Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach

Jane Huffman Hayes

Science Applications International Corporation
1213 Jefferson-Davis Highway, Suite 1300
Arlington, Virginia 22202

Abstract. The goal of this paper is to examine the testing of object-oriented systems and to compare and contrast it with the testing of conventional programming language systems, with emphasis on fault-based testing. Conventional system testing, object-oriented system testing, and the application of conventional testing methods to object-oriented software will be examined, followed by a look at the differences between testing of conventional (procedural) software and the testing of object-oriented software. An examination of software faults (defects) will follow, with emphasis on developing a preliminary taxonomy of faults specific to object-oriented systems. Test strategy adequacy will be briefly presented. As a result of these examinations, a set of candidate testing methods for object-oriented programming systems will be identified.

1 Introduction and Overview

Two major forces are driving more and more people toward Object-Oriented Programming Systems (OOPS): the need to increase programmer productivity, and the need for higher reliability of the developed systems. It can be expected that sometime in the near future there will be reusable "trusted" object libraries that will require the highest level of integrity. All this points to the need for a means of assuring the quality of OOPS. Specifically, a means for performing verification and validation (V&V) on OOPS is needed. The goal of this paper is to examine the testing of object-oriented systems and to compare and contrast it with the testing of conventional programming language systems, with emphasis on fault-based testing.

1.1 Introduction to Object-Oriented Programming Systems (OOPS)

Object-Oriented Programming Systems (OOPS) are characterized by several traits, with the most important one being that information is localized around objects (as opposed to functions or data) [3]. Meyer defines object-oriented design as "the construction of software systems as structured collections of abstract data type implementations" [15]. To understand OOPS, several high level concepts must be introduced: objects, classes, inheritance, polymorphism, and dynamic binding.

Objects represent real-world entities and encapsulate both the data and the functions (i.e., behavior and state) which deal with the entity. Objects are run-time instances of classes. *Classes* define a

set of possible objects. A class is meant to implement a user-defined type (ideally an Abstract Data Type (ADT) to support data abstraction). The goal is to keep the implementation details private to the class (information hiding). *Inheritance* refers to the concept of a new class being declared as an extension or restriction of a previously defined class [15].

Polymorphism refers to the ability to take more than one form. In object-oriented systems, it refers to a reference that can, over time, refer to instances of more than one class. The static type of the reference is determined from the declaration of the entity in the program text. The dynamic type of a polymorphic reference may change from instant to instant during the program execution. *Dynamic binding* refers to the fact that the code associated with a given procedure call is not known until the moment of the call at run-time [12]. Applying the principles of polymorphism and inheritance, it can be envisioned that a function call could be associated with a polymorphic reference. Thus the function call would need to know the dynamic type of the reference. This provides a tremendous advantage to programmers over conventional programming languages (often referred to as procedural languages): the ability to request an operation without explicitly selecting one of its variants (this choice occurs at run-time) [16].

1.2 Introduction to Verification and Validation (V&V)

Verification and validation refers to two different processes that are used to ensure that computer software reliably performs the functions that it was designed to fulfill. Though different definitions of the terms verification and validation exist, the following definitions from *ANSI/IEEE Standard 729-1983* [10] are the most widely accepted and will be used in this paper:

Verification is the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. *Validation* is the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

1.3 Introduction to Testing

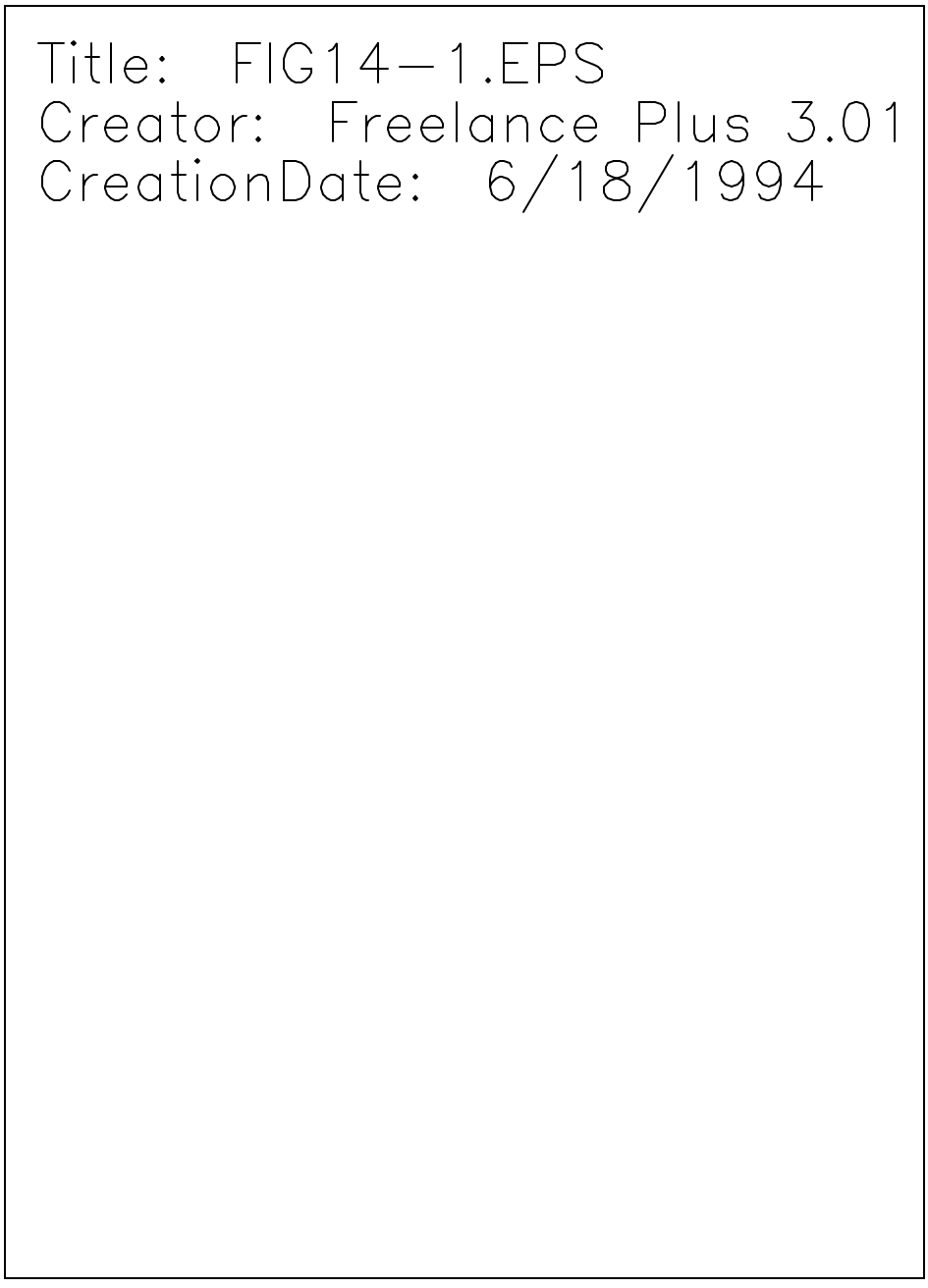
Testing is a subset of V&V. The IEEE definition of testing is "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results." [10]. Conventional software testing is divided into two categories: static testing and dynamic testing. Static testing analyzes a program without executing it while dynamic testing relies on execution of the program to analyze it [5]. The focus of this paper will be dynamic testing.

Dynamic testing is often broken into two categories: black-box testing and white-box testing. Black-box testing refers to functional testing which does not take advantage of implementation

details and which examines the program's functional properties. White-box testing, on the other hand, uses information about the program's internal control flow structure or data dependencies.

1.4 Research Approach

Figure 1.4-1 depicts the research approach to be taken in this paper. The examination of conventional system testing, object-oriented system testing, and the application of conventional testing methods to object-oriented software will be the first area of concern (sections 2.1 and 2.2). The differences between testing of conventional (procedural) software and the testing of object-oriented software will be examined next (section 2.3). An examination of software faults (defects) will follow, with emphasis on developing a preliminary taxonomy of faults specific to object-oriented systems (section 3). Test strategy adequacy will briefly be presented (section 4). As a result of these examinations, a set of candidate testing methods for object-oriented programming systems will be identified (section 5). It is also possible that some object-oriented software faults which are not currently detected by any testing methods (conventional or object-oriented specific) will be uncovered. It should be noted that despite the future research that is needed, a preliminary recommendation on test methods can be made based on the similarities between OOPS and procedural language systems.


Title: FIG14-1.EPS
Creator: Freelance Plus 3.01
CreationDate: 6/18/1994

2 Testing Methods

An examination of testing methods for conventional programming language systems follows as well as a look at the applicability of these testing methods to object-oriented programming systems. A discussion of testing methods specific to OOPS will then be presented.

2.1 Conventional Programming Language System Testing Methods

2.1.1 Testing Methods. Much has been written concerning the testing of conventional (or procedural) language systems. Some of the earlier works include The Art of Software Testing by Myers [22], "Functional Program Testing" by Howden [9], and more recently Software Testing Techniques by Boris Beizer [2]. The first reference [22] focused on explaining testing and leading the reader through realistic examples. It also discussed numerous testing methods and defined testing terminology. Howden's reference focused on functional testing, which is probably the most frequently applied testing method. Finally, Beizer's text provided a veritable encyclopedia of information on the many conventional testing techniques available and in use.

For this paper, the test method taxonomy of Miller is used [19]. Testing is broken into several categories: general testing; special input testing; functional testing; realistic testing; stress testing; performance testing; execution testing; competency testing; active interface testing; structural testing; and error-introduction testing.

General testing refers to generic and statistical methods for exercising the program. These methods include: unit/module testing [24]; system testing [7]; regression testing [23]; and ad-hoc testing [7]. Special input testing refers to methods for generating test cases to explore the domain of possible system inputs. Specific testing methods included in this category are: random testing [1]; and domain testing (which includes equivalence partitioning, boundary-value testing, and the category-partition method) [2].

Functional testing refers to methods for selecting test cases to assess the required functionality of a program. Testing methods in the functional testing category include: specific functional requirement testing [9]; and model-based testing [6].

Realistic test methods choose inputs/environments comparable to the intended installation situation. Specific methods include: field testing [28]; and scenario testing [24]. Stress testing refers to choosing inputs/environments which stress the design/implementation of the code. Testing methods in this category include: stability analysis [7]; robustness testing [18]; and limit/range testing [24].

Performance testing refers to measuring various performance aspects with realistic inputs. Specific methods include: sizing/memory testing [33]; timing/flow testing [7]; and bottleneck testing [24]. Execution testing methods actively follow (and possibly interrupt) a sequence of program execution steps. Testing methods in this category include: thread testing [11]; activity tracing [7]; and results monitoring [23].

Competency testing methods compare the output "effectiveness" against some pre-existing standard. These methods include: gold standard testing [28]; effectiveness procedures [13]; and workplace averages [28]. Active interface testing refers to testing various interfaces to the program. Specific methods include: data interface testing [24]; user interface testing [25]; and transaction-flow testing [2].

Structural testing refers to testing selected aspects of the program structure. Methods in this category include: statement testing [2]; branch testing [17]; path testing [30]; test-coverage analysis testing [2]; and data-flow testing [2]. Error introduction testing systematically introduces errors into the program to assess various effects. Specific methods include: error seeding [7]; and mutation testing [24].

When utilizing conventional programming languages, software systems are usually tested in a bottom-up fashion. First, units or modules are tested and debugged (unit testing). This is followed by integration testing, which exercises sets of modules. Testing of the fully integrated system (system testing) is accomplished next. In some cases system testing is followed by acceptance testing (usually accomplished by/for the customer and/or end user).

2.1.2 Applicability to Object-Oriented Systems. To understand the applicability of conventional testing methods to object-oriented programming systems, it is vital to examine the components of these systems. OOPS can be seen as having five components: (1) objects, (2) their associated messages and methods, (3) hierarchically-organized classes of objects, (4) external interfaces, and (5) tools and utilities. *Objects* are code modules that contain both data and procedures. The *methods* are one type of object-procedure and are responsible for actions of computation, display, or communication with other objects. Communication is accomplished through the sending of *messages*. Objects are described by abstract *classes* (or types). Specific objects are created as instances of a class. Inheritance is used to pass down information from parent classes to their sub-classes. *External interfaces* deal with the connection of OOPS systems to the databases, communication channels, users, etc. *Tools and utilities* refers to general application programs which may be used in building the objects or assisting in any other features of the OOPS [20].

As one might expect, certain OOPS components can be handled very easily by applying conventional testing methods to them, while other components will require distinctive treatment. The hierarchically-organized classes can be viewed as declarative knowledge structures. These components use syntax and naming conventions to explicitly represent details of application knowledge. They are therefore very amenable to a verification and validation philosophy of formal

verification. Formal verification refers to the use of formal mathematical theorem-proving techniques to prove a variety of properties about system components, such as redundancy, incompleteness, syntax violations, and inconsistencies [20]. Although this approach is not yet mature, it is the most effective approach for the class component. Formal methods are static methods, and therefore are outside the scope of this paper.

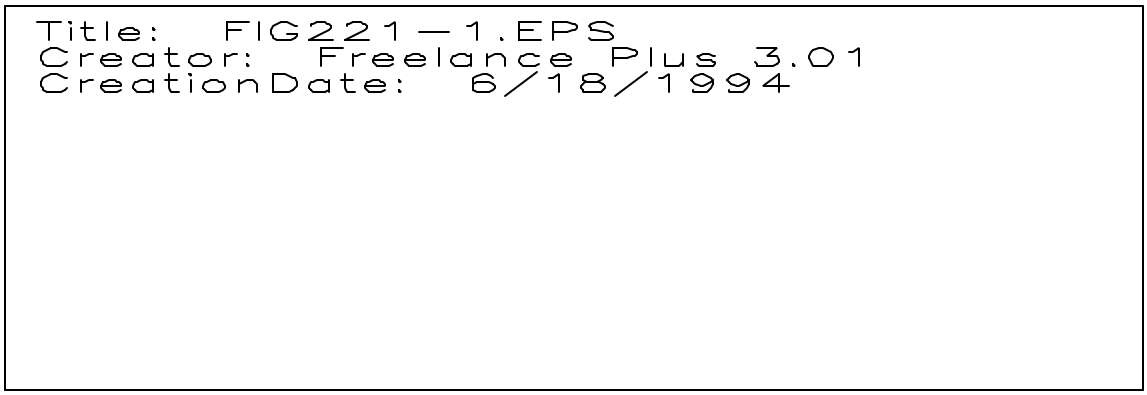
The tools and utilities component is seen as an example of a highly reusable component. Certain objects will also fall into this category (this must be evaluated on a case-by-case basis). A highly reusable component can be reused over a wide variety of applications without needing any customization to specific systems. A certification procedure is recommended for these components which establishes the functional and performance characteristics of the component, independent of the application. Software certification, like formal methods, could easily be the subject of an entire paper and will not be addressed here. The remaining components, including the integrated system itself, some objects, messages and methods, and external interfaces, fall into a third, catch-all category. The traditional set of conventional testing methods can be applied to these components.

In summary, OOPS can be seen as comprising of five components. Of these components, objects which are not highly reusable, messages and methods, and external interfaces can be tested using conventional testing methods. Formal methods should be applied to the class component. Certification procedures should be applied to the tools and utilities component and to highly reusable objects.

2.2 Object-Oriented System Specific Test Methods

In examining the literature on object-oriented programming systems and testing, several testing methods were discovered which are specific to OOPS. The unit repeated inheritance hierarchy testing method, inheritance method, identity method, the set and examine method, and the state-based testing method will be described below.

2.2.1 Unit Repeated Inheritance (URI) Hierarchy Method. Repeated inheritance is defined as a class (e.g., class D) that multiply inherits from two or more classes (e.g., classes B and C), and these classes (B and C) are descendants of the same parent class (e.g., class A). Repeated inheritance is depicted in Figure 2.2.1-1 using the classes described in the preceding sentence.



Chung and Lee [5] assert that errors such as name-confliction will arise when repeated inheritance is used. They build a directed graph and then apply an algorithm to find all unit repeated inheritances. The graph consists of classes and inheritance edges. A breadth-first traverse algorithm is used to traverse all the root classes. Parent classes are added to the children ancestor sets. The ancestor sets are then examined for unit repeated inheritance. Since there may be too many unit repeated inheritance instances to test, they classify all the repeated inheritances according to their euler region numbers (r) [4]. A hierarchy is built by dividing all the repeated inheritances into a set of closed regions denoted as URIs(n) where 1 <= n <= r. For example:

URIs(1): require every class in a repeated inheritance graph to be exercised at least once.
URIs(2): require every unit repeated inheritance with r = 2 to be exercised at least once
URIs(3): require every closed region with r = 3 to be exercised at least once
.
.

[5]. After finding the hierarchy, McCabe's cyclomatic testing strategy is applied (a structural test method) [14].

2.2.2 Inheritance Method. Smith and Robson [29] have identified a framework for testing object-oriented systems which uses seven different testing strategies (test methods in the terminology of this paper). Though not all these strategies are specific to object-oriented systems, the inheritance method is. The inheritance method uses regression analysis to determine which routines should be tested (when a change has been made to the system) and then performs the tests based upon how the superclass was successfully tested. This applies to sub-classes inherited from the parent class. The sub-class under test is treated as a flattened class except that the routines from the parent that are unaffected by the subclass are not retested [29].

2.2.3 Identity Method. Another method proposed by Smith and Robson is the identity method. This method searches for pairs (or more) of routines that leave the state as it was originally (before any routines were invoked). This list of routines is reported to the tester who can examine the pairs and ensure that the unaltered state is the desired result [29].

2.2.4 Set and Examine Method. This Smith and Robson method is similar to the identity method. Pairs of routines that set and examine a particular aspect of the state are related and are used in conjunction to run tests. For example, a clock object may have one routine that sets the time then another that checks the time. The time can be set, then immediately checked using this pair of routines. Boundary and error values can be checked using this method [29].

2.2.5 State-Based Testing Method. Turner and Robson [31] have suggested a new technique for the validation of OOPS which emphasizes the interaction between the features and the object's state. Each feature is considered as a mapping from its starting or input states to its resultant or output states affected by any stimuli [31]. Substates are defined which are the values of a data item at a specific point in time. These are then analyzed for specific and general values. Next, the set of states that the I^{th} feature actually accepts as input (I_i) and the set of states it is able to generate as output (O_i) are determined for all the features of the class. Test cases are then generated using general guidelines provided. For example, one test case should allocate one substate per data item. Turner and Robson have found this technique to work best for classes which have many interacting features.

2.3 Differences Between Testing Conventional and Object-Oriented Software

Some of the differences between testing conventional and object-oriented software were examined in Section 2.1.2 above. Firesmith [8] has examined these differences in some depth. Table 2.3-1 summarizes the differences noted by Firesmith.

Table 2.3-1. Differences Between Testing of Procedural and Object-Oriented Software [8].

Test Method	With Procedural Software	With Object-Oriented Software
Unit Testing	Test individual, functionally cohesive operations	Unit testing is really integration testing, test logically related operations and data
Integration Testing	Test an integrated set of units (operations and common global data)	Object-oriented's unit testing. No more bugs related to common global data (though could have errors associated with global objects and classes)
Boundary Value Testing	Used on units or integrated units and systems	Of limited value if using strongly typed object-oriented languages and proper data abstraction is used to restrict the values of attributes
Basis Path Testing	Generally performed on units	Limited to operations of objects. Must address exception handling and concurrency issues (if applicable). Lowered complexity of objects lessens the need for this
Equivalence and Black-Box Testing are	Used on units, integrated units and systems	Emphasized for object-oriented: objects black boxes, equivalence classes are messages

The difference between unit testing for conventional software and for object-oriented software, for example, arises from the nature of classes and objects. Unit testing really only addresses the testing of individual operations and data. This is only a subset of the unit testing that is required for object-oriented software since the meaning and behavior of the resources encapsulated in the classes depend on other resources with which they are encapsulated [8]. Integration testing (for conventional software) then is truly the unit testing of object-oriented software. The points made by Firesmith about boundary value testing and basis path testing are debatable in this author's opinion. Even with strongly typed languages, poor usages can still occur, and testing should be used to ensure that data abstraction and value restriction are implemented properly. Similarly, an argument that the object-oriented paradigm lowers complexity and hence lessens the need for basis path testing could be made just as easily for many structured languages such as Ada.

In summary, there are differences between how conventional software and object-oriented software should be tested. Most of these differences are attributable to the class component of object-oriented systems, and to the fact that these classes are not comparable to a "unit" in conventional software. However, the differences are minor and it is apparent that conventional testing methods can readily be applied to object-oriented software. As a minimum, a tester of an OOPS would want to apply the unit testing, integration testing, and black box testing methods discussed here. One or more of the OOPS-specific test methods would also be advisable to complete the test strategy.

3 OOPS-Specific Software Faults

A fault is defined as a textual problem with the code resulting from a mental mistake by the programmer or designer [10]. A fault is also called a defect. Fault-based testing refers to the collection of information on whether classes of software faults (or defects) exist in a program [32]. Since testing can only prove the existence of errors and not their absence, this testing approach is a very sound one. It is desirable to be able to implement such a testing approach for object-oriented systems. Although lists of error types can be found in the current object-oriented literature, at present there does not exist a comprehensive taxonomy of defect types inherent to object-oriented programming systems. This paper takes a first step toward such a taxonomy by consolidating the fault types found in the literature.

Three major sources of object-oriented faults were examined: [8], [21], and [27]. Each source examined object-oriented faults and attempted to describe the types of test methods that could be applied to detect the faults. Firesmith concentrated on conventional test methods such as unit testing and integration testing, while Miller et al concentrated on a prototype static analyzer called Verification of Object-Oriented Programming Systems (VOOPS) for detecting faults. Purchase and Winder [27] presented nine types of faults, seven which are detectable using debugging tools. A preliminary list of object-oriented software defects (faults) is presented in Table 3.0-1. For each defect, a proposed test method for detecting the defect is provided.

In order to cull the list of faults in Table 3.0-1 into a workable taxonomy, more fault types must be identified. Duplicate/related fault types must be eliminated or grouped. Dynamic testing methods should be identified to detect each of the faults currently detected by VOOPS (this is not mandatory, one can simply broaden the definition of testing to include static and dynamic methods). Similarly, static testing methods should be identified for as many fault types as possible. The taxonomy must then be organized to either address OOPS components (as does Firesmith), the object-oriented model (as does Miller), or causal and diagnostic fault types (as does Purchase and Winder). These are all areas for future research.

The approach proposed in this paper differs from that of Miller, Firesmith, Purchase & Winder in that it looks not only at object-oriented faults (as does Miller and Purchase & Winder) and not only at conventional methods applied to these faults (as does Firesmith). It looks at both of these items plus examines methods specific to OOPS. It is therefore a step toward a more comprehensive approach.

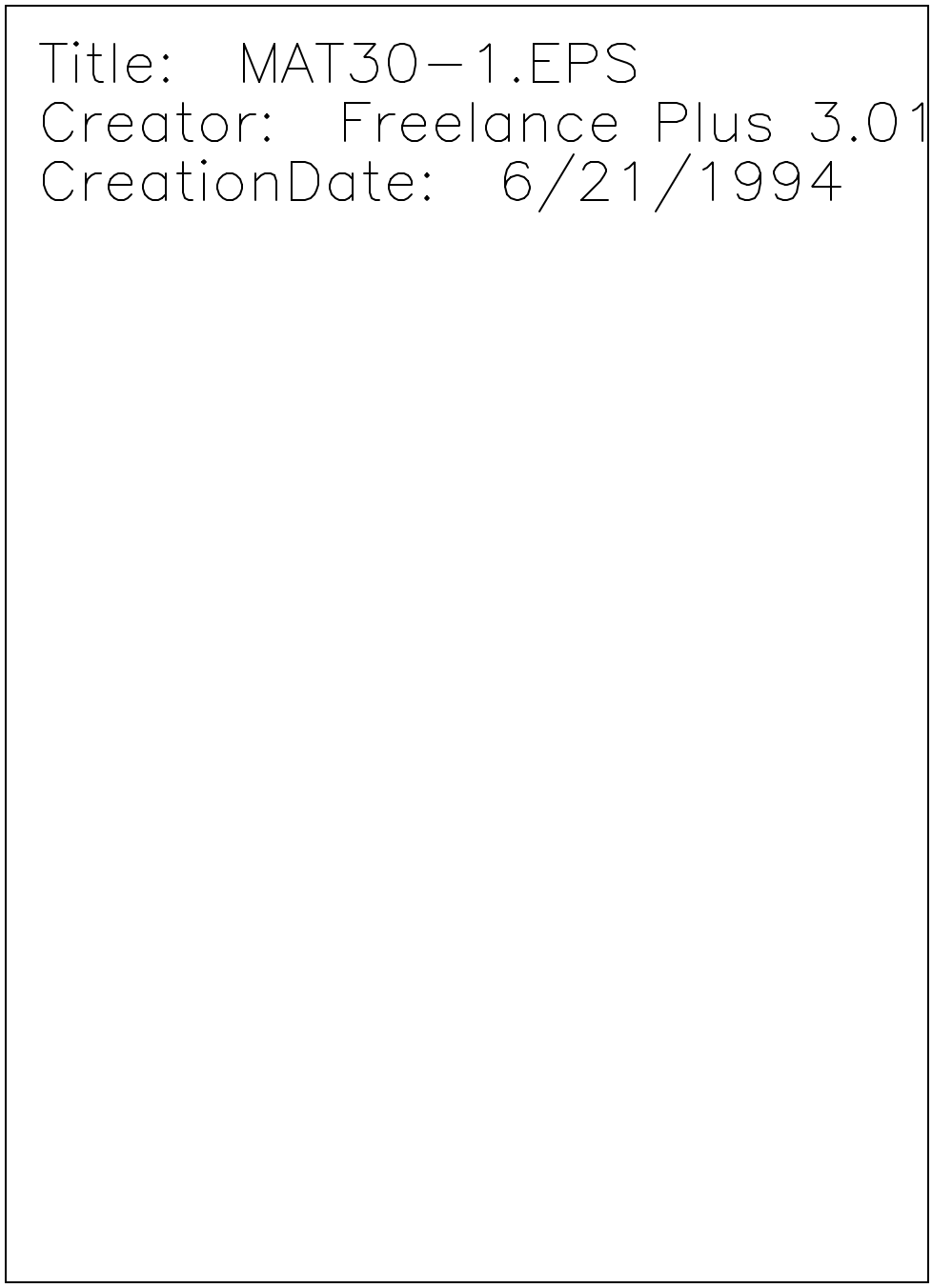
4 Test Adequacy

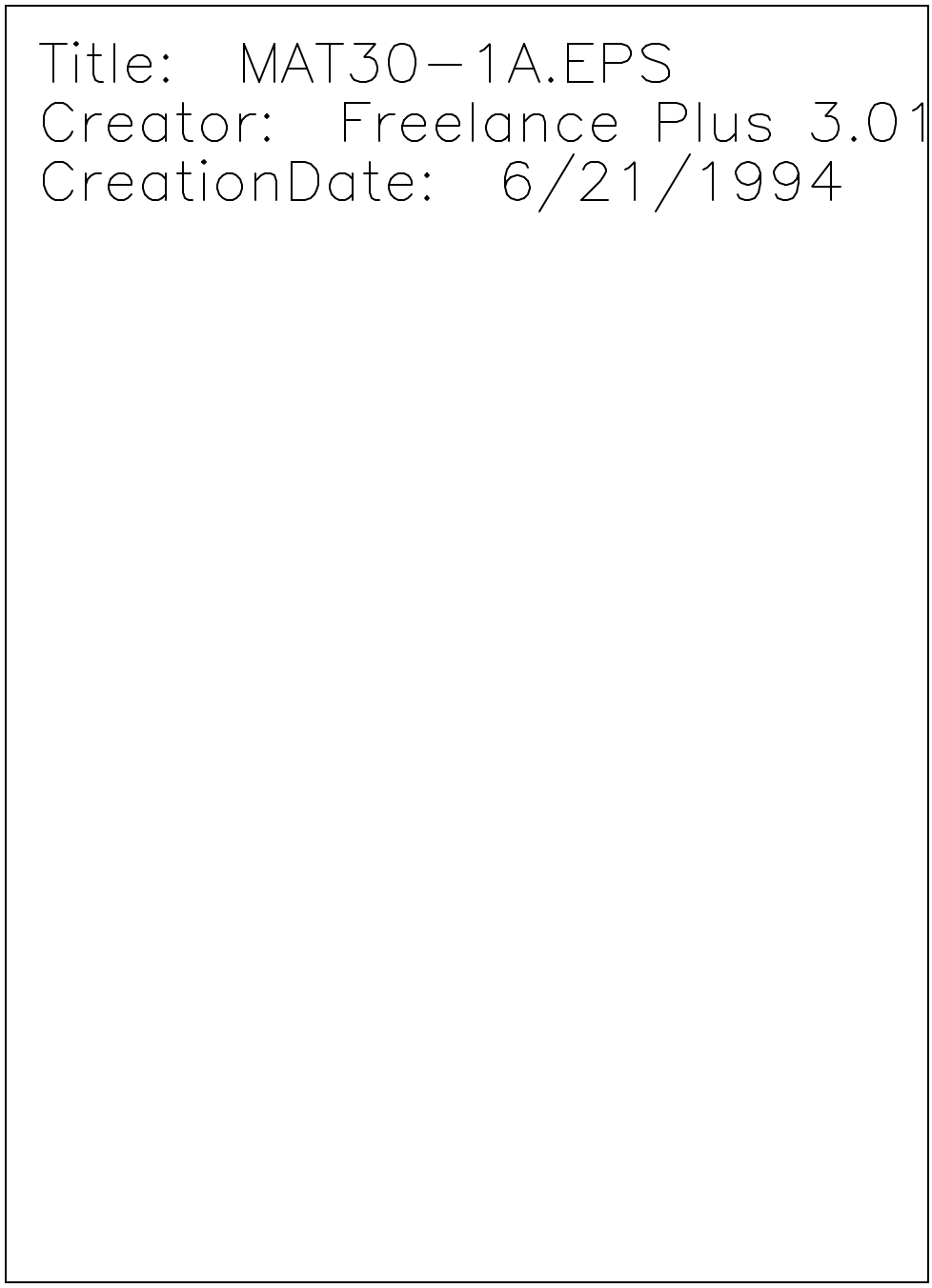
After examining the many test methods available for application to OOPS, the faults specific to OOPS, and the test methods that can be applied to detect these faults, a tester should be able to devise a set of suggested methods to apply to a given OOPS (some suggestions were made regarding this in Section 2.3). This set of test methods is termed a testing strategy or approach. The tester should then wonder whether or not this set of test methods is adequate to thoroughly exercise the software. Section 4 examines the topic of test strategy adequacy.

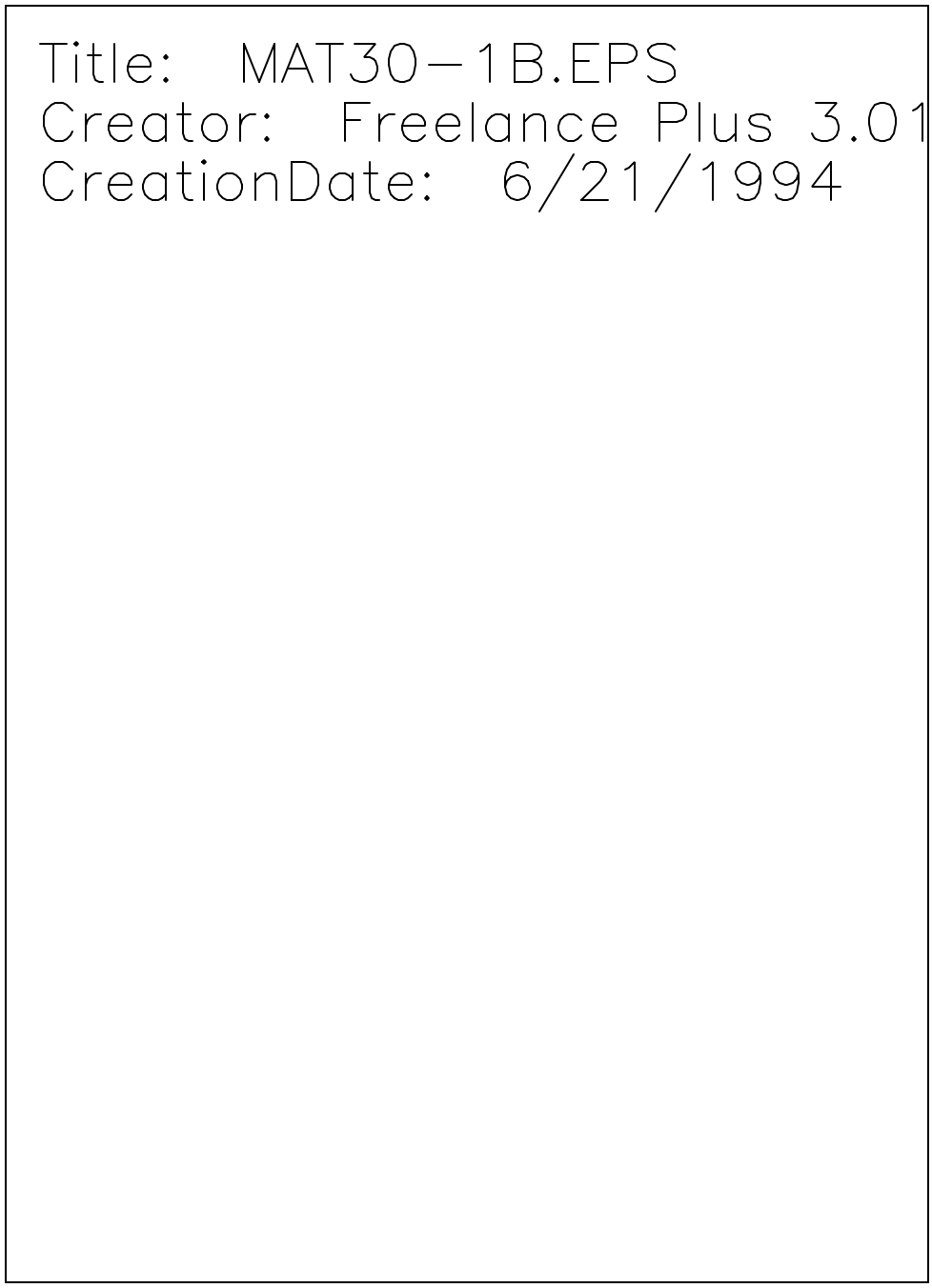
4.1 Test Adequacy Axioms

Elaine Weyuker has postulated eleven axioms for test set adequacy. The first axiom, *applicability*, states that for every program there exists an adequate test set. The axiom of *non-exhaustive applicability* states that there is a program P and a test set T such that P is adequately tested by T and T is not an exhaustive test set. The *monotonicity* axiom asserts that if T is adequate for P, and T is a subset of T', then T' is adequate for P. The *inadequate empty set* axiom postulates that the empty set is not an adequate test set for any program [34].

The *renaming* axiom states that if P is a renaming of Q, then T is adequate for P if and only if T is adequate for Q. The *complexity* axiom postulates that for every n, there is a program P such that P is adequately tested by a size n test set, but not by an size n - 1 test set.


Title: MAT30-1.EPS
Creator: Freelance Plus 3.01
CreationDate: 6/21/1994


Title: MAT30-1A.EPS
Creator: Freelance Plus 3.01
CreationDate: 6/21/1994


Title: MAT30-1B.EPS
Creator: Freelance Plus 3.01
CreationDate: 6/21/1994

The *statement coverage* axiom states that if T is adequate for P, then T causes every executable statement of P to be executed. The *antiextensionality* axiom asserts that there are programs P and Q such that P computes the same function as Q (they are semantically close), and T is adequate for P but is not adequate for Q [34].

The *general multiple change* axiom states that there are programs P and Q which are the same shape (syntactically similar), but a test set T that is adequate for P is not adequate for Q. The *antidecomposition* axiom asserts that there exists a program P and component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q. Finally, the *anticomposition* axiom postulates that there exist programs P and Q and test set T such that T is adequate for P, and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P;Q (where P;Q is the composition of P and Q) [34].

4.2 Evaluation of Adequacy of Test Strategies

It is desirable to apply these axioms to a set of test methods (test strategy or test set) to determine their adequacy. As one can imagine, this is not a simple task. Perry and Kaiser applied Weyuker's axioms to a test strategy pertaining to inherited code. They examined the popular philosophy that inherited code need not be retested when reused. Using the adequacy axioms, they found this philosophy to be erroneous. For example, if only one module of a program is changed, it seems intuitive that testing should be able to be limited to just the modified unit. However, the anticomposition axiom states that every dependent unit must be retested as well. Therefore one must always perform integration testing in addition to unit testing [26]. Another interesting observation made by Perry and Kaiser pertains to the testing of classes. When a new subclass is added (or an existing subclass is modified) all the methods inherited from each of its ancestor superclasses must be retested. This is a result of the antidecomposition axiom.

4.3 Future Research on Test Set Adequacy

Unfortunately, the work to date in this area has either been highly generic (as in the case of Weyuker's axioms) or has been too specific (Perry and Kaiser examined but one scenario of the object-oriented paradigm). In order for a tester to evaluate the set of test methods selected for use with an OOPS, there must be a way of easily examining each of the methods in the strategy, as well as the synergism of these methods. The adequacy evaluation should not depend on assumptions such as the system having previously been fully tested (e.g., the Perry and Kaiser scenario only examined the adequacy of test sets in testing changed code). Translating the very solid axioms of Weyuker into a usable method for evaluating actual test strategies is an area for further research.

5 Recommended Test Strategy for OOPS

There is much research that remains before well informed decisions can be made regarding which test methods to apply to OOPS. However, noting the great similarities between OOPS and procedural language systems, a preliminary set of methods can be recommended. Based on the conventional testing methods available which are applicable to object-oriented software, the object-oriented software specific testing methods available, the taxonomy of object-oriented software faults, and largely the author's personal testing experience, the following general test strategy (set of test methods) is recommended for object-oriented software:

Compilation Testing
Unit Testing
Unit Repeated Inheritance (URI) Hierarchy Method
Integration Testing
Boundary Value Testing
Basis Path Testing
State-Based Testing
Equivalence Partitioning Testing
Black-Box Testing
Acceptance Testing
Performance Testing

Note that Smith and Robson's identity, inheritance, and set and examine test methods were not selected. In the author's opinion, these methods are superseded by other selected methods such as the URI hierarchy method and unit testing.

The test methods are listed in the suggested execution order, with compilation testing being the obvious method to run first. It is also obvious that unit testing must precede integration testing which must precede acceptance testing. The order of the other methods is not as obvious and can be changed to suit the tester's needs. This test strategy is highly generic and should be tailored for the specific object-oriented system which is the subject of testing.

6 Summary and Conclusions

This paper first examined object-oriented programming systems, verification and validation, and testing. The research approach was presented next. Conventional system testing was examined, followed by object-oriented system testing, and a comparison of the testing of conventional and object-oriented software. Over sixty conventional testing methods were identified, but only four object-oriented software specific methods were discovered. The need for further research in the area of object-oriented software specific testing methods is apparent. The result of this examination was that conventional methods are highly applicable to object-oriented software, with

classes and tools and utilities being OOPS components which may require special attention. Unit testing, integration testing, and black-box testing were found to be especially useful methods for application to OOPS.

A proposed taxonomy of object-oriented defects (faults) was then presented, with an emphasis on the testing methods which could be applied to detect each fault. This is an area where further research is required. A discussion of test set adequacy followed. Though sound axioms for test set adequacy exist, there is not a usable means of applying these axioms to a set of test methods (such as those presented in Section 5.0). This presents another area for future research. Finally, a generic set of suggested test methods (a test strategy) for OOPS was proposed. This test strategy must be customized for the specific OOPS to be tested.

Acknowledgment

I would like to thank Dr. David Rine and Dr. Bo Sanden of George Mason University for their helpful comments and suggestions about this paper.

References

1. Barnes, M., P. Bishop, B. Bjarland, G. Dahll, D. Esp, J. Lahti, H. Valisuo, P. Humphreys. "Software Testing and Evaluation Methods (The STEM Project)." OECD Halden Reactor Project Report, No. HWR-210, May 1987.

2. Beizer, Boris. Software Testing Techniques. Second Edition. New York, Van Nostrand Reinhold, 1990.

3. Berard, Edward V. Essays on Object-Oriented Software Engineering. Prentice-Hall, Englewood Cliffs, New Jersey, 1993, p. 10.

4. Berge, C. Graph and Hypergraphs. North-Holland, Amsterdam, The Netherlands, 1973.

5. Chung, Chi-Ming and Ming-Chi Lee. "Object-Oriented Programming Testing Methodology", published in Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering, IEEE Computer Society Press, 15 - 20 June 1992, pp. 378 - 385.

6. Davis, A. Software Requirements: Analysis and Specification. New York, Prentice-Hall, Inc., 1990.

7. Dunn, R.H. Software Defect Removal. New York, McGraw-Hill, 1984.

8. Firesmith, Donald G. "Testing Object-Oriented Software," published in Proceedings of TOOLS, 19 March 1993.

9. Howden, W. E., "Functional Program Testing," IEEE Transactions on Software Engineering, SE-6(2): March 1980, 162-169.

10. IEEE 729-1983, "Glossary of Software Engineering Terminology," September 23, 1982.

11. Jensen, R.W. and C.C. Tonies. Software Engineering. Englewood Cliffs, NJ, Prentice-Hall, 1979.

12. Korson, Tim, and John D. McGregor. "Understanding Object-Oriented: A Unifying Paradigm." Communications of the ACM, Volume 33, No. 9, September 1990, pp. 40-60.

13. Llinas, J., S. Rizzi, and M. McCown, "The Test and Evaluation Process for Knowledge-Based Systems." SAIC final contract report, Contract #F30602-85-G-0313 (Task 86-001-01), prepared for Rome Air Development Center, 1987.

14. McCabe, T.J. "Structured Testing: A Testing Methodology Using the McCabe Complexity Metric," NBS Special Publication, Contract NB82NAAK5518, U.S. Department of Commerce, National Bureau of Standards, 1982.

15. Meyer, Bertrand. Object-oriented Software Construction. Prentice-Hall, New York, NY, 1988, p. 59, 62.

16. Meyer, Bertrand. Eiffel: The Language. Prentice-Hall, New York, NY, 1992, p. 17.

17. Miller, Edwin. "Better Software Testing" Proceedings of Third International Conference on Software for Strategic Systems, 27-28 February 1990, Huntsville, AL, 1-7.

18. Miller, Lance A., "Dynamic Testing of Knowledge Bases Using the Heuristic Testing Approach." in Expert Systems with Applications: An International Journal, 1990, 1, 249-269.

19. Miller, Lance A., Groundwater, Elizabeth, and Steven Mirsky. Survey and Assessment of Conventional Software Verification and Validation Methods (NUREG/CR-6018). U.S. Nuclear Regulatory Commission, April 1993, p. 9, 33, 35.

20. Miller, Lance A., Hayes, Jane E., and Steven Mirsky. Task 7: Guidelines for the Verification and Validation of Artificial Intelligence Software Systems. Prepared for United States Nuclear Regulatory Commission and the Electric Power Research Institute. Prepared by Science Applications International Corporation, May 28, 1993.

21. Miller, Lance A. Personal communication. September 1993.

22. Myers, G.J. The Art of Software Testing. Wiley, New York, New York, 1979.

23. NBS 500-93, "Software Validation, Verification, and Testing Technique and Tool Reference Guide," September 1982.

24. Ng, P. and R. Yeh (Eds.). Modern Software Engineering: Foundations and Current Perspectives. New York, Van Nostrand Reinhold, 1990.

25. NUREG/CR-4227, Gilmore, W., "Human Engineering Guidelines for the Evaluation and Assessment of Video Display Units." July 1985.

26. Perry, DeWayne E. and Gail E. Kaiser. "Adequate Testing and Object-Oriented Programming," Journal of Object-Oriented Programming, 2(5):13-19, 1990.

27. Purchase, J.A. and R.L. Winder. “Debugging Tools for Object-Oriented Programming,” Journal of Object-Oriented Programming, Volume 4, Number 3, June 1991, pp. 10-27.

28. Rushby, J., "Quality Measures and Assurance for AI Software." NASA Contractor Report No. 4187, prepared for Langley Research Center under Contract NAS1-17067, October 1988.

29. Smith, M.D. and D.J. Robson. "A Framework for Testing Object-Oriented Programs," Journal of Object-Oriented Programming, June 1992, pp. 45-53.

30. Tung, C., "On Control Flow Error Detection and Path Testing," Proceedings of Third International Conference on Software for Strategic Systems, Huntsville, AL, 27 - 28 February 1990, 144-153.

31. Turner, C.D. and D.J. Robson. “The State-based Testing of Object-Oriented Programs,” Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM-93), Montreal, Quebec, Canada, September 27 - 30, 1993, David Card - editor, pp. 302-310.

32. Voas, Jeffrey M. "PIE: A Dynamic Failure-Based Technique," IEEE Transactions on Software Engineering, Volume 18, No. 8, August 1992.

33. Wallace, Dolores R. and Roger U. Fujii. "Software Verification and Validation: An Overview." IEEE Software, Vol. 6, No. 3, May 1989.

34. Weyuker, Elaine J. "The Evaluation of Program-Based Software Test Data Adequacy Criteria," Communications of the ACM, 31:6, June 1988, pp. 668-675.