SYSTEMS DEVELOPMENT MANAGEMENT

TESTING OBJECT-ORIENTED SOFTWARE

Donald G. Firesmith

INSIDE

Testing Classes, Messages, Exceptions, Attributes, Operations, Classification and Inheritance, Scenarios, Aggregates, Testing During Object-Oriented Development, Testing Languages

INTRODUCTION

Testers who are very familiar with bugs associated with such procedural features as common global data and functional decomposition might not know as well the errors common in object-oriented software. This article defines some major features of object-oriented software (e.g., objects and messages), errors associated with these features, and testing techniques for finding these errors.

TESTING CLASSES

Classes are templates for the instantiation of objects. They must define all resources of their instances, including acceptable messages, exceptions, attribute types, attributes, and operations. Classes may also have class attributes (e.g., number of instances constructed), class messages (e.g., construct an instance), class operations, and class exceptions. Although classes, like objects, interact through message passing, they are primarily coupled through inheritance relationships (i.e., subclasses inherit part of their definition from their superclasses). Because classes are intended for massive reuse, quality is critical and testing should be thorough.

In many object-oriented programming languages, classes are compile-time entities used to create run-time objects. Classes in such

PAYOFF IDEA

Benefits of object-oriented programming languages can be lost when errors are introduced during the development process. Testing is one of several techniques to identify these errors; by executing the software with the explicit intention of uncovering errors that cause failures, object-oriented testing ensures that completed software exhibits, correctness, efficiency, interoperability, portability, reliability, and robustness. This article discusses testing classes, messages, exceptions, and classification and inheritance, and aggregates, among other components native to object-oriented languages. It also describes testing specific languages, including C++, Smalltalk, and Eiffel.

Auerbach Publications

languages do not execute and may not be tested directly. Thus, classes are often tested indirectly by testing their instances. If no superclasses change and no generics (i.e., parameterized classes) are involved, each instance of the same class is identical. Therefore, completely unit testing a single instance completely unit tests its corresponding class. When generic (i.e., parameterized) classes are used, each new set of generic parameters supplied to the class results in a different instance. Although all instances of a generic class typically share some code that may need to be tested only once, this common code may be in a new environment, and the entire instance may therefore need to be unit tested. In fact, when generic parameters are involved, the class may never be considered fully tested.

When the class to be tested inherits some of its resources from one or more superclasses, changes to superclasses may have unexpected effects on the class to be tested. Changes to superclasses can make test results obsolete and require significant regression testing, especially if configuration management and detailed analysis indicate impacts on the class being 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 also be conducted. Reuse repositories should store not only source code for classes, objects, and generic parameters but also test plans, procedures, drivers, stubs, and test cases as well as analysis and design information. The testing of a subclass should involve not only the new features introduced by the subclass but previously tested software inherited from the superclasses; unexpected errors can result in the interaction between the new subclass and the preexisting software.

When testing and development are being planned, it is advisable to flatten the hierarchy and trace the requirements to the resulting class, because it is not always easy to determine which resources are inherited from which superclasses. The testing of objects is usually application specific [see "Testing in Object-Oriented Versus Procedural Environments" (34-70-50)], but the testing of classes should be more general because the developer of a class cannot know in advance how instances of that class may be used on future projects.

Exhibit 1 lists errors associated with classes, the priority of these errors, and the testing techniques and tools used to detect these errors.

TESTING MESSAGES

Objects and classes interact by means of messages, though they may also interact through visible attribute types and the raising of exceptions. Messages may be used for the following three distinct purposes:

EXHIBIT 1—Common Errors and Testing Techniques for Classes

Errors Associated with Classes	Priority	Testing Techniques and Tools
Failure to meet the allocated requirements	Critical	Black-box testing
Abstraction violated	Critical	Black-box and white-box testing of assertions and exceptions, and stress testing using multiple messages
Incorrect state model	Critical	Inspection, black-box testing, and white-box testing of preconditions and postconditions
Invariants violated	Critical	White-box testing of assertions
Failures associated with inheritance	Critical	Integration testing
Failures associated with both instantiation (e.g., incompatible generic parameters) and destruction (e.g., problems with storage reclamation and dangling pointers)	Critical	Black-box testing
Inadequate achievement of software engineering goals or inadequate use of software engineering principles	High	Inspection and black-box testing (typically verified at the class level)
Failures associated with messages, exceptions, attributes, or operations	High	See sections on testing these features
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate the requirements of the class	Medium	Inspection
Associations not implemented by messages or attributes	Medium	Inspection and black-box testing (e.g., error guessing)
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous classes

- To request a corresponding service, which may be implemented by one or more operations.
- To provide notification that a specific event has occurred.
- To provide data in the form of actual parameters of the message.

Testers should consider using the following types of black-box protocol tests for messages:

- Equivalence-class and boundary-value testing based on combinations of messages and message parameters.
- Object instantiation and destruction.
- Generic class instantiation.

The sender and the receiver of a message have numerous and different responsibilities. These should determine whether these responsibilities have been met. Exhibit 2 lists the sender operations that should be tested, and Exhibit 3 lists the receiver responsibilities that should be met.

TESTING EXCEPTIONS

Objects cannot always respond to messages as expected. For example, objects interacting with and modeling entities in their environment (e.g., sensors, actuators) may have to deal with hardware failures, or objects may have to detect inappropriate data supplied as parameters. 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). Exhibit 4 lists the errors associated with testing exceptions as well as the priority of these errors and the techniques used to test them.

TESTING ATTRIBUTES

With object-oriented development, data is encapsulated in objects and classes as attributes; there is no common global data. Because attributes are encapsulated, the testing of attributes is performed primarily with white-box testing techniques. Attributes can be either constants or variables and are used for these purposes:

- To describe objects.
- To store state information and support the implementation of objects as state machines.
- To be used in assertions as:
  —Invariants of objects, classes, and operations.
  —Preconditions and postconditions of operations.
- To implement simple associations with other objects.
- To store data on the cardinality of classes and aggregate objects.
- To store secondary identifiers, which are similar to keys in relational data bases.

EXHIBIT 2—Common Errors and Testing Techniques for Message Senders

Errors Associated with Message Senders	Priority	Testing Techniques and Tools
Failure to meet the allocated requirements	Critical	Integration testing
Message sent to wrong receiver	Critical	Integration testing
Wrong type of message (e.g., sequential, synchronous, asynchronous)	High	Inspection and integration testing
Incorrect message priority	High	Inspection and integration testing
Incompatible actual parameters in the message for all formal parameters required by the corresponding message exported by the receiver	Medium to critical	Inspection, compilation, and integration testing
Message not declared in the interface (i.e., protocol) of the receiver	Medium to high	Inspection, compilation, and integration testing
Association not implemented by message	Medium	Inspection and black-box testing (e.g., error guessing)
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the message	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

EXHIBIT 3—Common Errors and Testing Techniques for Message Receivers

Errors Associated with Message Receivers	Priority	Testing Techniques and Tools
Associated operation incorrectly performed	Critical	Black-box and white-box unit testing
Abstraction violated by the performance of the associated operation	Critical	Assertions and white-box unit testing
Incorrect operation received the message	High	White-box unit testing
Required actual parameters of mode in or out not returned to the sender	High	Black-box unit testing
Incorrect message queue priority	High	Inspection and white-box unit testing
Receiver fails to raise an appropriate visible exception to the sender when a requested service could not be correctly and safely provided	Medium	Black-box unit testing
Receiver fails to either return to the state it had before receiving the message or fails to transition to an appropriate error state when a requested service could not be correctly and safely provided	Medium	White-box unit testing
Failure to check the compatibility of actual parameters in the message with all formal parameters required by the corresponding message exported by the receiver	Medium	Inspection, compilation, and black-box unit testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the message	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

EXHIBIT 4—Common Errors and Testing Techniques for Exceptions

Errors Associated with Exceptions	Priority	Testing Techniques and Tools
Failure to meet requirements regarding exceptions	Critical	Inspection and testing
Failure to correctly handle and exception	Critical	Inspection and white-box unit testing
Exceptions propagating out of scope	Critical	Integration testing using a debugger
Failure to use exception handling	High	Inspection and white-box unit testing
Improperly raising an exception from server to client	High	White-box unit testing and integration testing
Failure to raise an exception under proper circumstances	High	Inspection and white-box unit testing
Raising of an exception under improper circumstances	Medium	Inspection and white-box unit testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the exception	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

Testers must check for such actions as the reading of attributes before their initialization. To test whether attributes have their proper values, debuggers must be used, or the attributes' values must be exported because the attributes themselves are encapsulated. Messages should not be placed in the software to export attributes merely for testing purposes. Information hiding would be violated if these messages were available in the delivered software, and removing the messages after testing means that the delivered software is different from the software that was tested.

Exhibit 5 lists errors associated with attributes, the priority of these errors, and the testing techniques and tools used to detect these errors.

TESTING OPERATIONS

All operations—methods, services, and member functions—are encapsulated in objects and classes and can be accessed only through messages; there are no common global operations or utilities in pure object-oriented applications. Operations, therefore, cannot typically be tested in isolation, because they have meaning only in the context of their objects and classes, and their behavior depends on the attributes with which they interact and the exceptions that they raise and handle. The order of execution of the operations is significant because objects and classes often have a state, and their behavior depends 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 accessible only through debuggers and messages.

Testers may consider using basis-path (i.e., structured) testing that is based on McCabe's cyclometric complexity metric for the testing of individual operations. This metric must be adapted for object-oriented code so it can reflect such object-oriented features as polymorphism, dynamic binding, and interactions with the object's state. However, the number of basis paths within a single operation is often trivial because of the small size of the operations (e.g., typically 3 to 15 statements, depending on the language).

Exhibit 6 lists common errors associated with operations, the priority of these errors, and the techniques and tools for detecting these errors.

TESTING CLASSIFICATION AND INHERITANCE

Classification and inheritance are both fundamental concepts of object-oriented software. Testers must understand these concepts to know how they affect the testing process. The following paragraphs provide insight into these concepts.

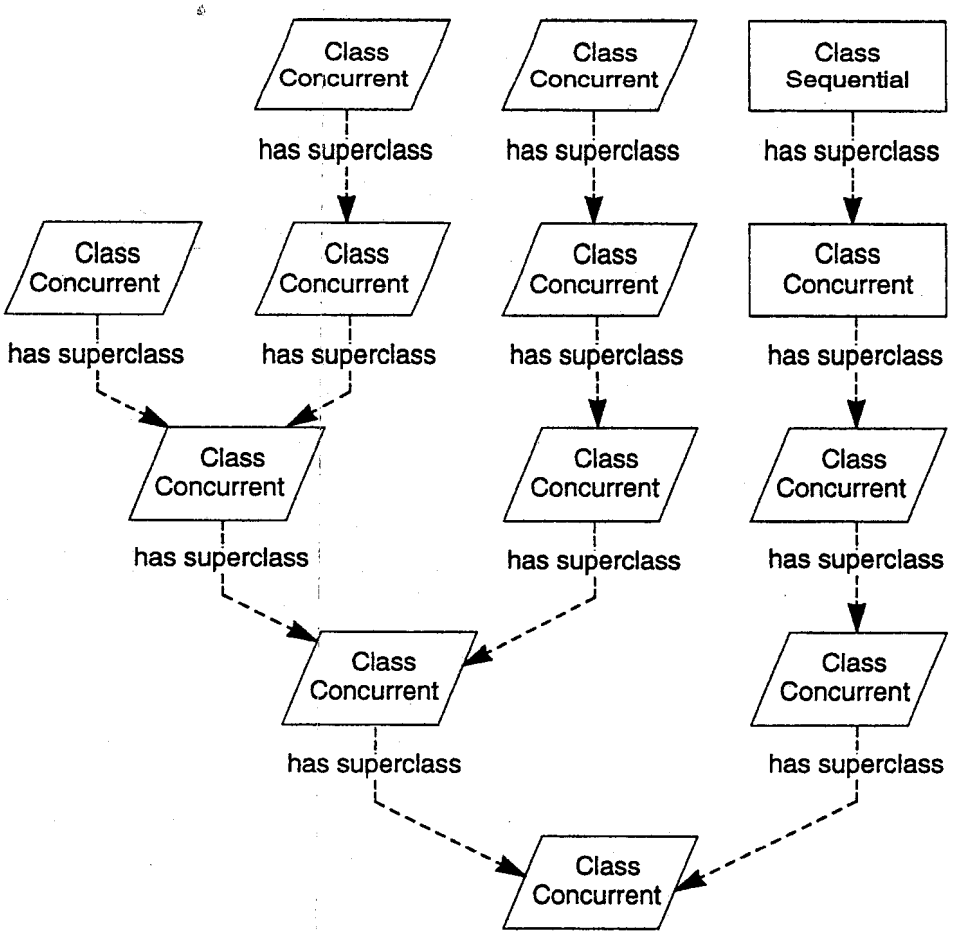
EXHIBIT 5—Common Errors and Testing Techniques for Attributes

Errors Associated with Attributes	Priority	Testing Techniques and Tools
Failure to meet requirements regarding attributes	Critical	Inspection and black-box unit testing
Invariant relationships between attributes violated (e.g., for a rectangle object, the area attribute should equal the product of the length and width attributes)	Critical	White-box unit testing of assertions
Failure of preconditions and postconditions of operations involving attributes	Critical	White-box unit testing of assertions
Incorrect values of attributes exported by messages	Critical	Black-box unit testing
Not initialized before being read	Critical	Inspection and debugger during unit testing
Out of range	Critical	Inspection and debugger during unit testing
Encapsulated in wrong object	Critical	Inspection
Unreachable states	High	Inspection and debugger during unit testing
Inappropriate state transitions	High	Debugger during unit testing
Attributes unnecessarily exported	High	Inspection and debugger during integration testing
Lack-of protection by mutual exclusion from corruption due to concurrent access	High	Inspection and debugger during integration testing
Incorrect or mission unit of measure	High	Inspection
Incorrect accuracy or precision	High	Inspection and debugger during unit testing
Inappropriate, inadequate, or incorrect update frequency	High	Inspection and debugger during unit testing
Values of attributes unnecessarily exported by operations	Medium	White-box unit testing of assertions
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the attribute	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

EXHIBIT 6—Common Errors and Testing Techniques for Operations

Errors Associated with Operations	Priority	Testing Techniques and Tools
Failure to meet requirements regarding operations	Critical	White-box unit testing
Message not received by correct operation or message received by incorrect operation	Critical	White-box unit testing or compilation (if language requires one-to-one mapping)
Precondition, postcondition, or invariants violated (e.g., operation executed in an incompatible state)	Critical	Assertions and white-box unit testing
Operation incorrectly performed	Critical	White-box unit testing
Corruption of attributes due to lack-of critical regions and concurrent access	Critical	Integration testing
Inaccurate timing or missed deadline	Critical	White-box unit testing using a performance analyzer
Proper value not returned or improper value returned by operation	High	White-box unit testing
Correct exception not raised by correct operation	High	White-box unit testing
Correct exception not handled by correct operation	High	White-box unit testing
Improper state following raising of exception	High	White-box unit testing
Improper priority of operation	High	Integration testing
Unreachable statements (e.g., dead code)	Medium	White-box unit testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the operation	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

EXHIBIT 7—An Object-Oriented Inheritance Diagram



Objects are almost always instances of classes, and classification concerns the has-class or is-a relationship between instances and their classes. Inheritance concerns the has-superclass (see Exhibit 7) or a-kind-of relationship among subclasses that inherit the resources from their superclasses. Subclasses may add, modify, or delete requirements from their superclasses.

Testers of object-oriented software should be concerned with the following classes:

- *Concrete Class.* This is used to instantiate objects and can be tested in terms of their instances.
- *Abstract Class.* This incomplete class forms the foundation on which subclasses are built. Abstract classes are indirectly tested through their concrete subclasses.
- *Deferred Class.* This is an abstract class. A deferred class declares resources that are supplied by their subclasses and that constrain the structure of their subclasses.

- *Generic Class.* This is a parameterized abstract class that requires the user to supply parameters before instantiation.

Multiple inheritance occurs when a subclass inherits from more than one superclass. Multiple inheritance increases the potential for confusion because different versions of the same resources may be inherited from different superclasses. Different compilers and methodologies use different conflict resolution techniques to clear up this confusion. Multiple inheritance and large inheritance hierarchies can also make it difficult to determine from where and what a subclass inherits.

Different classes in the same inheritance hierarchy may contain analogous resources (e.g., messages, operations, or attributes). Because of this polymorphism, testers should expect implementations different from those that are inherited. In addition, the implementation of these polymorphic resources may not be known until run-time, which makes white-box testing difficult.

Inheritance is contrary to the principle of localization because the entire definition of a subclass is found in multiple locations (i.e., the subclass and its direct and indirect superclasses). Inheritance is also contrary to the principle of encapsulation because subclasses have access to the encapsulated resources of their superclasses. Because inheritance features violate the software engineering principles of localization and encapsulation, they are a source of potential errors.

It has not been determined how much regression testing is considered adequate for testing the inherited resources of a subclass. A major benefit of inheritance is that it enables subclasses to reuse existing classes. If the superclasses have been satisfactorily tested, resources inherited from them may not require extensive regression testing. However, new resources added by subclasses may invalidate the soundness of inherited resources because of unexpected interactions through common local attributes. Subclasses may also accidentally delete inherited resources required by their abstract superclasses. For these reasons, testers should perform the following:

- Testing the new resources introduced by the subclass.
- Regression testing any inherited resources that interact with these new resources.
- Testing the integration of all resources, both new and inherited, during the black-box and white-box testing of the flattened subclass. A flattened subclass is simulated by creating a class containing all the resources the subclass can inherit.

Exhibit 8 lists errors associated with classification and inheritance, the priority of these errors, and the testing techniques and tools used to detect these errors.

EXHIBIT 8—Common Errors and Testing Techniques for Inheritance and Classification

Errors Associated with Inheritance and Classification	Priority	Testing Techniques and Tools
Failure to meet requirements regarding inheritance	Critical	Integration testing
Abstract class instantiated	Critical	Integration testing and compilation
Unexpected changes in subclass due to changes in superclass	Critical	Regression testing
Deletion of resource by subclass violates abstraction of superclass	Critical	Inspection and integration testing
Generic classes improperly instantiated	Critical	Integration testing
Wrong resource inherited	Critical	Integration testing
Original resource in superclass not overwritten in subclass	High	Integration testing
Original resource in superclass not deleted in subclass	High	Integration testing
Ad hoc support for dynamic classification fails	High	Integration testing
Deferred resource not provided by subclass of deferred superclass	High	Integration testing
Incompatibility between subclass resources and existing superclass resources	High	Regression testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the operation	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (usually verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

TESTING SCENARIOS

Objects and classes do not operate in isolation; they operate together by sending messages, raising and handling exceptions, and providing in strongly typed languages visibility to attribute types. Several objects may have to operate together to fulfill a single requirement, and scenarios (i.e., use cases) of objects are often the optimum way to specify requirements and organize integration testing.

Scenarios can be tested during unit testing, integration testing, and software system testing. Scenarios involving attributes and black-box operations are unit tested, scenarios involving black-box objects and classes are integration tested, and scenarios involving black-box systems and devices are checked during systems integration testing and software system testing. Exhibit 9 lists common errors associated with scenarios, the priority of each error, and the techniques and tools for finding these errors.

TESTING AGGREGATES

Objects may be aggregates of component objects. Component objects may be exported, or they may be encapsulated in an aggregate object. Encapsulated component objects can be accessed only indirectly through messages to the aggregate objects that contain them. The state of the aggregate object may be determined by the states of its component objects. The component objects must be unit tested, and the aggregate object must undergo integration testing to find errors and inconsistencies in the integration of the components to form the aggregate. Exhibit 10 lists common errors associated with aggregates, the priority of each error, and the techniques and tools for finding these errors.

TESTING DURING OBJECT-ORIENTED DEVELOPMENT

Object-oriented software is typically not developed according to the phases of the traditional waterfall development cycle. Instead, it is often recursively developed in small increments (i.e., subassemblies, subsystems, clusters, and subjects) that involve significant iteration and parallel development. The scheduling of testing during object-oriented development should take into account the fact that errors cost more to fix the longer they go undetected. The recursive nature of object-oriented development and the small increment size (e.g., 3 to 10 collaborating objects or classes) means that errors can be identified more quickly after they have been made and that requirements and designs can be incrementally validated as they are developed.

In projects using a top-down recursive development cycle, nonterminal subassemblies contain objects or classes that must be temporarily stubbed out, because they must send messages to as-yet unidentified objects and classes in subassemblies (see Exhibit 11). Thus software is

EXHIBIT 9—Common Errors and Testing Techniques for Scenarios

Errors Associated with Scenarios	Priority	Testing Techniques and Tools
Failure to meet requirements of the scenario or subassembly	Critical	Integration testing
Inadequate performance, inaccurate timing, and missed deadlines	Critical	Performance analyzer
Failures associated with instantiation and destruction	Critical	Black-box unit testing
Messages sent to objects that no longer exist	Critical	Integration testing
Incorrect message passed to the right object	Critical	Integration testing
Incorrect exception raised to the right object	Critical	Integration testing and debugger
Correct exception raised to the wrong object	High	Integration testing and debugger
Responsibility for object destruction incompletely or inconsistently allocated	High	Integration testing
Correct message passed to the wrong object	High	Integration testing
Deadlock due to circular dependency from daisy chain of messages	High	Integration testing
Interface incompatibility with non-object-oriented language, operating system, or data base	High	Integration testing
Memory not reclaimed when object destroyed or inadequate memory due to memory leakage	High	Black-box unit testing, integration testing, and memory checker tool
Such concurrency problems as unnecessary polling, starvation, deadlock, priority inversion, and inconsistent priorities	High	Inspection, integration testing, and performance analyzer
C++ friend functions corrupting attributes or implementing inappropriate dependencies on encapsulated resources	High	Inspection and integration testing
Object or class inappropriately exported from subassembly	Medium	Integration testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the scenario or subassembly	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

EXHIBIT 10—Common Errors and Testing Techniques for Aggregates

Errors Associated with Aggregates	Priority	Testing Techniques and Tools
Failure to meet requirements of the aggregate	Critical	Integration testing
Components missing	Critical	Inspection and integration testing
Inconsistent components	Critical	Inspection, assertions, and integration testing
Components not created or initialized when aggregate created or initialized	Critical	Assertion and integration testing
Incorrect visibility of components	High	Inspection and black-box testing of aggregate
Components not destroyed when aggregate destroyed	Medium	Assertion and integration testing
Documentation inconsistent with the code	Medium	Inspection
Failure to allocate requirements of the scenario or subassembly	Medium	Inspection
Violation of design and coding standards	Medium	Inspection, standard checker, and pretty printer (typically verified at the class level)
Syntax errors	Low	Instantiation of class or direct compilation of objects of anonymous class

typically integrated and initially tested top down by subassembly within the assembly and bottom up by object and class within the subassembly. Final unit and integration testing occurs top down by subassembly as the stubs are filled during the development of the subassemblies.

Iteration is critical to the successful development of quality object-oriented software. It is not unusual for classes to be updated and modified several times during the course of the project as requirements change and developers learn more about the requirements. Regression testing and the reuse of test plans, procedures, software, and cases are important to avoid repeating test work that has already been performed.

Reuse is a major consideration, especially on subsequent projects, as new subclasses are derived through inheriting from existing superclasses. Reuse also emphasizes the need for automated regression testing.

Object-oriented development has a major impact on the scheduling of testing and the overlap of the testing activity with other development activities. It affects the sequence of testing operations—in increments or all at once as well as top down or bottom up—and affects the emphasis placed on regression testing and the reuse of testing data, procedures, and code.

LANGUAGE-SPECIFIC TESTING

The following sections discuss points to be considered when testing C++, Smalltalk, and Eiffel, three object-oriented programming languages.

Testing C++

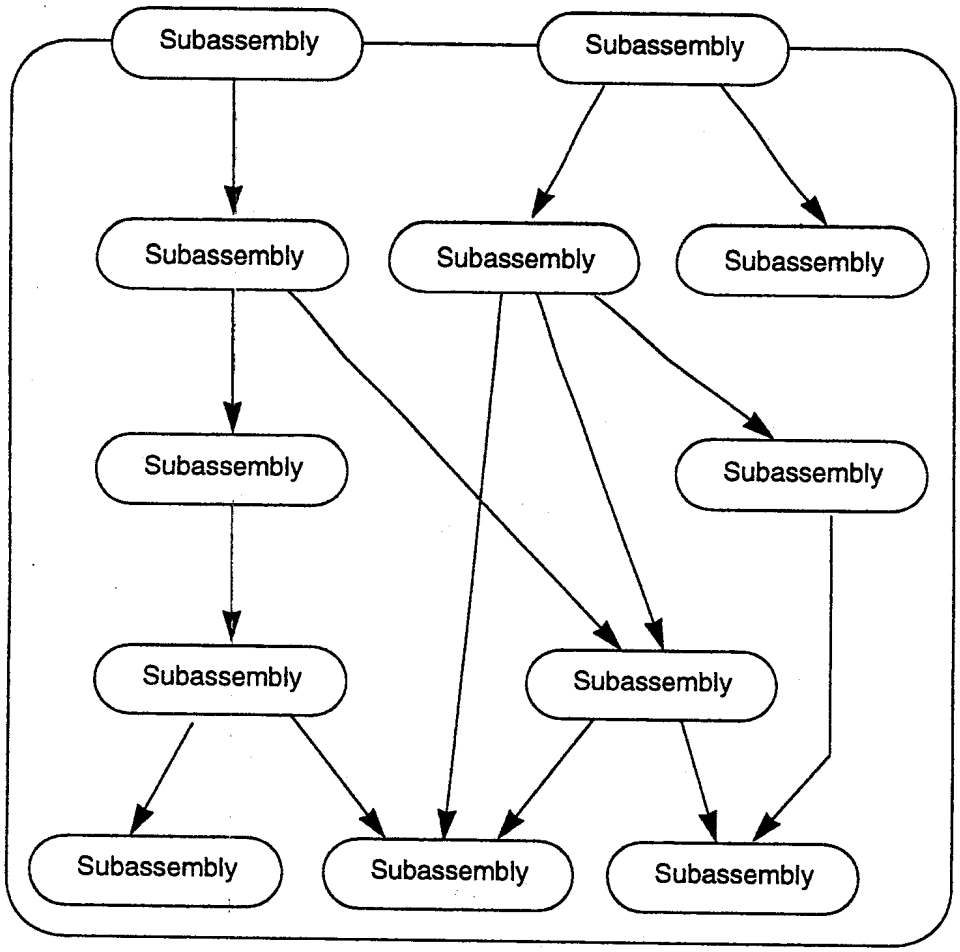
C++ is currently the most popular object-oriented language, especially in the engineering community. It is an improved extension of C that adds strong typing, overloading, and templates (i.e., generics). C++ also supports object-orientation, including both single and multiple inheritance, polymorphism, and static and dynamic binding.

Unfortunately, C++ is often misused and has numerous problems affecting testing. C++ is a complex, hybrid language that emphasizes efficiency and programmer freedom (and consequently demands greater programmer responsibility). It contains many subtle features that are easily overlooked or misunderstood. It is most often used by beginners as merely a better C, with little use of its object-oriented features. It still lacks ANSI standardization, and its class libraries vary from vendor to vendor.

The following guidelines are recommended for avoiding and localizing bugs in C++. The programmer should:

- Use C++'s strong typing to detect errors at compile time.
- Hide type casts and int, float, and void* values inside classes so at least their clients may remain type secure.
- Avoid default parameters that destroy locality because their values are clear in neither the definition nor the language. Default parame-

EXHIBIT 11—An Object-Oriented Assembly Diagram



ters import the default value from a separate context and have scope rules that are often deceptive.

- Be careful using overloading that can cause difficulty in finding typing errors because the rules are subtle and have gone through numerous revisions.
- Always make destructors virtual in the base class and initialize data members (or attributes) before use.
- Use exception handling, if available, rather than return flags.

Debugging C++ is not the same as debugging C. Different C++ compilers generate different information for their debuggers, and some debuggers may understand only C source code. Programmers need a tool that lists the descendants of base classes. Debugging C++ requires dealing with:

- Overload functions.
- Name mangling.

- C++ source code mapping for front implementations.
- The class of the object (when dealing with polymorphic classes).
- Object identities and references rather than values.
- Memory management.

Far more likely to contain bugs than either Smalltalk or Eiffel, C++ is the only major object-oriented programming language with numerous articles and tutorials at conferences discussing common bugs and how to avoid them. Common C++ errors include (but are not limited to) the following:

- Lack of proper and adequate analysis and design.
- C++ merely used as a better C.
- C++ is often unreadable.
- Memory management and garbage collection.
- Pointer errors.
- Initialization and destruction errors:
  —Initialization order errors.
  —Uninitialized data members (or attributes).
- Violations of encapsulation:
  —Improper use of friend functions.
  —Misuse of public versus protected versus private.
  —Improper use of type casts or unencapsulated type casts.
- Inheritance errors:
  —Overuse of implementation inheritance.
  —Member functions were not made virtual in the appropriate ancestor to support polymorphic via dynamic binding.
  —Use of type conversions (or casts) and unions to circumvent type safety.
  —Misuse of pointers and arrays violating type safety.
  —Base class references-derived classes.
- Use of null.
- Asserts not used as preconditions and postconditions.
- Exception handling not used.
- Overloading between public and hidden features may be ambiguous.
- Overloading ignores return types leading to ambiguities.
- References are not objects:
  —No size.
  —No addresses.
- Dominance errors.

Testing Smalltalk

Smalltalk is a pure object-oriented language in which everything is an object, including classes and literals, and in which communication is totally via message passing. Encapsulation support is more limited than in C++. All attributes are hidden, and all methods (or operations) are visible.

Smalltalk only supports single inheritance, and all operations are polymorphic. Unlike C++, Smalltalk provides automatic garbage collection, freeing the programmer from numerous memory management errors.

Like C++, Smalltalk is not yet standardized. There is no ANSI standardization, and class libraries vary from vendor to vendor. Smalltalk has excellent integrated environments that are extremely flexible. The source code for environment tools is accessible to the programmer. Smalltalk is quite popular in the MIS community, because of its graphical user interface applications and rapid prototyping.

Smalltalk contains fewer bugs than C++; its errors include (but are not limited to) the following:

- A lack of analysis and design due to iterative hacking and a prototyping mindset.
- Development of a prototype rather than a production version.
- Run-time typing errors due to a lack of type checking.
  —Identifier may suggest wrong class.
  —Message sent to object with missing method will raise a run-time exception because of no compiler check.
- Exception handler will probably be missing.
- Problems caused by deletion of class, movement of class in inheritance hierarchy, or deletion or modification of a message or operation.

Testing Eiffel

Eiffel is a pure object-oriented specification, design, and programming language. It provides both inheritance and generics. It also provides both static and dynamic (default) binding, with an optimizer deciding which is appropriate. Eiffel is very readable. It supports Meyer's Design by Contract method, which emphasizes correctness, reliability, and robustness via assertions (e.g., preconditions and postconditions on routines and invariants on classes). Eiffel has a very powerful exception handling mechanism tied to its assertions.

Eiffel contains the fewest errors of the three languages covered in this chapter. Errors include (but are not limited to) the following:

- Assertions may:
  —Be missing.
  —Not be used effectively.
- Assertion checking is used only for debugging and is turned off in delivered software, making it less reliable.
- Overuse of implementation inheritance.
- Design by contract may place too many responsibilities on the client, especially in a concurrent environment.

CONCLUSION

Object-oriented software has a structure and mode of operation that are fundamentally different from those of traditional procedural software. This article reviews the major features of object-oriented software (e.g., objects and messages), errors commonly associated with these features, and testing techniques for finding these errors. [A companion article, "Testing in Object-Oriented Versus Procedural Environments" (34-70-50) describes the differences between procedural and object-oriented programming languages and defines classes and errors commonly associated with them.] Priorities for detecting these errors are given so the tester can judge the seriousness of these errors.

In addition to the differences in structure and behavior, object-oriented and procedural software are developed in very different ways. This article also covers development, and how it affects how and when the software is tested.

Donald G. Firesmith is a senior member of the technical staff at Knowledge Systems Corporation in Cary NC.