform and then using them and their variants as tests. Of course, this design work is best performed before the product is implemented. It is really an offshoot of a careful attempt at "requirements elicitation". These scenarios will also tend to flush out interaction bugs. They are more complex and more realistic than fault-based tests. They tend to exercise multiple subsystems in a single test, precisely because that is what users do. The tests will not find everything, but they will at least cover the higher-visibility interaction bugs[24].

Objective of Research

This research work consists of the following:

- Designing an object-oriented testing architecture template at the class diagram level.
- Using this architecture we represent different operations for each testing technique and associated different attributes using certain testing technique operations with other testing operations (from a set of operations it is capable of performing, it changes its attribute values, which may cause changes to the attribute values of other objects).

Preliminary Class Architecture

The outcome of the present work is shown in figure1, and the necessary discussion of the testing concepts involved is given here. In figure1, object-oriented testing is divided into three parts based on their functionality.

The first category consists of functional testing, class testing and its derived classes. This category is directly based on the requirements and specifications of software products, which involves the following:

1. Input the functional specification for function level testing of any testing tools.
2. Accordingly, functional specifications construct class-level testing.
3. Class level testing is dividing into two parts-partitioning class testing and random testing.

Partitioning-based testing and random testing are derived from class-level testing and uses some properties of class testing.

In the second category, integration-based testing is further divided into three parts-threads, cluster and use-based testing:

1. Thread-based testing integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually.
2. Use-based testing begins the construction of the system by testing those classes (called independent classes) that use very few (if any) server classes. After the independent classes are tested, the next layers of classes, called dependent classes, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.
3. Cluster testing is one step in the integration testing of OO software. Here, a cluster of

collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

The third part consists of fault-based testing and scenario-based testing

1. The objective of fault-based testing within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, the preliminary planning required to perform fault-based testing begins with the analysis model. The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.
2. Fault-based testing misses two main types of errors:

(1) Incorrect specifications.
(2) Interactions among subsystems.

When errors associated with incorrect specifications occur, the product does not do what the customer wants. Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests. Scenarios uncover interaction errors. However, to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test.

CONCLUSION

The maturation of testing techniques has been fruitful but not adequate. Pressure to produce higher-quality software at lower cost is increasing, and the existing techniques used in practice are not sufficient for this purpose. Empirical studies and fundamental research that addresses the challenging problems, development of methods and tools should be conducted so that we can significantly improve the way we test software. The successful use of these techniques in industrial software development will validate the results of the research and drive future research. The pervasive use of software and the increased cost of validating it will motivate the creation of partnerships between industry and researchers to develop new techniques and facilitate their transfer to practice. Development of efficient testing techniques and tools that will assist in the creation of high-quality software will become one of the most important research areas in the near future.

This research work, first establishes a total set of requirement specifications for a comprehensive software-testing tool. In an object-oriented environment, these requirements will address various testing methods and strategies for object-oriented development scenarios. This work will propose architectural design object-oriented paradigms that will satisfy the established requirements specifications. These designs can be further translated into practical industrial tools.

Future Work

In addition, this study will propose a class diagram to use, which will be relevant for obtaining measurements of the proposed architectures. These measurements will be used to draw inferences for understanding the behavior of the metrics in relation to the proposed architectures for improving the designs by optimizing their quality.

REFERENCES

1. Anderson John L. Jr., “How to Produce Better Quality Test Software”, IEEE Instrumentation & Measurement Magazine, (2005).
2. Agarwal K. K., Sinha Y., Kaur A. and Malhotra R.,” Exploring Relationships among coupling metrics in object oriented systems. *Journal of CSI* **37**(1): (2007).
3. Bertolino A., Inverardi ‘P., Muccini H. and Rosetti A., “An approach to integration testing based on architectural descriptions,”

Proceedings of the IEEE ICECCS- 97, pp. 77-84.

4. Beizer B., "Software Testing Techniques," Second Edition, Van Nostrand Reinhold Company Limited, ISBN 0-442-20672-0 (1990).
5. Bernet G., Bouaziz L. and LeGall P., "A Theory of Probabilistic Functional Testing," Proceedings of the 1997 International Conference on Software Engineering, pp. 216 –226 (1997).
6. Briand Lionel C. and Daly J., "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems", Fraunhfer IESE, 1999.
7. Beydeda S. and Gruhn V., "An integrated testing technique for component-based software," ACS/IEEE International Conference on Computer Systems and Applications, pp 328 – 334 (2001).
8. Bitman William R., " Balancing software composition & inheritance to improve reusability cost & error rate", Johns Hopkins APL Technical Digest, 18 (1997).
9. Chidamber S. and Kemerer C., "A metrics suite for object oriented design", IEEE Trans. Software Eng., **20**: pp. 476-493, 1994.
10. Edward Miller and William E. Howden, Tutorial: Software Testing & Validation Techniques. IEEE Computer Society Press, second edition, (1981).
11. Gelperin D. and Hetzel B., "The Growth of Software Testing", Communications of the ACM, **31**(6): pp. 687-695 (1988).
12. Goodenough J.B. and Gerhart L., "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, June pp. 156-173 (1975).
13. Hartmann J., Imoberdorf C. and Meisinger M., "UML-Based Integration Testing," Proceedings of the International Symposium on Software Testing and Analysis, ACM SIGSOFT Software Engineering Notes, (2000).
14. Huang J. C., "An Approach to Program Testing," ACM Computing Surveys, pp.113-128 (1975).
15. Harrison R., Counsell S. and Nithi R., "Coupling metrics for object oriented design", Software metrics, symposium, MD, USA, 19 (1998).
16. Howden W. E., "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Testing, 208-215 (1976).
17. Howden W. E., "Functional Testing and Design Abstractions", the *Journal of System and Software*, **1**: pp. 307-313 (1980).
18. Jalote P. and Muralidhara Y. R., "A coverage based model for software reliability estimation", Proceedings of First International Conference on Software Testing, Reliability and Quality Assurance, pp. 6 –10 (IEEE) (1994).
19. Jilles V. G., Object Oriented Testing Reports, software verification and validation, DAD404, IDE, University of karlskrona/Ronneby, 1998.
20. Kao G. M., Tang M. H. and Chen M. H., Investigating test effectiveness on object oriented software - a case study. In Proceedings of Twelfth Annual International Software Quality Week, (1999).
21. Larman James Gain and Blank George, of NJITplus Glenn Blank's elaborations and expansions. Notes from New Jersey Institute of Technology, (2009).
22. Miller E. F., "Introduction to Software Testing Technology", Tutorial: Software Testing & Validation Techniques, Second Edition, IEEE Catalog No. EHO 180-0, pp. 4-16.
23. Marciniak J. J., "Encyclopedia of software engineering", Volume 2, New York, NY: Wiley, pp.1327-1358 (1994).
24. Pressman Roger S., "Software Engineering – A Practitioner's Approach" McGraw Hill International Edition sixth (2004).
25. Richardson D., O'Malley O. and Title C., "Approaches to specification-based testing", ACM SIGSOFT Software Engineering Notes, 14(9): 86-96 (1989).
26. Rosenberg Linda H., "Applying & interpreting object oriented Metrics", (2008).
27. Robert M. Poston, "Testing tool combines best of new and old", IEEE Software. March (2005).
28. Rapps S. and Weyuker E. J., "Selecting

Software Test Data Using Data Flow Information", IEEE Transactions on Software Engineering, pp. 367-375 (1985).

29. Redwine S. and Riddle W., "Software technology maturation", Proceedings of the Eighth International Conference on Software Engineering, 189-200 (1985).
30. Suganya G. and Neduncheliyan S., A Study of Object Oriented Testing Techniques: Survey and Challenges. IEEE Feb. pp: 1 – 5 (2010).
31. Shaw M., "Prospects for an engineering discipline of software", IEEE Software, pp.15-24 (1990).
32. Whittaker J. A., "What is Software Testing? And Why Is It So Hard?", IEEE Software, pp. 70-79 (2000).
33. White L. J. and Cohen E. I., "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, 247-257 (1980).