Object-Oriented Regression Testing

Donald G. Firesmith

Regression testing—the repetition of testing for the purpose of finding new errors in previously tested software—is an absolutely critical requirement if object technology is to achieve its promised benefits. This article discusses the increased need for regression testing due to object technology and stresses the need for regression testing to be automated. It then presents three major techniques for automating regression testing, along with a discussion of their pros and cons.

The Increased Need for Regression Testing

Regression testing is the repetition of testing for the purpose of finding errors that may have been introduced into software after initial testing. Regression testing has many uses; its application is especially critical for object-oriented projects:

- *Reuse* will not occur without developer confidence, which regression testing can both provide and justify.
- Given an *object-oriented development cycle*, it can be used to determine whether an *iteration* has inadvertently introduced bugs into previously tested software or if a new increment invalidates software developed during a previous increment.
- Regression testing can also be used to validate the proper use of *inheritance*, which in turn can be used to support the regression testing effort.

Well-engineered object-oriented software is reusable software, and I do not restrict that statement to the ever growing number of class libraries that one

TIP

Developers will not reuse software they do not trust.

can buy from commercial vendors. As more companies embrace the object paradigm, more managers will recognize that the only way to maximize the goals promised by object technology is through the reuse of *patterns* of classes—especially domain-specific frameworks that will allow managers to achieve truly significant levels of reuse, productivity, and quality. Unless companies join with their competitors to form consortia to produce and share the necessary classes and patterns, they will find that they must develop them in-house. Developers will not reuse software they do not trust, and they often have far too many unfortunate experiences with internally produced software to trust it. The best way to gain that trust is via regression testing, but only if such testing can be made relatively quick and painless via automation.

Object-oriented development is best performed as part of an *incremental, iterative, parallel* development cycle. Because the software is developed incrementally, each new increment depends on and must be integrated with the software that was developed during the previous increments. The original software must not only continue to work, it must do so in a new environment containing classes that were either added or modified by the increment. Similarly, iteration implies making changes to existing software that must then be retested to ensure that the corrections and improvements have not inadvertently introduced bugs into the existing software. The parallel nature of the object-oriented development cycle also has testing ramifications. The developer of a class may not be the one who must later make changes to it or create new software that must collaborate with it. Thus, use of an object-oriented development cycle also increases the need for regression testing.

Perhaps the most important cause of the increased importance of regression testing is inheritance. Classes are intended to be developed incrementally by defining new derived classes in terms of one or more existing base classes. The features of these base classes must continue to function in the environment of the new derived class. Yet, combining correct new features with correct inherited features may cause subtle new bugs. For example, both new and inherited operations may interact via inherited attributes, and this use of "common local data" may cause bugs similar to those caused by common global data in traditional procedural software.* Inherited operations must

*Although the encapsulation of object-oriented software eliminates the all too common bugs due to the common global data of traditional procedural software, it may contain bugs due to both common local data in the form of encapsulated properties and common global objects due to a lack of language-supported building blocks larger than classes, i.e., clusters.[1]

continue to provide the correct functional abstraction, even though they may be overridden in the derived class. As in the Biblical metaphor, building on shifting sands can be dangerous. Luckily, inheritance is a double-edged sword. Not only can it cause a host of bugs,[2] inheritance can also be used for the testing of test software, thereby greatly lessening the retest effort and achieving the same benefits for test software that it has produced for deliverable software.

Automating Regression Testing of Units

TIP

Regression testing should be automated.

Because object technology makes regression testing inevitable, regression testing should be automated using a standard retest technique and test software. Standardizing on an automated regression test technique has many benefits. For example, it

- decreases the test effort, which indirectly produces an increase in software quality;
- allows all developers to reuse some of the same test software, such as test drivers and error logs;
- minimizes training costs and makes it easier for different developers, test personnel, and quality assurance personnel to communicate; and
- formalizes the testing process, making it more likely that adequate testing will be performed and that test results will be captured.

The units of object-oriented software are classes, which should be incrementally developed as part of an inheritance structure. Beginning with the root class, new classes are derived from one or more base classes, and these classes should be incrementally unit tested as they are developed. Because classes are typically used only as templates for objects and are not themselves executable, they are first instantiated and then indirectly tested via one or more of their instantiated objects. Abstract classes are either instantiated for test purposes only or else used to derive concrete classes, which are then tested.

There are three main techniques that can be used to automate the regression testing of classes in an inheritance hierarchy. All three techniques use inheritance to inherit test software and test cases as well as normal software. These techniques, which are discussed in the following sections, include the following:

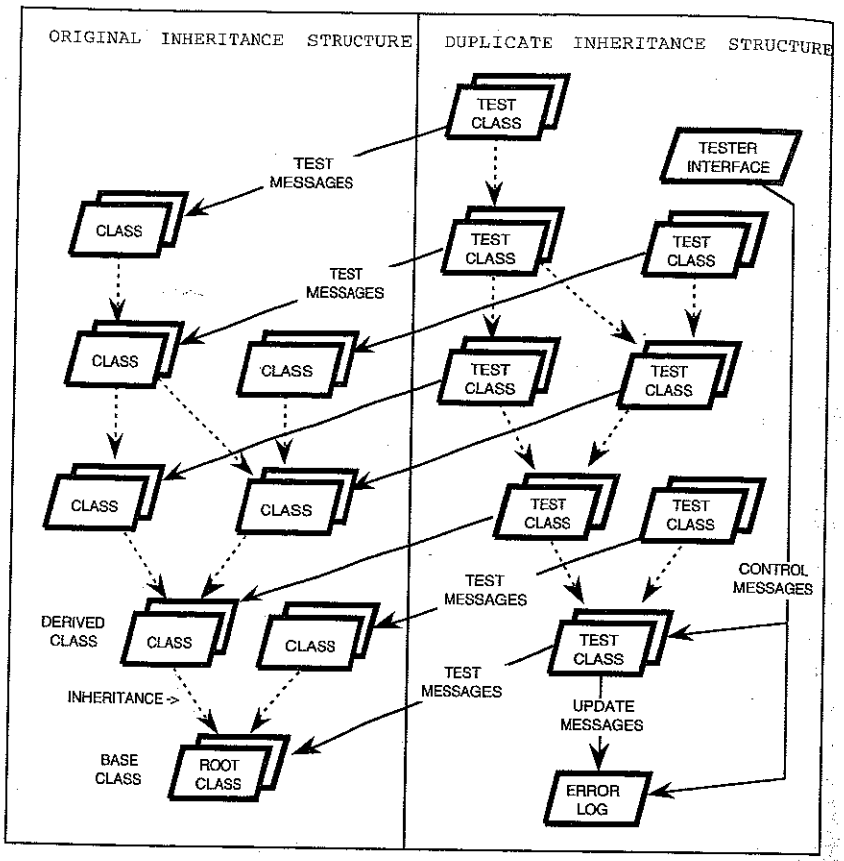

FIGURE 1. *Automated regression testing using duplicate inheritance structure.*

- developing a separate and analogous inheritance structure of test drivers;
- embedding test operations within the classes of the single inheritance structure;
- developing a separate and analogous inheritance structure of mixins that contain embeddable test operations and test cases.

Technique 1: Duplicate Inheritance Structure of Test Drivers

As illustrated in Figure 1, this technique creates a duplicate inheritance structure of test drivers that has the same topology as the inheritance structure to

be tested. As each new class to be tested is derived, a corresponding test-driver class is derived. These test drivers inherit from the root test-driver class the ability to log errors and be driven by a tester interface object.

The main advantage of this approach is that derived classes of test drivers use inheritance to obtain, extend, and modify the test cases of their base classes. Numerous guidelines for the incremental testing of inheritance structures have been developed as part of the Hierarchical Incremental Testing technique.[3]

TIP

Derived classes of test drivers use inheritance to obtain, extend, and modify the test cases.

The main disadvantage of this technique is that it is restricted to black-box testing, and it can be very difficult to test objects from the outside due to their encapsulation and state. The behavior of an object may well depend on its state, which almost always should be hidden from its clients in the form of encapsulated attributes, links, and component subobjects. Objects often do not export operations that allow a test driver to easily place it in the appropriate pretest state and determine if it subsequently attained the appropriate posttest state. The inclusion of *get* and *set* operations in the protocol of a class merely to circumvent encapsulation for testing has several drawbacks. Such operations may be used by clients as trapdoors for other purposes, with obvious detrimental effects. Removing such test operations prior to delivery implies that different code is delivered than was tested. The use of assertions and a debugger significantly reduce these problems, but only if you are willing to give up the benefits of automated regression testing. The use of C++ offers an interesting workaround for this problem. If the classes to be tested declare their associated test classes as "friends," then these test drivers may violate the encapsulation of the objects under test and directly read and set their state.

Technique 2: Embedded Test Operations

To intentionally circumvent encapsulation and enable the easy setting and evaluation of pretest and posttest states, this technique embeds test operations directly within the classes to be tested. As illustrated in Figure 2, this technique embeds standard kinds of test operations in each class and instance. Testing a class involves the following steps:

1. Testing begins when the *tester interface* object sends the *test class* message to the class under test. The test class message will eventually

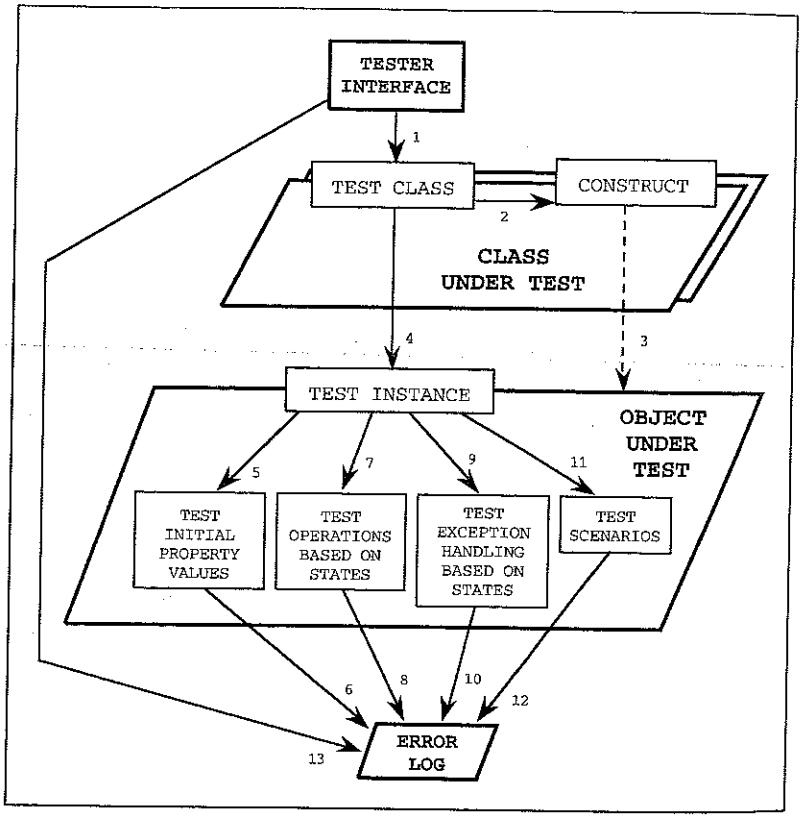

FIGURE 2. *Automated regression testing using encapsulated test operations.*

return the number of errors found to the *tester interface* object.

2. The associated *test class* operation calls the constructor.
3. The constructor constructs one or more instances (e.g., the object under test) that will be used to indirectly test the class under test.
4. The *test class* operation then sends the *test instance* message to the newly constructed object under test. After testing is complete, this message will return the total number of errors found to the *test class* operation.
5. The *test instance* operation then calls the hidden *test initial property values* operation that determines whether the object under test was properly initialized.*

*The hidden test operations may contain the necessary pretest and posttest states, operation parameters, and expected results as local attributes, or this test data may be stored as test attributes of the class.

6. Any errors found are then documented in the *error log* object, and the number of errors is returned to the *test instance* operation.
7. The *test instance* operation then calls the hidden *test operations based on states* operation, which tests each operation in each state by placing the object in the proper pretest state, executing the operation, and checking for the proper result including the proper posttest state.
8. Any errors found are then documented in the *error log* object and the number of errors returned to the *test instance* operation.
9. The *test instance* operation then calls the hidden *test exception handling based on states* operation, which determines if the object under test raises and handles exceptions properly.
10. Any errors found are then documented in the *error log* object, and the number of errors is returned to the *test instance* operation.
11. The *test instance* operation then calls the hidden *test scenarios* operation, which tests each usage scenario of the object. This test operation may not be necessary due to the previous test operations.
12. Any errors found are then documented in the *error log* object, and the number of errors is returned to the *test instance* operation.
13. If the number of errors returned is greater than zero, the *tester interface* object will display the number of errors on the screen and print an error report based on the *error log* object.

The primary advantage of this second technique over the first technique is that it bypasses the encapsulation of objects that make them difficult to test from the outside. The test operations have direct visibility and control over the properties of the objects, which makes the development of the test operations simpler. It also simplifies the interpretation of test results because the properties are local to the test operations. Like the first technique, this technique uses inheritance to minimize the regression test effort. Like the previous workaround using C++ `friends`, this technique allows for the automation of white-box as well as black-box unit tests. By placing the test operations within the classes to be tested, there is no need to produce a duplicate inheritance structure of test drivers. The test operations are also more likely to be used, updated, and kept consistent with the classes under test during the normal iterative incremental development process.

Unfortunately, this technique is not without its disadvantages. It bloats the size of the source code with the test operations, and *manually* removing test code prior to delivery would cause the same problems as mentioned

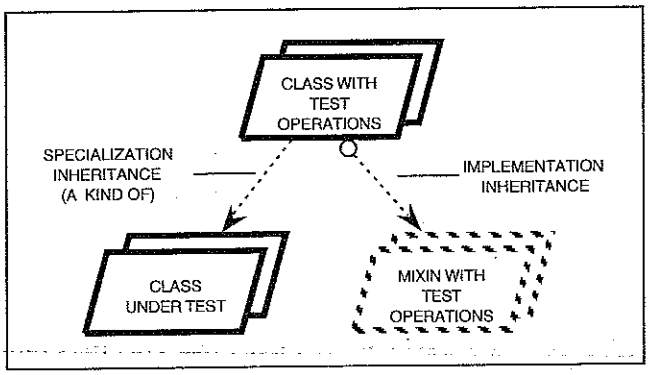

FIGURE 3. *Use of test mixins to combine both inheritance structures.*

TIP

Note the disadvantages of this technique.

previously. This should not be a real problem, however, because a good optimizer should automatically delete the test operations from the deliverable object code as dead code. The proper use of assertions and exception handling supplies many of the benefits of embedded test operations and should be used where practical to minimize the development of test operations and to create more reliable and robust software. Using assertions, however, cannot replace regression testing; assertions cannot be automatically exercised in a test mode and will often find errors only during manual debugging or informally after the software has been delivered.

Technique 3: Duplicate Inheritance Structure of Test Mixins

As illustrated in Figure 3, the third technique is a combination of the first two. Like the first technique, it involves creating a duplicate inheritance structure of test classes. But this time, the test classes are mixins,* each of which contains the test software that would have been embedded in the corresponding class under test using the second technique. By multiply inheriting from both the class to be tested and its corresponding mixin class of test operations, one obtains the same classes that would have been

*A mixin class is defined as an abstract class that embodies a single abstraction that is used to augment the protocol and functionality of other classes by providing common resources via multiple inheritance; the abstraction of a mixin (e.g., persistence) is usually orthogonal to the abstraction of the class(es) with which it is combined.

developed using the second technique. However, the original class without the test operations can be delivered if so desired, and, although it is different from the class that was tested, the only difference is in the missing test operations. The result is the same as if a macro were used to strip out the test operations from the classes developed using the second technique.

CONCLUSION

This column has addressed the critical need for the automated regression testing of classes on projects using object technology. Three useful techniques for accomplishing this were presented. Although each involves a significant amount of effort to set up, much of this effort would have to be performed anyway during initial unit testing, and it clearly pays for itself during subsequent development and iteration. Although the use of friends in the first technique is restricted to C++ and the use of mixins in the third technique is restricted to languages (such as C++ and Eiffel) that support multiple inheritance, the ideas presented in this article can be used on any object-oriented project concerned with quality and with minimizing both the development and maintenance costs.

References

1. Firesmith, D. Clusters of classes: A bigger building block, REPORT ON OBJECT ANALYSIS & DESIGN, 1(4):18–25, 1994.
2. Firesmith, D. Testing Object-Oriented Software, in SOFTWARE ENGINEERING STRATEGIES, 1(5):15–35, 1993.
3. Harrold, M. and J. McGregor. Hierarchical incremental testing, TECHNICAL REPORT TR91-111, Dept. of Computer Science, Clemson Univ., 1991.