Testing Object-Oriented Software

15 December 1992

by

Donald G. Firesmith
President, Advanced Software Technology Specialists (ASTS)
17124 Lutz Road
Ossian, IN 46777
voice: (219) 639-6305
fax: (219) 747-9389

Director of Object Technology, Software Consulting Specialists (SCS)
1943 Lakeview Drive
Fort Wayne, IN 46808
voice: (219) 432-3975
fax: (219) 432-3651

Abstract

Unlike traditional procedural software, pure object-oriented languages (e.g., Eiffel, Smalltalk), hybrid object-oriented languages (e.g., Ada9X, C++), and object-based languages (e.g., Ada83, Modula-2) have radically different structures and behaviors based on objects, classes of objects, subassemblies of objects and classes, message passing, and inheritance. Rather than being functionally decomposed according to the classic waterfall development cycle, object-oriented software is most often incrementally analyzed, designed, coded, and tested using new development methods in accordance with iterative and recursive development cycles. These fundamental changes cause major impacts on the type and scheduling of testing.

This paper discusses the definition and purpose of software testing, the structure and behavior of object-oriented software, and the testing of object-oriented software including the testing of objects, classes, and their resources (e.g., messages, exceptions, attributes, operations). This paper also discusses the testing of classification and inheritance hierarchies as well as the testing of scenarios involving objects and classes.

1) Definition and Purpose of Software Testing

Software testing is the execution of software for the explicit purpose of finding errors that cause failures (i.e., deviations between the actual and intended structure or behavior of the software). Object-oriented testing (OOT) is the appropriate testing of object-oriented software. OOT is therefore testing in a manner that takes into account the different types of errors that occur in object-oriented software because of its specific structures and behavior.

The errors that can be found by testing may cause deficiencies in the following goals of software engineering:

- *Correctness* is the degree to which the software meets its specified requirements.
- *Efficiency* is the degree to which the software uses hardware resources effectively.
- Interoperability is the degree to which the software correctly interacts with other software.
- *Portability* is the ease with which the software can be transitioned to another hardware or software environment.
- *Reliability* is the degree to which the software behaves correctly over time.
- *Robustness* is the degree to which the software continues to function correctly under abnormal circumstances.
- *Safety* is the degree to which the software works without accidental harm to life or property.
- *Security* (a.k.a., integrity) is the degree to which the software protects itself from unauthorized access or modification.
- *User-friendliness* is the ease with which humans can use the software.

While testing can (and should) be used to improve the quality of the tested software, it cannot ensure that arbitrary software contains no errors because exhaustive testing to find all errors in non-trivial software is either impossible or impractical with current and foreseeable technology. Testing should therefore be performed in a manner that maximizes the number errors found (prioritized by probability of occurrence and severity of impact) while minimizing the effort (in terms of cost and schedule) required to prepare, run, and evaluate the results of the testing.

Testing can be performed at five different levels because errors can occur at five different levels. Errors may be in parts of units, in individual units, in subassemblies (i.e., small collections of interacting modules), in hierarchies of units, or in assemblies of subassemblies. Whereas all of these levels of testing are impacted by the use of object-oriented software, the primary impacts occur at the level of individual units (i.e., unit testing) and the level of hierarchies (e.g., inheritance testing and scenario-based testing of object-oriented software).

2) The Structure and Behavior of Object-Oriented Software

The basic building block of the functional decomposition methods and procedural languages is the "function", a relatively independent operation that was often specified algorithmically. Software requirements specifications, designs, and code where organized in terms of the major functions the software performs, their subfunctions, and their subfunctions. Because decomposing requirements and designs by function scatters the data throughout the resulting software with significant data coupling between many of the resulting functions, data dictionaries were necessary to keep track of common data definitions. Using these methods and languages, the functions and data were thus analyzed, designed, coded, and documented separately as individual functional units and common global data, with significant effort required to ensure consistency and protect against accidental corruption.

This decomposition of the specifications and designs corresponded to the basic building block of the older languages such as ALGOL, C, COBOL, FORTRAN, and Pascal. The units (i.e., subroutines) in these procedural languages:

- theoretically implemented a single functional requirement or abstraction

- should therefore maximize functional cohesion
- interfaced with other units via:
  - subroutine calls
  - common global data

The structure of such functional specifications, designs, and software was relatively simple because there were only two types of entities (i.e.., functions and common global data) and two types of hierarchies (i.e., the calling "tree" and the data structures). For this reason, data flow diagrams (which showed how the functions decomposed and how the functions and data interacted) and classic structure charts (which showed the calling trees of the functions) were both appropriate and reasonably adequate. Figure 1 shows the structure of functional software, ignoring aggregation (i.e., nesting) and database interfaces.

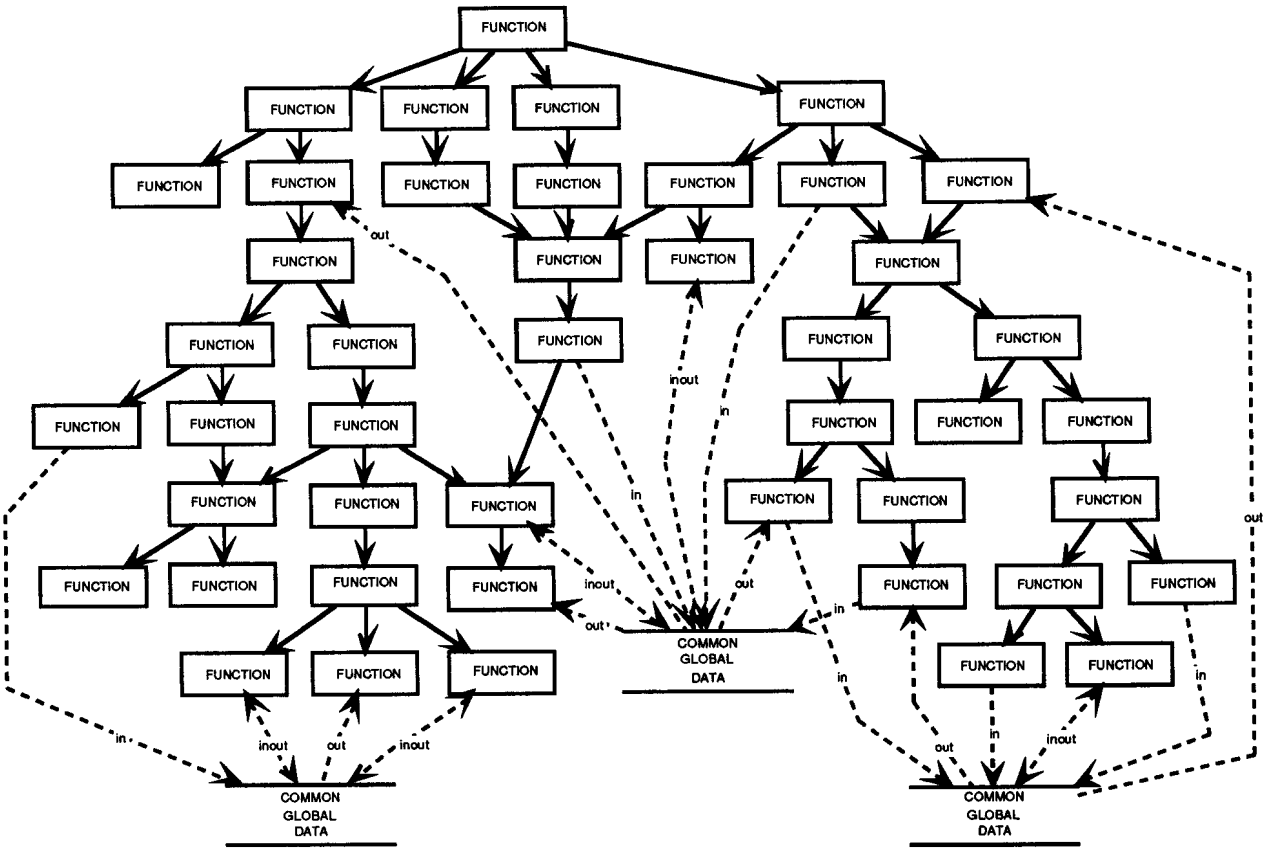

Figure 1: Structure of Functional Software

Object-oriented software has radically different structures than software written in more traditional languages. The basic building blocks of object-oriented methods different and larger: objects, classes of objects, and subassemblies of objects and classes. Each of these, in turn, has its own hierarchies. Objects interact primarily via message passing, so there are interaction hierarchies of collaborating objects. Classes are related primarily via inheritance, so there are inheritance (i.e., "a kind of") hierarchies of subclasses and

superclasses. The object and class hierarchies are connected via classification (i.e., "is a") relationships between instances (i.e., objects) and their classes. Objects and classes are grouped into subassemblies for methodological and management reasons, and there exist hierarchies of subassemblies related by dependency. Object-oriented software (and its designs and specifications) thus have three, relatively orthogonal structures. Figure 2 shows the basic structure of object-oriented software in terms of its collaborating objects. Figure 3 shows the basic structure of object-oriented software in terms of its classes of objects related via inheritance. Figure 4 shows the basic structure of object-oriented software in terms of its subassemblies of objects and classes.

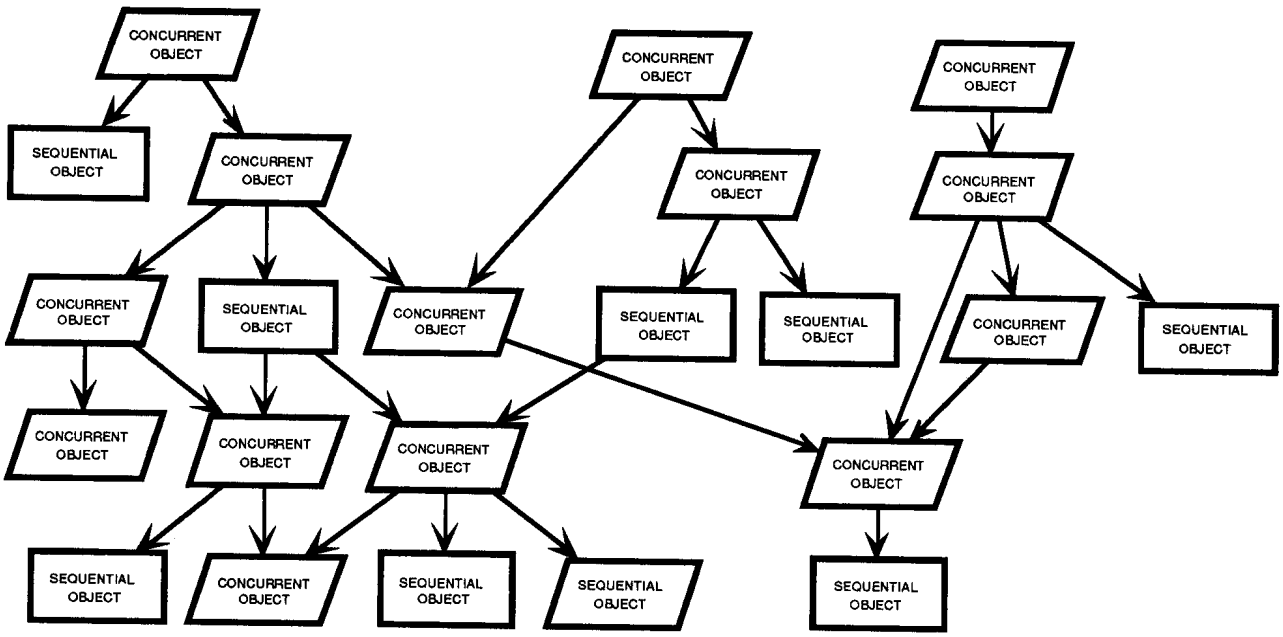

Figure 2: Structure of Software in terms of Objects

These different structures cause different types and proportions of errors that require different approaches to testing if the aforementioned goals of testing are to be optimized. Previously, unit testing of functionally decomposed software was the testing of individual, functionally-cohesive operations. Because the meaning and behavior of the resources encapsulated within objects and classes depend on the other resources with which they are encapsulated, the testing of individual operations and data is now only part of the unit testing of object-oriented software. Previously, integration testing of functionally decomposed software was the testing of an integrated set of operations and common global data. The testing of a logically related set of operations and data is now object-oriented unit testing, the independent testing of individual objects and classes using test software (e.g., test drivers and test stubs) and lowerCASE tools (e.g., debuggers, profilers, performance analyzers). Bugs related to common global data no longer exist because common global data no longer exists; all data (i.e., attributes) are now local within objects. Boundary value testing is of limited value when the object-oriented language is strongly typed and proper data abstraction is used to restrict the values of attributes. Basis path (or structured) testing is now limited to pieces of units (i.e., operations of objects) and must address exception handling and concurrency issues (if the object is concurrent). The

importance of equivalence class and blackbox testing is now emphasized due to the fact that objects can be considered software black boxes, although the equivalence classes of test data are now defined in terms of messages and the protocols of the objects.

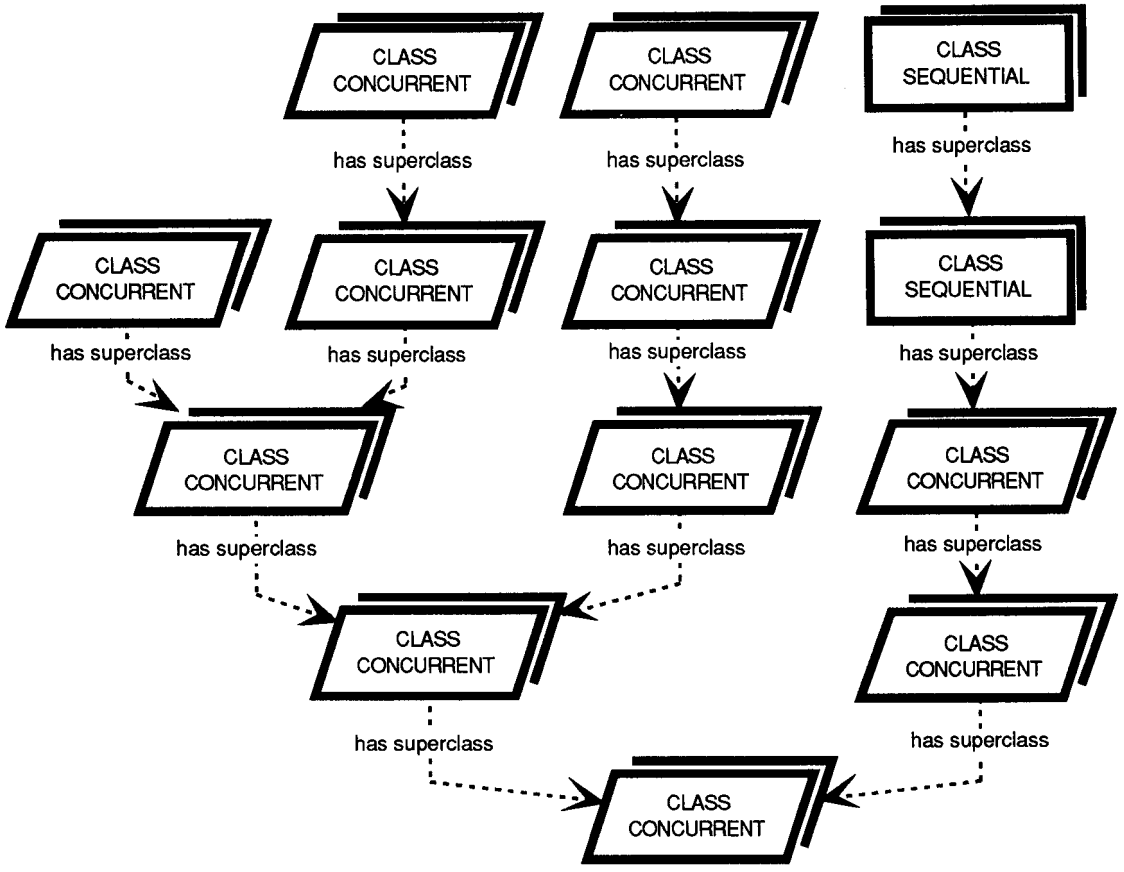

Figure 3: Structure of Software in terms of Classes

3) Testing Object-Oriented Software

3.1) Testing Objects

Objects are models of application domain entities that typically:

- Are instances of classes
- Encapsulate (i.e., localize and hide) attribute types, attributes, operations, and exceptions
- Send and receive messages

Objects are the primary executable modules of object-oriented programming languages. Because objects often contain 10 to 20 operations (a.k.a., methods, services, member functions), objects should not be functionally cohesive. Therefore, objects typically do not perform a single function. Unit testing is thus object testing, rather than subroutine testing (as it was in languages such as C, COBOL, and FORTRAN).

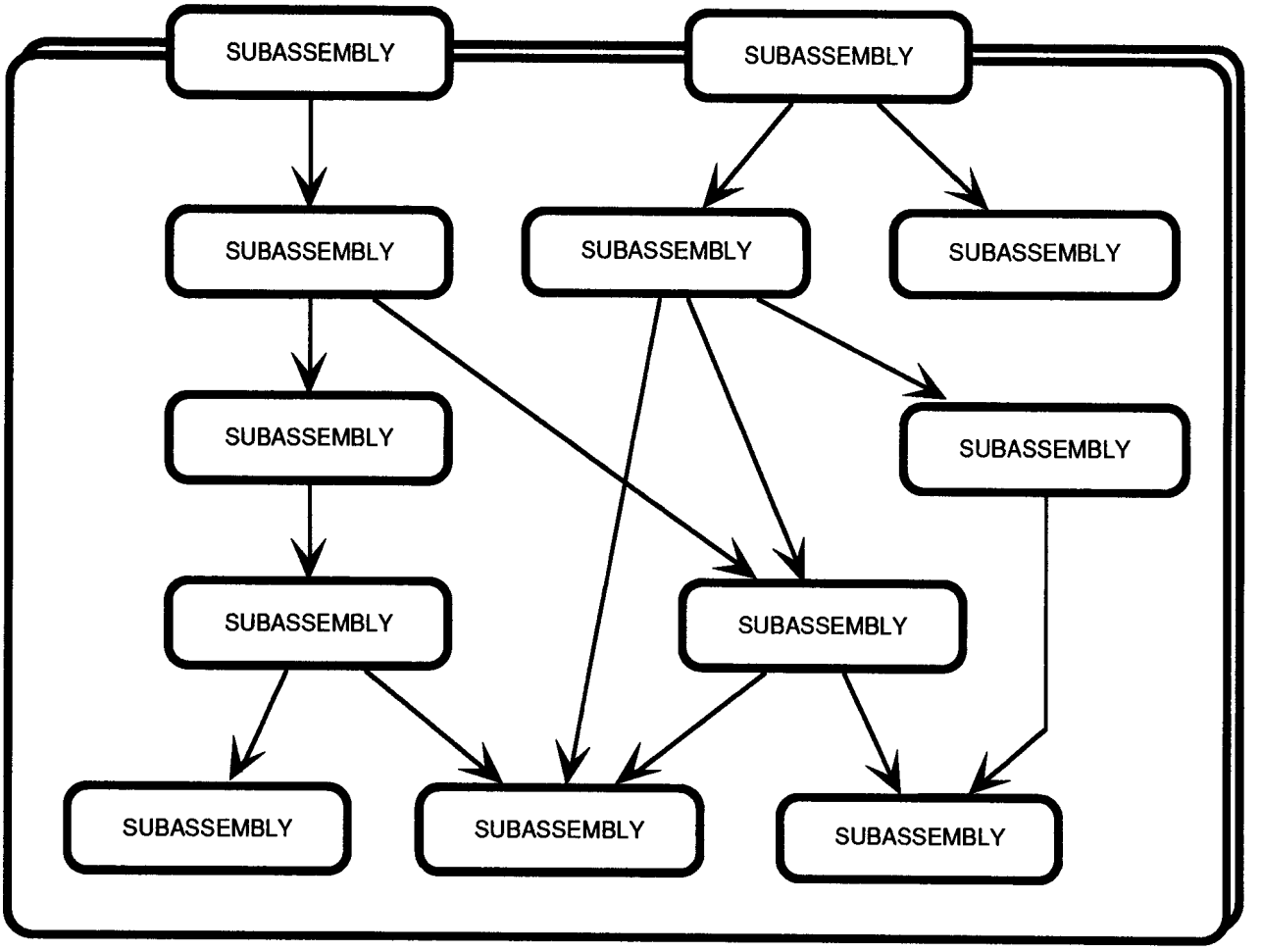

Figure 4: Structure of Software in terms of Subassemblies

Production-quality debuggers are critical because objects encapsulate their resources, which are ordinarily accessible only via their interface protocol of messages, visible attribute types, and exceptions. Test drivers must send appropriate messages to the associated objects under test and must be able to properly handle both data returned as parameters of messages and also exceptions raised by the objects under test. Test stubs must handle and potentially log messages sent by the objects under test.

Testers should test for the following errors associated with objects:

- Abstraction violated
- Persistence problems:
  - Object not stored when necessary
  - Object stored unnecessarily
- Documentation out of sync with the code
- Incorrect state model
- Invariants violated
- Failures associated with instantiation and destruction
- Concurrency problems such as:
  - Unnecessary polling

  - Starvation
  - Deadlock
  - Priority inversion
- Inadequate performance
  - Excessive memory utilization
  - Memory not deallocated when finished
  - Inadequate performance
  - Missed deadlines
- Failure to meet required behavior of the object
- Failures associated with messages, exceptions, attributes, or operations (see paragraphs 3.3 through 3.6 below)

3.2) Testing Classes

Classes are templates for the instantiation (i.e., construction) of objects. As such, them must define all of the resources of their instances including acceptable messages, exceptions, attribute types, attributes, and operations. Classes may also have class attributes (e.g., number of instances) and class operations (e.g., construct an instance). Like objects, classes primarily interact via message passing.

Testing of classes is often indirect. In many object-oriented programming languages (OOPLs), classes are compile-time entities used to create run-time objects. Classes in such languages do not execute and may therefore not be tested directly. Classes are therefore often tested indirectly by testing their instances. Each instance of the same class is identical, provided that no superclasses change and generics (i.e., parameterized classes) are not involved. Therefore completely unit testing a single instance completely tests its corresponding class. When generic classes are used, each new set of generic parameters supplied to the class results in a different instance. Thus, although all of the instances of a generic typically share some code that may only need to be tested once, that common code may be in a new environment and the entire object may therefore need to be unit tested. When generics are involved, the class may never be considered fully tested. When inheritance is involved (i.e., when the class to be tested inherits some of its resources from one or more super classes), then changes to superclasses may have unexpected effects on the class to be tested. Changes to superclasses may therefore obsolete test results and require significant regression testing, especially if configuration management and detailed analysis does not rule out impacts on the class to be tested.

The primary purpose of classes and inheritance hierarchies is reuse. For this reason, initial testing should be exhaustive and include stress testing. Regression testing should be the norm. Reuse repositories should store not only source code for the classes (and objects) but also test plans, procedures, drivers, stubs, and test cases. Testing should concentration not only on new features introduced by subclasses, but also include regression testing of previously tested resources from the superclasses. Testing should also look for errors resulting from the synergistic interaction of new and old resources. Because it is not always easy for human developers to determine what resources are being inherited from what superclasses (especially if the inheritance hierarchies are large or if multiple inheritance is used), it is typically advisable to flatten the hierarchy and compare the resulting class to the requirements of the actual subclass when doing test planning and development. Whereas the testing of objects is typically application-

specific, the testing of classes should be more general because the developer of a class cannot know in advance how the instances of that class may be used on future projects.

Testers should test for the following errors associated with classes:

- Abstraction violated
- Documentation out of sync with the code
- Incorrect state model
- Invariants violated
- Failures associated with instantiation and destruction
- Failures associated with inheritance (see paragraph 3.7 below)
- Failure to meet required behavior of the class
- Failures associated with messages, exceptions, attributes, or operations (see paragraphs 3.3 through 3.6 below)

3.3) Testing Messages

Objects and classes primarily interact via messages, although they may also interact via the visibility of attribute types (if the language is typed) and the raising of exceptions. Messages may be used for the following three distinct purposes:

1) To request a corresponding service (which may or may not be implemented, by a single, associated operation).
2) To provide notification that a specific event has occurred.
3) To provide data (in the form of actual parameters of the message).

The sender and the receiver of the message have numerous responsibilities to be tested. The tester should test for errors of the sender to:

- Send the message to the right receiver(s)
- Determine the right type of message (e.g., sequential, synchronous, asynchronous)
- Assign the right priority to the message
- Only send messages that are declared in the specification of the receiver(s)
- Supply compatible actual parameters in the message for all formal parameters required by the corresponding message exported by the receiver

Testers should also test for any errors of the receiver's responsibility to:

- Check the compatibility of all actual parameters with the corresponding formal parameters
- Assign the right priority to each message queue
- Ensure that the correct operation receives the message
- Correctly perform the associated operation if possible
- Ensure that its abstraction is not violated by the performance of the operation by ensuring that all preconditions, postconditions and invariants of the corresponding operation have been met
- Return to the sender all required actual parameters of mode inout or out
- If the operation could not be correctly and safely performed:
  - Raise an appropriate visible exception to the sender
  - Either return to the state it had prior to receiving the message or transition to an appropriate error state

Testers should consider using the following types of blackbox protocol tests of messages:

- Equivalence class testing
- Boundary value testing
- Object instantiation and destruction
- Generic class instantiation

3.4) Testing Exceptions

Objects are not always able to respond as expected to messages. Objects interacting with and modeling entities in their environment (e.g., sensors, actuators) may have to deal with hardware failures. Objects may need to detect inappropriate data supplied as parameters of messages, especially if the implementation language is not strongly typed and such errors are neither prevented nor detected at compile time. The abstraction of an object may be violated (e.g., if it is not designed in a robust manner). Various exceptions must therefore be raised from the server object to its clients and the clients must properly handle such exceptions. Exceptions may be developer-defined and raised (e.g., failure of the modeled hardware) or language-defined and raised by the run-time system (e.g., constraint error).

Testers should test for the following errors associated with exceptions:

- Unit testing:
  - Failure to use exception handling
  - Failure to raise an exception under proper circumstances
  - Raising of an exception under improper circumstances
  - Failure to handle an exception
  - Improper handling of an exception
  - Failure to meet unit-level requirements regarding exceptions
- Integration testing:
  - Improper passing of exception from server to client
  - Exceptions propagating out of scope
  - Failure to meet subassembly-level requirements regarding exceptions
- Acceptance testing:
  - Failure to meet assembly-level requirements regarding exceptions

3.5) Testing Attributes

All data are encapsulated as attributes within the bodies of objects and classes; there is no common global data. Because attributes are hidden, the testing of attributes is primarily done during white box testing, although blackbox and integration testing are also used. Attributes can be either constants or variables and are used:

- To describe objects
- To store state information and support the implementation of objects as state machines
- In assertions:
  - Invariants of objects, classes, and operations
  - Preconditions and postconditions of operations
- To implement simple associations (via pointers to other objects)
- To store cardinality of classes and aggregate objects
- Occasionally to store secondary identifiers (similar to keys in relational databases)

Certain types of bugs due to the unexpected update of common global data do not occur in object-oriented software. However, other bugs (e.g., the reading of attributes before their initialization) remain. To test whether attributes have their proper values, debuggers must be used or the values of the attributes must be exported by operations accessible via messages because the attributes themselves are not directly visible. Do not use messages to export attributes merely for testing sake as this would either violate information hiding (if these messages are available in the delivered software) or would result in different software being delivered than was tested (if the messages were removed after testing).

Testers should test for the following errors associated with attributes:

- Unit testing:
  - Not initialized before being read
  - Out of range (note that the proper use of strongly typed languages may make such bugs impossible)
  - Unreachable states
  - Inappropriate state transitions
  - Invariant relationships between attributes violated (e.g., for a rectangle object, the area attribute should equal the product of the length and width attributes)
  - Failure of preconditions and postconditions involving attributes
  - Values of attributes unnecessarily exported via operations
  - Failure to meet unit-level requirements regarding attributes
- Integration testing:
  - Attributes unnecessarily exported (possible with hybrid language)
  - Lack of protection by mutual exclusion from corruption due to concurrent access
  - Failure to meet subassembly-level requirements regarding attributes
- Acceptance testing:
  - Failure to meet assembly-level requirements regarding attributes

3.6) Testing Operations

All operations (a.k.a., methods, services, member functions) are encapsulated in objects and classes and are only accessible via messages; there are no common global operations or utilities in pure object-oriented applications. Operations, therefore, cannot typically be tested in isolation. Operations only make sense in the context of their objects and classes, and their behavior is dependent on the attributes with which they interact and the exceptions that they raise and handle. The order of execution of the operations is often significant because objects and classes often have state and their behavior depend on their state. Operations often have preconditions, postconditions, and invariants that must be met. Although sequential and hybrid languages may export operations, most operations are hidden and are only accessible via debuggers and messages.

Testers should test for the following errors associated with operations:

- Unit testing:
  - Message not received by correct operation or message received by incorrect operation
  - Precondition, postcondition, or invariants violated (e.g., operation executed while in an incompatible state.
  - Operation incorrectly performed

  - Unreachable statements
  - Proper value not returned or improper value returned by operation
  - Values of attributes unnecessarily exported via operations
  - Correct exception not raised by correct operation
  - Correct exception not handled by correct operation
  - Improper state following raising of exception
  - Failure to meet unit-level requirements regarding operations
- Integration testing:
  - Improper priority of operation
  - Corruption of attributes due to lack of critical regions and concurrent access
  - Failure to meet subassembly-level requirements regarding operations
- Acceptance testing:
  - Failure to meet assembly-level requirements regarding operations

Testers should consider using basis path testing based on McCabes cyclometric complexity.

3.7) Testing Classification and Inheritance

Classification and inheritance are both fundamental aspects of object-oriented software. Objects are typically instances of classes, and classification concerns this "has class" or "is a" relationship between instances and their templates (i.e., classes). Inheritance is the relationship between new classes (i.e., subclasses) that are built from existing classes (i.e., superclasses) whereby the subclasses inherit the resources (e.g., attributes, operations) from their superclasses. Concrete classes can be used to instantiate objects, whereas abstract classes are incomplete and are used as foundations on which to build subclasses. Some abstract superclasses declare deferred resources that are to be supplied by their subclasses and which constrain the structure of their subclasses. Multiple inheritance occurs when subclass inherits from more than one superclass. When inheritance hierarchies are large or when different variants of the same resource may be multiply inherited from two different superclasses, it is often difficult

Testers should test for the following errors associated with inheritance and classification:

- Integration testing:
  - Abstract class incorrectly instantiated
  - Deferred resource not provided by subclass of deferred superclass
  - Wrong resource (e.g., attribute type, attribute, operation) inherited (e.g., due to confusion resulting from multiple inheritance)
  - Original resource in superclass not overwritten in subclass
  - Original resource in superclass not deleted in subclass
  - Generic classes improperly instantiated (e.g., with incorrect parameters)
  - Failure to meet requirements regarding inheritance (e.g., involving taxonomies)
- Regression testing:
  - Unexpected changes in subclass due to changes in superclass

3.8) Testing Scenarios

Objects and classes are not islands; they collaborate by sending messages, raising and handling exceptions, and providing visibility to attribute types (when strongly typed

languages are used). Several objects working together may be necessary to fulfill a single customer requirement, and scenarios of object interaction are often the optimum way to specify requirements and organize integration testing.

During integration testing, testers should test for the following errors associated with scenarios:

- Failure to meet required behavior of the scenario
- Correct message passed to the wrong object
- Incorrect message passed to the right object
- Correct exception raised to the wrong object
- Incorrect exception raised to the right object
- Concurrency problems such as:
  - Unnecessary polling
  - Starvation
  - Deadlock
  - Priority inversion
- Inadequate performance and missed deadlines

4) Testing during Object-Oriented Development (OOD)

Object-oriented software is typically not developed according to the classic (and now obsolete) waterfall development cycle. Rather, it is often developed in small increments (a.k.a., subassemblies, subsystems, clusters, subjects) according to "analyze a little, design a little, code a little, test a little "development cycles involving significant iteration and often based on recursion.

On projects using a recursive development cycle, non-terminal subassemblies contain objects or classes that must be temporarily stubbed out because they must send messages to as yet unidentified objects and classes in lower-level child subassemblies. On such projects, software is typically integrated and initially tested top-down by subassembly within the assembly and bottom-up (due to compilation order restrictions) by object and class within the subassembly. Final unit and integration testing then occurs top-down by subassembly as the stubs are filled in during the development of the lower-level child subassemblies.

Reuse is a major consideration, especially on subsequent projects, as new subclasses are derived via inheritance from existing superclasses. Regression testing and the reuse of test plans, procedures, software, and test cases become critically important if testing is avoid unnecessarily reinventing wheels.

Therefore, object-oriented development has major impacts on testing. It affects the order in which the software is tested, whether testing is incremental or big-bang, whether testing is top-down or bottom-up, and the relative emphasis on regression testing and the reuse of testing as well as code.

5) Conclusion

Whereas traditional procedural software consists of functional modules and common global data connected by subroutine classes and data flow, object-oriented software

consists of objects and classes connected by message passing and inheritance. Whereas traditional software is typically developed according to the waterfall development cycle, object-oriented software is most often incrementally developed according to a recursive, iterative development cycle. These differences result in major differences in the type and scheduling of both unit and integration testing. The categories and probabilities of errors also changes, with certain categories of bugs (e.g., those associated with common global data) disappearing completely. What was formally integration testing (i.e., the testing of multiple operations) now becomes unit testing (i.e., the testing of individual objects). Integration testing now consists of the testing of multiple objects and classes collaborating via message passing and exception raising.

Early books and articles on object technology concentrated on object-oriented programming languages and associated programming styles. More recent publications have addressed language-independent design and requirements analysis. Books on object-oriented systems analysis, systems design, domain analysis, and enterprise modeling are currently appearing. What is unfortunately missing is widespread research, publication, and training on object-oriented testing (and integration).

Bibliography of Books and Articles addressing Object-Oriented Testing

Donald G. Firesmith, *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach*, John Wiley and Sons, New York, NY, 1993.

Ivar Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, Reading, MA, 1992.

DeWayne E. Perry and Gail E. Kaiser, "Adequate testing and object-oriented programming," *Journal of Object-Oriented Programming*, 2(5):13-19, January/February 1990.

M. D. Smith and D. J. Robson, "A framework for testing object-oriented programs," *Journal of Object-Oriented Programming*, 5(3):45-53, June 1992.

About the Author

Donald G. Firesmith is President of Advanced Software Technology Specialists (ASTS) and Director of Object Technology at Software Consulting Specialists (SCS). He is the author of *Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach* and is currently writing Object-Oriented Development Methods, Standards, and Procedures. He has developed the object-oriented software development method, ASTS Development Method 3 (ADM3), which was designed for large, complex, real-time systems and which is currently supported by the CASE tools ObjectMaker and Paradigm Plus. He is also the developer of the OOSDL object-oriented specification and design language. He provides consulting, training, and independent verification and validation (IV&V) in the areas of object technology and Ada.

Prior to founding ASTS in 1988, he was the Division-Level Software Methodologist for Magnavox Electronic Systems Company. There, he developed the project OOD Manual

and ran an OOD help desk for the Advanced Field Artillery Tactical Data Systems (AFATDS) project that generated approximately 1.2 million non-comment, non-blank lines of object-based Ada software. He was the founding chairman of the SIGAda Software Development Standards and Ada Working Group (SDSAWG) which represented the Ada Community during the formal government and industry review of DOD-STD-2167A. He was also an Electronic Industry Associations (EIA) representative on the Council of Defense and Space Industries Association (CODSIA) Software Development Standards Task Force. He received a personal commendation in 1988 from Major General David J. Teal, USAF for "his outstanding contribution to the Joint Logistics Commanders Computer Resource Management Group in support of the development of Defense System Software Development Standard, DOD-STD-2167A."