8 Testing Object-Oriented Programs

http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/235/chapter8.htm

Object-oriented software differs significantly from traditional procedural software in terms of analysis, design, structure, and development techniques, so specific testing support is also required [FIRESMITH]. The object-oriented language features of encapsulation, polymorphism, and inheritance require special testing support, but also provide opportunities for exploitation by a testing strategy. To adapt structured testing for object-oriented programs, consider both module testing and integration testing. Structured testing at the module level is directly applicable to object-oriented programs, although that fact is not immediately obvious. At the integration level, structured testing does require modification to address the dynamic binding of object-oriented methods. The rest of this section discusses the specific implications of object-oriented programming for structured testing. The majority of this information was previously published in [MCCABE3] and [MCCABE4].

8.1 Benefits and dangers of abstraction

Object-oriented languages and techniques use new and higher levels of abstraction than most traditional languages, particularly via the inheritance mechanism. There are both benefits and dangers to this abstraction. The benefit is that the closer the implementation is to the logical design, the easier it is to understand the intended function of the code. The danger is that the further the implementation is from the machine computation, the harder it is to understand the actual function of the code. Figure 8-1 illustrates this trade-off. The abstraction power of object-oriented languages typically make it easier for programmers to write misleading code. Informally, it is easy to tell what the code was supposed to do, but hard to tell what it actually does. Hence, automated support for analysis and testing is particularly important in an object-oriented environment.

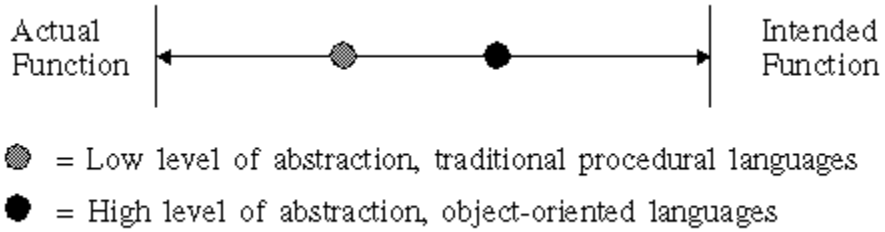

Figure 8-1. Abstraction level trade-off.

When abstraction works well, it hides details that do not need to be tested, so testing becomes easier. When abstraction works poorly, it hides details that need to be tested, so testing becomes harder (or at worst, infeasible). Typically, testing must occur at some compromise level of abstraction, gaining leverage from the abstractions that clarify the software's functionality while penetrating the abstractions that obscure it. Often, a software developer's level of familiarity with object-oriented development techniques is a key factor in determining whether abstraction is used in a positive or negative way, which in turn determines whether testing is helped or hindered by the unique features of object-oriented languages. As discussed in section 8-3, the basic object-oriented structured testing approach can be adapted to unusually positive or negative situations. Metrics designed specifically for object-oriented software, such as those of [CHIDAMBER], can help assess the effectiveness with which the object-oriented paradigm is used. If class methods are not cohesive and there is substantial coupling between objects, the abstractions of the programming language may have been misused in the software. This in turn suggests that extra caution should be taken during testing.

8.2 Object-oriented module testing

Object-oriented methods are similar in most respects to ordinary functions, so structured testing (as well as other structural testing criteria) applies at the module level without modification. Since methods in object-oriented languages tend to be less complex than the functions of traditional procedural programs, module testing is typically easier for object-oriented programs.

The inheritance and polymorphism features of object-oriented languages may seem to complicate module testing, because of the implicit control flow involved in resolving dynamic method invocations. For example, if an object reference could result in any one of several alternative methods being executed at run time, it is possible to represent the reference as a multi-way decision construct ("case" statement) on the flow graph in which a different possible resolution method for the object reference is called from each branch. Since this representation of implicit control flow would increase the cyclomatic complexity, it would also increase the number of tests required to perform structured testing for the module containing the reference. Figure 8-2 illustrates implicit control flow.

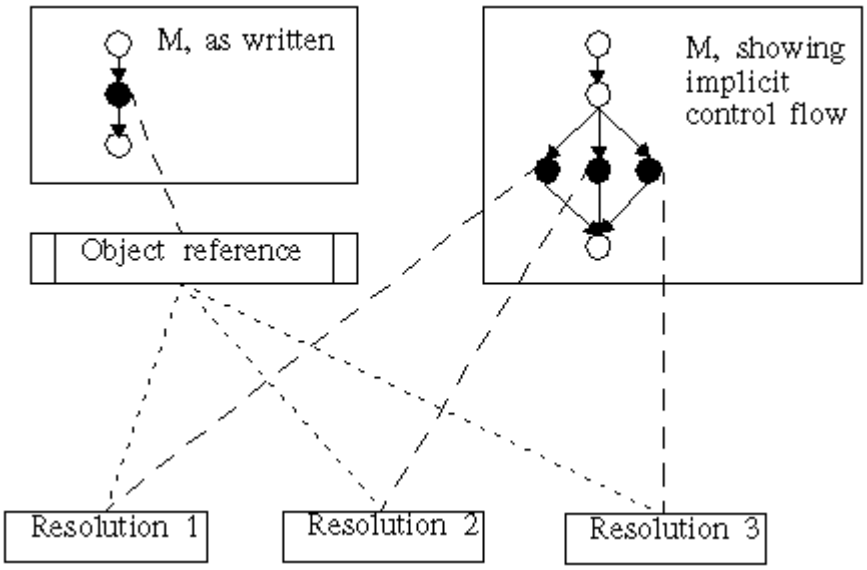

Figure 8-2. Implicit control flow.

Although implicit control flow has significant implications for testing, there are strong reasons to ignore it at the level of module testing. First, the number of possible resolutions of a particular object reference depends on the class hierarchy in which the reference occurs. Hence, if module testing depended on implicit control flow, it would only be possible to perform module testing in the context of a complete class hierarchy, a clearly unrealistic requirement. Also, viewing implicit complexity as a module property rather than a system property defeats one of the major motivations for using an object-oriented language: to develop easily reusable and extensible components. For these reasons, consideration of implicit control flow is deferred until the integration phase when applying structured testing to object-oriented programs.

8.3 Integration testing of object-oriented programs

The most basic application of structured testing to the integration of object-oriented programs is essentially the direct application of the techniques of section 7. It is vitally important, however, to consider the implicit method invocations through object references (not to be confused with the implicit control flow of dynamic binding) when identifying function call nodes to perform design reduction. These method invocations may not be

apparent from examination of the source code, so an automated tool is helpful. For example, if variables "a" and "b" are integers, the C++ expression "a+b" is a simple computation. However, if those variables are objects of a class type, the exact same expression can be a call to a method of that class via the operator overloading mechanism of C++, which requires integration testing and hence must be preserved by design reduction.

While the treatment of implicit method calls due to object references is straightforward, the situation with implicit control flow is more complicated. In this case, several possible approaches are worth considering, depending on the extent to which the object-oriented abstraction has a positive or negative impact. This section describes three approaches: optimistic, pessimistic, and balanced. Each approach builds upon the techniques of section 7, requiring that at least a basis set of tests through the design-reduced graph of each module be exercised in an integration context, treating the implicit method invocations due to object references as calls when performing design reduction. The pessimistic and balanced approaches also require additional tests based on implicit control flow. Figure 8-3 shows the structure of a small subset of a program that illustrates each approach. Each of modules "A," "B," and "C" invokes the dynamically bound "Draw" method, which can be resolved at run time to any of "Line::Draw," "Polygon::Draw," and "Ellipse::Draw." For the rest of this section, the methods that may be invoked as a result of a dynamically bound object reference will be referred to as "resolutions."

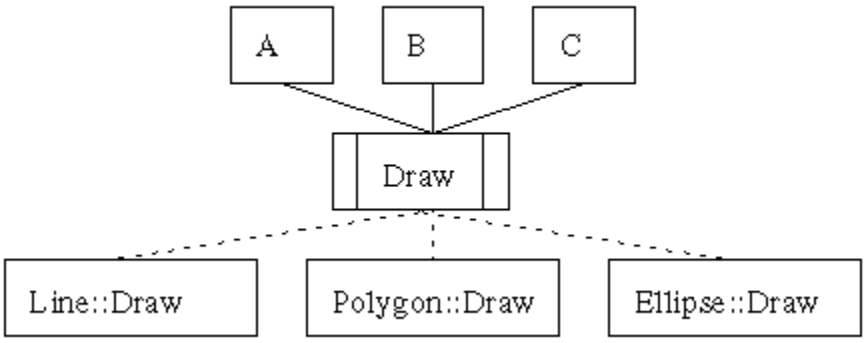

Figure 8-3. Implicit control flow example.

The optimistic approach is the most straightforward integration testing approach. With this approach, it is expected that abstraction will have a positive impact, and therefore testing will be confined to the level of abstraction of the source code. The integration testing techniques of section 7 apply directly. The consequences for implicit control flow are that each call site exercises at least one resolution, and each resolution is exercised by at least one call site. Assuming that no errors are uncovered by testing those interfaces, the object-oriented abstraction is trusted to gain confidence that the other possible interfaces are also correct. Figure 8-4 shows a set of interface tests that are adequate to satisfy the optimistic approach for the example of Figure 8-3.

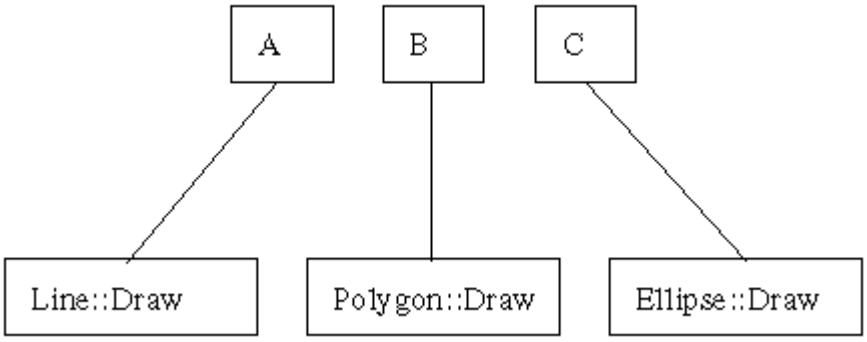

Figure 8-4. The optimistic approach to implicit control flow.

The pessimistic approach is another fairly straightforward approach. With this approach, abstraction is expected to have a negative impact, and therefore testing is required at the level of abstraction of the machine

computation underlying the source code. In addition to the integration testing techniques of section 7, implicit control flow is tested by requiring that every resolution be tested from every call site. Thus, each interface is tested directly, and no trust is placed in the object-oriented abstraction. The drawback is that the testing effort can quickly become infeasible as complexity increases. Figure 8-5 shows a set of interface tests that are adequate to satisfy the pessimistic approach for the example of Figure 8-3.

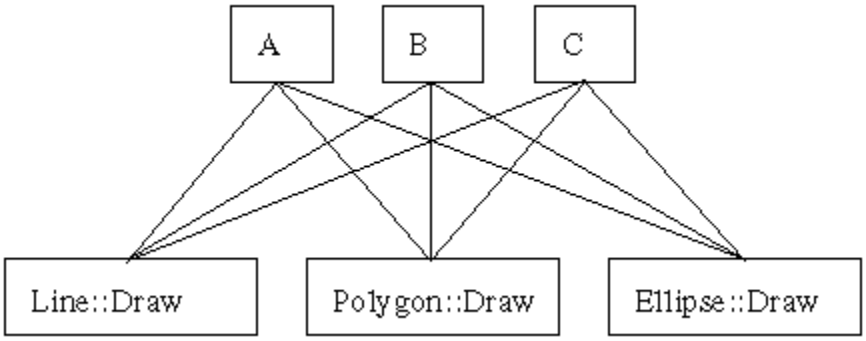

Figure 8-5. The pessimistic approach to implicit control flow.

The balanced approach is a compromise between the optimistic and pessimistic approaches, and is more complicated than either of them. The idea is to test at a level of abstraction between that of the code and that of the underlying mechanics. Abstraction is expected to hide some details that require testing, but also to provide a framework that can be exploited to facilitate testing. In addition to the integration testing techniques of section 7, it is required that some call site exercise the entire set of possible resolutions. The effect of this requirement is to provide evidence that the set of possible resolutions from a given call site form an "equivalence class" in the sense that if exercising one of those resolutions from a new call site works properly than exercising the other possible resolutions are also likely to work properly. This property is assumed by the optimistic approach and exhaustively tested by the pessimistic approach. The balanced approach provides more thorough testing than the optimistic approach without requiring as many tests as the pessimistic approach, and is therefore a good candidate for use in typical situations. Also, the call site used to establish the "equivalence classes" could be a test driver rather than part of the specific application being tested, which provides added flexibility. For example, a test driver program could first be written to test all resolutions of the "Draw" method in the "Shape" class hierarchy, after which the optimistic approach could be used without modification when testing "Draw" method invocations in programs using the "Shape" hierarchy. Figure 8-6 shows a set of interface tests that are adequate to satisfy the balanced approach for the example of Figure 8-3.

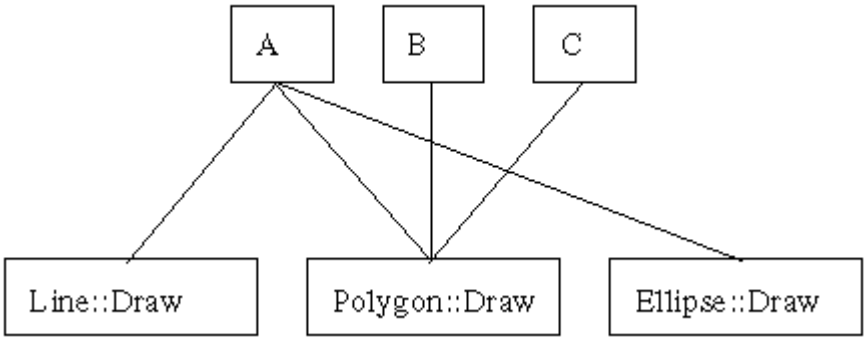

Figure 8-6. The balanced approach to implicit control flow.

Specific resolutions to dynamic control flow are often of interest. For example, the bulk of a drawing application's typical usage may involve polygons, or the polygon functionality may have been recently re-implemented. In that case, it is appropriate to consider a system view in which all shapes are assumed to be polygons, for example connecting all the polymorphic "Draw" calls directly to "Polygon::Draw" and removing alternatives such as "Ellipse::Draw" from consideration. For such a set of resolutions, the *object integration complexity*, *OS*1, is defined as the integration complexity (S1) of the corresponding resolved system. Object

integration complexity is very flexible, since its measurement is based on any desired set of resolutions. Those resolutions could be specified either for specific polymorphic methods, or more generally for entire classes.

8.4 Avoiding unnecessary testing

Object-oriented systems are often built from stable components, such as commercial class libraries or re-used classes from previous successful projects. Indeed, component-based reuse is one of the fundamental goals of object-oriented development, so testing techniques must allow for it. Stable, high integrity software can be referred to as "trusted." The concept of trustedness can apply to individual modules, classes, class hierarchies, or a set of potential resolutions for dynamically bound methods, but in each case the meaning is that trusted software is assumed to function correctly and to conform to the relevant object-oriented abstraction. In addition to trusted commercial and reused software, new software becomes trusted after it has been properly tested. In structured testing, the implementation of trusted software is not tested, although its integration with other software is required to be tested. Trusted software is treated as an already-integrated component using the incremental integration techniques of section 7-6.

Trusted software does not have to be tested at the module level at all, and calls internal to a trusted component do not have to be tested at the integration level. The handling of trusted modules, classes, and class hierarchies is straightforward, in that only integration testing need be performed, and even then applying the design reduction technique based on only those calls that cross a boundary of trustedness. For the case of a trusted set of potential resolutions for a dynamically bound method, only one resolution need be tested from each call site even when using the pessimistic approach for testing non-trusted code. When an automated tool is used to display trustedness information, integration tests can be stopped at the trustedness boundary.