Chapter 15

Testing Object-Oriented Software

Systematic testing of object-oriented software is fundamentally similar to systematic testing approaches for procedural software: We begin with functional tests based on specification of intended behavior, add selected structural test cases based on the software structure, and work from unit testing and small-scale integration testing toward larger integration and then system testing. Nonetheless the differences between procedural software and object-oriented software are sufficient to make specialized techniques appropriate.

Required Background

- Chapters 11, 12, and 13, and 14.
  This chapter builds on basic functional, structural, and model-based testing techniques, including data flow testing techniques. Some basic techniques, described more thoroughly in earlier chapters, are recapped very briefly here to provide flexibility in reading order.
  - Chapter 5
    Many of the techniques described here employ finite state machines for modeling object state.

15.1 Overview

Object-oriented software differs sufficiently from procedural software to justify reconsidering and adapting approaches to software test and analysis. For example, methods in object-oriented software are typically shorter than procedures in other software, so faults in complex intra-procedural logic and control flow occur less often and merit
less attention in testing. On the other hand, short methods together with encapsulation of object state suggest greater attention to interactions among method calls, while polymorphism, dynamic binding, generics, and increased use of exception handling introduce new classes of fault that require attention.

Some traditional test and analysis techniques are easily adapted to object-oriented software. For example, code inspection can be applied to object-oriented software much as it is to procedural software, albeit with different checklists. In this chapter we will be concerned mostly with techniques that require more substantial revision (like conventional structural testing techniques) and on introduction of new techniques for coping with problems associated with object-oriented software.

### 15.2 Issues in Testing Object-Oriented Software

Characteristics of object-oriented software that impact test design are summarized in the sidebar on page 325 and discussed in more detail below.

The behavior of object-oriented programs is inherently stateful: The behavior of a method depends not only on the parameters passed explicitly to the method, but also on the state of the object. For example, method `CheckConfiguration()` of class `Model`, shown in Figure 15.1, returns `True` or `False` depending on whether all components are bound to compatible slots in the current object state.

In object-oriented programs, `public` and `private` parts of a class (fields and methods) are distinguished. Private state and methods are inaccessible to external entities, which can only change or inspect private state by invoking public methods. For example, the instance variable `modelID` of class `Model` in Figure 15.1 is accessible by external entities, but `slots` and `legalConfig` are accessible only within methods of the same class. The constructor `Model()` and the method `checkConfiguration()` can be used by external entities to create new objects and to check validity of the current configuration, while method `openDB()` can be invoked only by methods of this class.

Encapsulated information creates new problems in designing oracles and test cases. Oracles must identify incorrect (hidden) state, and test cases must exercise objects in different (hidden) states.

Object-oriented programs include classes that are defined by extending or specializing other classes through inheritance. For example, class `Model` in Figure 15.1 extends class `CompositeItem`, as indicated in the class declaration. A child class can inherit variables and methods from its ancestors, overwrite others, and add yet others. For example, the class diagram of Figure 15.3 shows that class `Model` inherits the instance variables `sku`, `units`, and `parts`, and methods `validItem()`, `getUnitPrice()` and `getExtendedPrice()`. It overwrites methods `getHeightCm()`, `getWidthCm()`, `getDepthCm()` and `getWeightGm()`. It adds

---

1 Object-oriented languages differ with respect to the categories of accessibility they provide. For example, nothing in Java corresponds exactly to the “friend” functions in C++ that are permitted to access the private state of other objects. But while details vary, encapsulation of state is fundamental to the object-oriented programming paradigm, and all major object-oriented languages have a construct comparable to Java’s private field declarations.

---

Draft version produced February 22, 2006
<table>
<thead>
<tr>
<th>Summary: Relevant Characteristics of Object-Oriented Software</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State Dependent Behavior</strong>: Testing techniques must consider the state in which methods are invoked. Testing techniques that are oblivious to state (e.g., traditional coverage of control structure) are not effective in revealing state-dependent faults.</td>
</tr>
<tr>
<td><strong>Encapsulation</strong>: The effects of executing object-oriented code may include outputs, modification of object state, or both. Test oracles may require access to private (encapsulated) information to distinguish between correct and incorrect behavior.</td>
</tr>
<tr>
<td><strong>Inheritance</strong>: Test design must consider the effects of new and overridden methods on the behavior of inherited methods, and distinguish between methods that require new test cases, ancestor methods that can be tested by re-executing existing test cases, and methods that do not need to be retested.</td>
</tr>
<tr>
<td><strong>Polymorphism and Dynamic Binding</strong>: A single method call may be dynamically bound to different methods depending on the state of the computation. Tests must exercise different bindings to reveal failures that depend on a particular binding or on interactions between bindings for different calls.</td>
</tr>
<tr>
<td><strong>Abstract Classes</strong>: Abstract classes cannot be directly instantiated and tested, yet they may be important interface elements in libraries and components. It is necessary to test them without full knowledge of how they may be instantiated.</td>
</tr>
<tr>
<td><strong>Exception Handling</strong>: Exception handling is extensively used in modern object-oriented programming. The textual distance between the point where an exception is thrown and the point where it is handled, and the dynamic determination of the binding, makes it important to explicitly test exceptional as well as normal control flow.</td>
</tr>
<tr>
<td><strong>Concurrency</strong>: Modern object-oriented languages and toolkits encourage and sometimes even require multiple threads of control (e.g., the Java user interface construction toolkits AWT and Swing). Concurrency introduces new kinds of possible failures, such as deadlock and race conditions, and makes the behavior of a system dependent on scheduler decisions which are not under the tester’s control.</td>
</tr>
</tbody>
</table>
public class Model extends Orders.CompositeItem {
    public String modelID; // Database key for slots
    private int baseWeight; // Weight excluding optional components
    private int heightCm, widthCm, depthCm; // Dimensions if boxed
    private Slot[] slots; // Component slots

    private boolean legalConfig = false; // memoized result of isLegalConf
    private static final String NoModel = "NO MODEL SELECTED";

    /* Constructor, which should be followed by selectModel */
    public Model(Orders.Order _order) {
        super(_order);
        modelID = NoModel;
    }

    /** Is the current binding of components to slots a legal
     * configuration? Memo-ize the result for repeated calls */
    public boolean isLegalConfiguration() {
        if (! legalConfig) {
            checkConfiguration();
        }
        return legalConfig;
    }

    /** Are all required slots filled with compatible components? * It is impossible to assign an incompatible component,
     * so just to check that every required slot is filled. */
    private void checkConfiguration() {
        legalConfig = true;
        for (int i=0; i < slots.length; ++i) {
            Slot slot = slots[i];
            if (slot.required && ! slot.isBound()) {
                legalConfig = false;
            }
        }
    }

    // ...
}

Figure 15.1: Part of a Java implementation of class Model
1 public class Model extends Orders.CompositeItem {

/** Bind a component to a slot. */
* @param slotIndex Which slot (integer index)?
* @param sku Key to component database.
* Choices should be constrained by web interface, so we don’t
* need to be graceful in handling bogus parameters.
*/

public void addComponent(int slotIndex, String sku) {
    Slot slot = slots[slotIndex];
    if (componentDB.contains(sku)) {
        Component comp = new Component(order, sku);
        if (comp.isCompatible(slot.slotID)) {
            slot.bind(comp);
            // Note this cannot have made the
            // configuration illegal.
        } else {
            slot.unbind();
            legalConfig = false;
        }
    } else {
        slot.unbind();
        legalConfig = false;
    }
}

/** Unbind a component from a slot. */
public void removeComponent(int slotIndex) {
    // assert slotIndex in 0..slots.length
    if (slots[slotIndex].isBound()) {
        slots[slotIndex].unbind();
        legalConfig = false;
    }
}

Figure 15.2: More of the Java implementation of class Model. Because of the way
method isLegalConfig is implemented (see Figure 15.1), all methods that modify
slots must reset the private variable legalConfig.
Inheritance brings in optimization issues. Child classes may share several methods with their ancestors. Sometimes an inherited method must be re-tested in the child class, despite not having been directly changed, because of interaction with other parts of the class that have changed. Many times, though, one can establish conclusively that the behavior of an inherited method is really unchanged and need not be re-tested. In other cases, it may be necessary to re-run tests designed for the inherited method, but not necessary to design new tests.

Most object-oriented languages allow variables to dynamically change their type, as long as they remain within a hierarchy rooted at the declared type of the variable. For example, variable subsidiary of method getYTD摊丁买长() in Figure 15.4 can be dynamically bound to different classes of the Account hierarchy, and thus the invocation of method subsidiary.getYTD摊丁买长() can be bound dynamically to different methods.

Dynamic binding to different methods may affect the whole computation. Testing a call by considering only one possible binding may not be enough. Test designers need testing techniques that select subsets of possible bindings that cover a sufficient range of situations to reveal faults in possible combinations of bindings.

Some classes in an object-oriented program are intentionally left incomplete and cannot be directly instantiated. These abstract classes\(^2\) must be extended through subclasses; only sub-classes that fill in the missing details (e.g., method bodies) can be instantiated. For example, both classes LineItem of Figure 15.3 and Account of Figure 15.4 are abstract.

If abstract classes are part of a larger system, such as the Chipmunk web presence, and if they are not part of the public interface to that system, then they can be tested by testing all their child classes: classes Model, Component, CompositeItem, and SimpleItem for class LineItem and classes USAccount, UKAccount, JPAccount, EUAccount and OtherAccount for class Account. However, we may need to test an abstract class either prior to implementing all child classes, for example if not all child classes will be implemented by the same engineers in the same time frame, or without knowing all its implementations, for example if the class is included in a library whose reuse cannot be fully foreseen at development time. In these cases, test designers need techniques for selecting a representative set of instances for testing the abstract class.

Exceptions were originally introduced in programming languages independently of object-oriented features, but they play a central role in modern object-oriented programming languages and in object-oriented design methods. Their prominent role in object-oriented programs, and the complexity of propagation and handling of exceptions during program execution, call for careful attention and specialized techniques in testing.

The absence of a main execution thread in object-oriented programs make them

---

\(^2\)Here we include the Java interface construct as a kind of abstract class.
Figure 15.3: An excerpt from the class diagram of the Chipmunk web presence that shows the hierarchy rooted in class `LineItem`.
public abstract class Account {
   ...
   /**
    * The YTD Purchased amount for an account is the YTD
    * total of YTD purchases of all customers using this account
    * plus the YTD purchases of all subsidiaries of this account;
    * currency is currency of this account.
    */
   public int getYTDpurchased() {
      if (ytdPurchasedValid) { return ytdPurchased; }
      int totalPurchased = 0;
      for (Enumeration e = subsidiaries.elements() ; e.hasMoreElements(); )
      {
         Account subsidiary = (Account) e.nextElement();
         totalPurchased += subsidiary.getYTDpurchased();
      }
      for (Enumeration e = customers.elements(); e.hasMoreElements(); )
      {
         Customer aCust = (Customer) e.nextElement();
         totalPurchased += aCust.getYearlyPurchase();
      }
      ytdPurchased = totalPurchased;
      ytdPurchasedValid = true;
      return totalPurchased;
   }
   ...
}

Figure 15.4: Part of a Java implementation of Class Account. The abstract class is specialized by the regional markets served by Chipmunk into USAccount, UKAccount, JPAccount, EUAccount, OtherAccount, which differ with regard to shipping methods, taxes, and currency. A corporate account may be associated with several individual customers, and large companies may have different subsidiaries with accounts in different markets. Method getYTDpurchased() sums the year-to-date purchases of all customers using the main account and the accounts of all subsidiaries.
Object-oriented design and programming greatly impact analysis and testing. However, test designers should not make the mistake of ignoring traditional technology and methodologies. A specific design approach mainly affects detailed design and code, but there are many aspects of software development and quality assurance that are largely independent of the use of a specific design approach. In particular, aspects related to planning, requirements analysis, architectural design, deployment and maintenance can be addressed independently of the design approach. Figure 15.5 indicates the scope of the impact of object-oriented design on analysis and testing.

### 15.3 An Orthogonal Approach to Test

Testing all aspects of object-oriented programs simultaneously would be difficult and expensive; fortunately it is also unnecessary. It is more cost-effective to address different features individually, using appropriate techniques for each, and to explicitly
address significant interactions (e.g., between inheritance and state-dependent behavior) rather than blindly exploring all different feature combinations.

The proper blend of techniques depends on many factors: the application under test, the development approach, team organization, application criticality, the development environment and the implementation languages, the use of design and language features, and project timing and resource constraints. Nonetheless, we can outline a general approach that works in stages, from single classes to class and system interactions. A single “stage” is actually a set of interrelated test design and test execution activities. The approach is summarized in the sidebar on page 333 and described in more detail below in the order that tests related to a particular class would be executed, although test design and execution activities are actually interleaved and distributed through development.

The smallest coherent unit for unit testing of object-oriented testing is the class. Test designers can address inheritance, state-dependent behavior and exceptions with intra-class testing. For example, when testing class Model of Figure 15.3, test designers may first use testing histories (see Section 15.10) to infer that method getExtendedPrice need not be retested, since it has already been tested in class LineItem. On the other hand, test designers must derive new test cases for the new methods and for those affected by the modifications introduced in class Model.

After considering individual methods, test designers can proceed to design functional test cases from the statechart specification of class Model (see Section 15.5) and structural test cases from data flow information (see Section 15.7). To execute test cases, test designers may decide to use equivalent scenarios as oracles (see Section 15.8). Test designers will then create test cases for exceptions thrown or handled by the class under test (see Section 15.12). Class Model does not make polymorphic calls, so no additional test cases need be designed to check behavior with variable bindings to different classes.

Integration (inter-class) tests must be added to complete the testing for hierarchy, polymorphism, and exception-related problems. For example, when testing integration of class Model within the Chipmunk web presence, test designers will identify class Slot as a predecessor in the integration order and will test it first, before testing its integration with class Model (see Sections 15.5 and 15.7). They will also derive test cases for completing the test of exceptions (see Section 15.12) and polymorphism (see Section 15.9).

System and acceptance testing check overall system behavior against user and system requirements. Since these requirements are (at least in principle) independent of the design approach, system and acceptance testing can be addressed with traditional techniques. For example, to test the business logic subsystem of the Chipmunk web presence, test designers may decide to derive test cases from functional specifications using category partition and catalog based methods (see Chapter 11).

### 15.4 Intra-Class Testing

Unit and integration testing aim to expose faults in individual program units and in their interactions, respectively. The meaning of “unit” is the smallest development
Steps in Object-Oriented Software Testing

Object-oriented testing can be broken into three phases, progressing from individual classes toward consideration of integration and interactions.

**Intra-class:** testing classes in isolation (unit testing)

1. If the class-under-test is abstract, derive a set of instantiations to cover significant cases. Instantiations may be taken from the application (if available) and/or created just for the purpose of testing.
2. Design test cases to check correct invocation of inherited and overridden methods, including constructors. If the class-under-test extends classes that have previously been tested, determine which inherited methods need to be re-tested and which test cases from ancestor classes can be re-used.
3. Design a set of intra-class test cases based on a state machine model of specified class behavior.
4. Augment the state machine model with structural relations derived from class source code and generate additional test cases to cover structural features.
5. Design an initial set of test cases for exception handling, systematically exercising exceptions that should be thrown by methods in the class under test and exceptions that should be caught and handled by them.
6. Design an initial set of test cases for polymorphic calls

**Inter-class:** testing class integration (integration testing)

1. Identify a hierarchy of clusters of classes to be tested incrementally.
2. Design a set of functional inter-class test cases for the cluster-under-test.
3. Add test cases to cover data flow between method calls.
4. Integrate the intra-class exception handling test sets with test with inter-class exception handling test cases for exceptions propagated across classes.
5. Integrate the polymorphism test sets with tests that check for inter-class interactions of polymorphic calls and dynamic bindings.

**System and Acceptance:** apply standard functional and acceptance testing techniques to larger components and the whole system.
work assignment for a single programmer that can reasonably be planned and tracked. In procedural programs, individual program units might be single functions or small sets of strongly related functions and procedures, often included in a single file of source code. In object-oriented programs, small sets of strongly related functions or procedures are naturally identified with classes, which are generally the smallest work units that can be systematically tested.

Treating an individual method as a unit is usually not practical, because methods in a single class interact by modifying object state, and because the effect of an individual method is often visible only through its effect on other methods. For example, method check_configuration of class computer, shown in Figure 15.1, can be executed only if the object is in a given state, and its result depends on the current configuration. The method may execute correctly in a given state, i.e., for a given configuration, but may not execute correctly in a different state, e.g., accepting malformed configurations or rejecting acceptable configurations. Moreover, method check_configuration might produce an apparently correct output (return value) but leave the object in an incorrect state.

15.5 Testing with State Machine Models

Since the state of an object is implicitly part of the input and output of methods, we need a way to systematically explore object states and transitions. This can be guided by a state machine model, which can be derived from module specifications.

A state machine model can be extracted from an informal, natural language specification of intended behavior, even when the specification does not explicitly describe states and transitions. States can be inferred from descriptions of methods that act differently or return different results depending on the state of the object; this includes any description of when it is allowable to call a method. Of course, one wants to derive only a reasonable number of abstract states as representatives of a much larger number of concrete states, and some judgment is required to choose the grouping. For example, if an object kept an integer count, we might choose “zero” and “non-zero” as representative states, rather than creating a different state for every possible value. The principle to observe is that we are producing a model of how one method affects another, so the states should be refined just enough to capture interactions. Extracting a state machine from an informal specification, and then creating test cases (sequences of method calls) to cover transitions in that model, is illustrated in the sidebar on page 335.

Sometimes an explicit state machine model is already available as part of a specification or design. If so, it is likely to be in the form of a statechart (also known as a state diagram in the UML family of notations). Statecharts include standard state transition diagrams, but also provide hierarchical structuring constructs. The structuring facilities of statecharts can be used to organize and hide complexity, but this complexity must be exposed to be tested.

The most common structuring mechanism in statecharts is grouping of states in super-states (also called OR-states). A transition from a super-state is equivalent to a transition from every state contained within it. A transition to a super-state is equivalent to a transition to the initial state within the super-state. We can obtain an...
An Informal Specification of Class Slot

Slot represents a configuration choice in all instances of a particular model of computing. It may or may not be implemented as a physical slot on a bus. A given model may have zero or more slots, each of which is marked as required or optional. If a slot is marked as "required," it must be bound to a suitable component in all legal configurations.

Class slot offers the following services:

- **Incorporate:** Make a slot part of a model, and mark it as either required or optional. All instances of a model incorporate the same slots. *Example:* We can incorporate a required primary battery slot and an optional secondary battery slot on the Chipmunk C20 laptop that includes two battery slots. The C20 laptop may then be sold with one battery or two batteries, but it is not sold without at least the primary battery.

- **Bind:** Associate a compatible component with a slot. *Example:* We can bind slot primary battery to a Blt4, Blt6 or Blt8 lithium battery or to a Bcdm4 nickel cadmium battery. We cannot bind a disk drive to the battery slot.

- **Unbind:** The unbind operation breaks the binding of a component to a slot, reversing the effect of a previous bind operation.

- **IsBound:** returns true if a component is currently bound to a slot, or false if the slot is currently empty.

The Corresponding Finite State Machine

A simple analysis of the informal specification of class Slots allows one to identify states and transitions. Often an analysis of natural language specifications will reveal ambiguities that must be resolved one way or the other in the model; these may suggest additional test cases to check the interpretation, or lead to refinement of the specification, or both. For class slot, we infer that the bind operation makes sense only after the slot has been incorporated in a model, and that it is initially empty.

The Generated Test Case Specifications

A single test case will be given as a sequence of method calls. For class Slot, the following test cases suffice to execute each transition in the state machine model:

TC-1 incorporate, isBound, bind, isBound
TC-2 incorporate, unBind, bind, unBind, isbound
ordinary state machine by “flattening” the statechart hierarchy, replacing transitions to and from super-states to transitions among elementary states.

Figure 15.6 shows a statechart specification for class Model of the business logic of the Chipmunk web presence. Class Model provides methods for selecting a computer model and a set of components to fill logical and physical slots. The state modelSelected is decomposed into its two component states, with entries to modelSelected directed to the default initial state workingConfiguration.

Table 15.1 shows a set of test cases that cover all transitions of the finite state machine of Figure 15.7, a flattened version of the statechart of Figure 15.6. Notice that transition selectModel of the statechart corresponds to a single transition in the FSM, since entry to the super-state is directed to the default initial state, while transition deselectModel of the statechart corresponds to two transitions in the FSM, one for each of the two children states, since the super-state can be exited while in either component state.

In covering the state machine model, we have chosen sets of transition sequences that together exercise each individual transition at least once. This is the transition coverage criterion introduced in Chapter 14. The stronger history-sensitive criteria described in that chapter are also applicable in principle, but are seldom used because of their cost.

Even transition coverage may be impractical for complex statecharts. The number
Test Case $T_{CA}$
- selectModel(M1)
- addComponent(S1,C1)
- addComponent(S2,C2)
- isLegalConfiguration()

Test Case $T_{CB}$
- selectModel(M1)
- deselectModel()
- selectModel(M2)
- addComponent(S1,C1)
- addComponent(S2,C2)
- removeComponent(S1)
- isLegalConfiguration()

Test Case $T_{CC}$
- selectModel(M1)
- addComponent(S1,C1)
- removeComponent(S1)
- addComponent(S1,C2)
- isLegalConfiguration()

Test Case $T_{CD}$
- selectModel(M1)
- addComponent(S1,C1)
- addComponent(S2,C2)
- addComponent(S3,C3)
- deselectModel()
- selectModel(M1)
- addComponent(S1,C1)
- isLegalConfiguration()

Test Case $T_{CE}$
- selectModel(M1)
- addComponent(S1,C1)
- addComponent(S2,C2)
- addComponent(S3,C3)
- removeComponent(S1)
- addComponent(S2,C4)
- isLegalConfiguration()

Table 15.1: A set of test cases that satisfies the transition coverage criterion for the statechart of Figure 15.6.
of states and transitions can explode in “flattening” a statechart that represents multiple threads of control. Unlike flattening of ordinary super-states, which leaves the number of elementary states unchanged while replicating some transitions, flattening of concurrent state machines (so-called “AND-states”) produces new states that are combinations of elementary states.

Figure 15.8 shows the statechart specification of class Order of the business logic of the Chipmunk web presence. Figure 15.9 shows the corresponding “flattened” state machine. Flattening the AND-state results in a number of states equal to the Cartesian product of the elementary states \(3 \times 3 = 9\) states and a corresponding number of transitions. For instance, transition \textit{add item} that exits state \textit{not scheduled} of the statechart corresponds to three transitions exiting the states \textit{not schedXcanc no fee}, \textit{not schedXcanc fee}, and \textit{not schedXnot canc}, respectively. Covering all transitions at least once may result in a number of test cases that exceeds the budget for testing the class. In this case, we may forgo flattening and use simpler criteria that take advantage of the hierarchical structure of the statechart.

Table 15.5 shows a test suite that satisfies the simple transition coverage criterion, which requires the execution of all transitions that appear in the statechart. The criterion requires that each statechart transition is exercised at least once, but does not guarantee that transitions are exercised in all possible states. For example, transition \textit{add item}, which leaves the initial state, is exercised from at least one sub-state, but not from all possible sub-states as required by the transition coverage criterion.

15.6 Inter-class Testing

Inter-class testing is the first level of integration testing for object-oriented software. While intra-class testing focuses on single classes, inter-class testing checks interactions among objects of different classes. As in integration testing of imperative programs, test designers proceed incrementally, starting from small clusters of classes.

Since the point of inter-class testing is to verify interactions, it is useful to model potential interactions through a use/include relation. Classes A and B are related by the use/include relation if objects of class A make method calls on objects of class B, or if objects of class A contain references to objects of class B. Inheritance is ignored (we do not consider a subclass to use or include its ancestors) and abstract classes, which cannot directly participate in interactions, are omitted. Derivation of the use/include relation from a conventional UML class diagram is illustrated in Figures 15.10 and 15.11.

Inter-class testing strategies usually proceed bottom-up, starting from classes that depend on no others. The implementation-level use/include relation among classes typically parallels the more abstract, logical \textit{depends} relation among modules (see sidebar on page 344), so a bottom-up strategy works well with cluster-based testing. For example, we can start integrating class slotDB with class slot, and class Component with class ComponentDB, and then proceed incrementally integrating classes ModelDB and Model, up to class Order.

Well-designed systems normally have nearly acyclic dependence relations, with dependence loops limited to closely related clusters. When there are larger loops in
Figure 15.8: Statechart specification of class `Order`. This is a conceptual model in which both methods of class `Order` and method calls by class `Order` are represented as transitions with names that differ from method names in the implementation (e.g., `5DaysBeforeShipping` is not a legal method or field name).
Figure 15.9: Finite state machine corresponding to the statechart of Figure 15.8
<table>
<thead>
<tr>
<th>Test Case $TC_A$</th>
<th>Test Case $TC_B$</th>
<th>Test Case $TC_C$</th>
<th>Test Case $TC_D$</th>
<th>Test Case $TC_E$</th>
<th>Test Case $TC_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_item()</td>
<td>add_item()</td>
<td>add_item()</td>
<td>add_item()</td>
<td>add_item()</td>
<td>add_item()</td>
</tr>
<tr>
<td>add_item()</td>
<td>add_item()</td>
<td>add_item()</td>
<td>package()</td>
<td>package()</td>
<td>package()</td>
</tr>
<tr>
<td>package()</td>
<td>remove_item()</td>
<td>package()</td>
<td>get_shipping_cost()</td>
<td>get_discount()</td>
<td>get_discount()</td>
</tr>
<tr>
<td>get_shipping_cost()</td>
<td>get_discount()</td>
<td>purchase()</td>
<td>purchase()</td>
<td>purchase()</td>
<td>purchase()</td>
</tr>
<tr>
<td>purchase()</td>
<td>place_order()</td>
<td>place_order()</td>
<td>place_order()</td>
<td>place_order()</td>
<td>place_order()</td>
</tr>
<tr>
<td>24_hours()</td>
<td>24_hours()</td>
<td>5_days()</td>
<td>schedule()</td>
<td>schedule()</td>
<td>schedule()</td>
</tr>
<tr>
<td>5_days()</td>
<td>5_days()</td>
<td>24_hours()</td>
<td>5_days()</td>
<td>5_days()</td>
<td>5_days()</td>
</tr>
<tr>
<td>schedule()</td>
<td>schedule()</td>
<td>schedule()</td>
<td>ship()</td>
<td>ship()</td>
<td>ship()</td>
</tr>
<tr>
<td>ship()</td>
<td>ship()</td>
<td>ship()</td>
<td>deliver()</td>
<td>deliver()</td>
<td>deliver()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case $TC_G$</th>
<th>Test Case $TC_H$</th>
<th>Test Case $TC_I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_item()</td>
<td>add_item()</td>
<td>add_item()</td>
</tr>
<tr>
<td>add_item()</td>
<td>add_item()</td>
<td>add_item()</td>
</tr>
<tr>
<td>package()</td>
<td>package()</td>
<td>package()</td>
</tr>
<tr>
<td>get_shipping_cost()</td>
<td>get_shipping_cost()</td>
<td>get_shipping_cost()</td>
</tr>
<tr>
<td>get_discount()</td>
<td>get_discount()</td>
<td>get_discount()</td>
</tr>
<tr>
<td>purchase()</td>
<td>purchase()</td>
<td>purchase()</td>
</tr>
<tr>
<td>place_order()</td>
<td>place_order()</td>
<td>place_order()</td>
</tr>
<tr>
<td>schedule()</td>
<td>schedule()</td>
<td>schedule()</td>
</tr>
<tr>
<td>ship()</td>
<td>ship()</td>
<td>ship()</td>
</tr>
<tr>
<td>address unknown</td>
<td>address_unknown()</td>
<td>cancel()</td>
</tr>
</tbody>
</table>

Table 15.2: A test suite that satisfies the simple transition coverage criterion for the statechart of Figure 15.8. Transitions are indicated without parameters for simplicity.
Figure 15.10: Part of a class diagram of the Chipmunk web presence. Classes Account, Lineitem, and CSVdb are abstract.
Figure 15.11: Use/include relation for the class diagram of figure 15.10. Abstract classes are not included. Two classes are related if one uses or includes the other. Classes that are higher in the diagram include or use classes that are lower in the diagram.
Dependence

The hierarchy of clusters for inter-class testing is based on a conceptual relation of dependence, and not directly on concrete relations among implementation classes (or implementation-level design documentation).

Module \( A \) depends on module \( B \) if the functionality of \( B \) must be present for the functionality of \( A \) to be provided.

If \( A \) and \( B \) are implemented as classes or clusters of closely related classes, it is likely that the logical depends relation will be reflected in concrete relations among the classes. Typically, the class or classes in \( A \) will either call methods in the class or classes in \( B \), or classes in \( A \) will have references to classes in \( B \) forming a contains relation among their respective objects.

Concrete relations among classes do not always indicate dependence. It is common for contained objects to have part-of relations with their ancestors in the containment hierarchy, but the dependence is normally from container to contained object and not vice versa. It is also common to find calls from framework libraries to methods that use those libraries. For example, the SAX API for parsing XML is an event-driven parsing framework, which means the parsing library makes calls (through interfaces) on methods provided by the application. This style of event handling is most familiar to Java programmers through the standard Java graphical user interface libraries. It is clear that the application depends on the library and not vice versa.

The depends relation is as crucial to other software development processes as it is to testing. It is essential to building a system as a set of incremental releases, and to scheduling and managing the construction of each release. The depends relation may be documented in UML package diagrams, and even if not documented explicitly it is surely manifest in the development build order. Test designers may (and probably should) be involved in defining the build order, but should not find themselves in the position of discovering or re-creating it after the fact.

The relation, or when a use/include relation among classes runs contrary to the depends relation (e.g., an “up-call” to an ancestor in the depends relation), the loop can be broken by substituting a stub for the ancestor class. Thus we always work with an acyclic graph of clusters.

In principle, while climbing the dependence relation, a thorough inter-class testing should consider all combinations of possible interactions. If for example a test case for class \( \text{Order} \) includes a call to a method of class \( \text{Model} \), and the called method calls a method of class \( \text{Slot} \), each call should be exercised for all possible relevant states of the different classes, as identified during intra-class testing. However, this suffers from the same kind of combinatorial explosion that makes flattening concurrent state diagrams impractical. We need to select a subset of interactions among the possible combinations of method calls and class states. An arbitrary or random selection of interactions may be an acceptable solution, but in addition one should explicitly test any significant interaction scenarios that have been previously identified in design and analysis.
Interaction scenarios may have been recorded in the form of UML interaction diagrams, expressed as sequence or collaboration diagrams. These diagrams describe interactions among objects, and can be considered essentially as test scenarios created during the course of design.

In addition to testing the scenarios spelled out in sequence or collaboration diagrams, the test designer can vary those scenarios to consider illegal or unexpected interaction sequences. For example, replacing a single interaction in a sequence diagram with another interaction which should not be permitted at that point yields a test case that checks error handling.

Figure 15.12 shows a possible pattern of interactions among objects, when a customer assembling an order first selects the computer model C20, then adds a hard disk HD60 which is not compatible with the slots of the selected model, and then adds “legal” hard disk HD20. The sequence diagram indicates the sequence of interactions among objects and suggests possible testing scenarios. For example, it suggests adding a component after having selected a model, i.e., it indicates interesting states of objects of type ModelDB and Slots when testing class Model.

Unlike statecharts, which should describe all possible sequences of transitions that an object can undergo, interaction diagrams illustrate selected interactions that the designers considered significant because they were typical, or perhaps because they were difficult to understand. Deriving test cases from interaction diagrams is useful as a way of choosing some significant cases among the enormous variety of possible interaction sequences, but it is insufficient as a way of ensuring thorough testing. Integration tests should at the very least repeat coverage of individual object states and transitions in the context of other parts of the cluster under test.

15.7 Structural Testing of Classes

In testing procedural code, we take specifications as the primary source of information for test design (functional testing) and then analyze implementation structure and add test cases as needed to cover additional variation (structural testing). The same approach applies to object-oriented programs, and for the same reasons. The techniques described in previous sections are all based on specification of intended behavior. They should be augmented (but never replaced) by structural techniques.

If we compare the implementation of class Model shown in Figures 15.1 and 15.2 with its specification in Figures 15.3 and 15.6, we notice that the code uses an instance variable legalConfig and an internal (private) method checkConfiguration to optimize the implementation of method isLegalConfiguration. The functional test cases shown in Table 15.1 do not include method checkConfiguration, though some of them will call it indirectly through isLegalConfiguration. An alert test designer will note that every modification of the object state that could possibly invalidate a configuration should reset the hidden legalConfig variable to False, and will derive structural test cases to cover behaviors not sufficiently exercised by functional test cases.

The chief difference between functional testing techniques for object oriented software and their counterparts for procedural software (Chapters 10, 11 and 14) is the cen-
Figure 15.12: A (partial) sequence diagram that specifies the interactions among objects of type Order, Model, ModelDB, Component, ComponentDB, Slots and SlotDB, to select a computer, add an illegal component, and then add a legal one.
Central role of object state and of sequences of method invocations to modify and observe object state. Similarly, structural test design must be extended beyond consideration of control and data flow in a single method to take into account how sequences of method invocations interact. For example, tests of isLegalConfiguration would not be sufficient without considering the prior state of private variable legalConfig.

Since the state of an object is comprised of the values of its instance variables, the number of possible object states can be enormous. We might choose to consider only the instance variables that do not appear in the specification, and add only those to the state machine representation of the object state. In the example of class Model, we will have to add only the state of the boolean variable legalConfig, which can at most double the number of states (and at worst quadruple the number of transitions). While we can model the concrete values of a single boolean variable like legalConfig, this approach would not work if we had a dozen such variables, or even a single integer variable introduced in the implementation. To reduce the enormous number of states obtained by considering the combinations of all values of the instance variables, we could select a few representative values.

Another way to reduce the number of test cases based on interaction through instance variable values while remaining sensitive enough to catch many common oversights is to model, not the values of the variables, but the points at which the variables receive those values. This is the same intuition behind data flow testing described in Chapter 13, although it requires some extension to cover sequences in which one method defines (sets) a variable, and another uses that variable. Definition-use pairs for instance variables are computed on an intra-class control flow graph that joins all the methods of a single class, and thus allows pairing of definitions and uses that occur in different methods.

Figure 15.13 shows a partial intra-class control flow graph of class Model. Each method is modeled with a standard control flow graph (CFG), just as if it were an independent procedure, except that these are joined to allow paths that invoke different methods in sequence. To allow sequences of method calls, the class itself is modeled with a node class Model connected to the CFG of each method. Method Model includes two extra statements that correspond to the declarations of variables legalConfig and modelDB that are initialized when the constructor is invoked.

Sometimes definitions and uses are made through invocation of methods of other classes. For example method addComponent calls method contains of class componentDB. Moreover, some variables are structured, e.g., the state variable slot is a complex object. For the moment, we simply “unfold” the calls to external methods, and treat arrays and objects as if they were simple variables.

A test case to exercise a definition-use pair (henceforth DU pair) is a sequence of method invocations that starts with a constructor, includes the definition followed by the use without any intervening definition (a definition-clear path). A suite of test cases can be designed to satisfy a data-flow coverage criterion by covering all such pairs. In that case we say the test suite satisfies the all DU pairs coverage criterion.

\[ \Delta \text{ all DU pairs coverage criterion} \]

\[ ^{3}\text{We have simplified Figure 15.13 by omitting methods getHeightCm, getWidthCm, getDepthCm, and getWeightGm, since they depend only on the constructor and do not affect other methods. Exception handlers are excluded since they will be treated separately, as described in Section 15.12 below.} \]
Figure 15.13: A partial intra class control flow graph for the implementation of class Model in Figure 15.1.
Consider again the private variable `legalConfig` in class `Model`, Figures 15.1 and 15.2. There are two uses of `legalConfig`, both in method `isLegalConfiguration`, one in the `if` and one in the `return` statement; and there are several definitions, in methods `addComponent`, `removeComponent`, `checkConfiguration`, and in the constructor, which initializes `legalConfig` to `False`. The all DU pairs coverage criterion requires a test case to exercise each definition followed by each use of `legalConfig` with no intervening definitions.

Specifications do not refer to variable `legalConfig`, and thus do not directly consider method interactions through `legalConfig` or contribute to defining test cases to exercise such interactions. This is the case, for example, in the invocation of method `checkConfiguration` in `isLegalConfiguration`: The specification suggests that a single invocation of method `isLegalConfiguration` can be sufficient to test the interactions involving this method, while calls to method `checkConfiguration` in `isLegalConfiguration` indicate possible failures that may be exposed only after two calls of method `isLegalConfiguration`. In fact, a first invocation of `isLegalConfiguration` with value `True` for `legalConfig` implies a call to `checkConfiguration` and consequent new definitions of `legalConfig`. Only a second call to `isLegalConfiguration` would exercise the use of the new value in the `if` statement, thus revealing failures that may derive from bad updates of `legalConfig` in `checkConfiguration`.

The all DU pairs coverage criterion ensures every assignment to a variable is tested at each of the uses of that variable, but like other structural coverage criteria it is not particularly good at detecting missing code. For example, if the programmer omitted an assignment to `legalConfig`, there would be no DU pair connecting the missing assignment to the use. However, assignments to `legalConfig` are correlated with updates to `slots`, and all DU pairs coverage with respect to `slots` is likely to reveal a missing assignment to the boolean variable. Correlation among assignments to related fields is a common characteristic of the structure of object-oriented software.

Method calls and complex state variables complicate data-flow analysis of object-oriented software, as procedure calls and structured variables do in procedural code. As discussed in Chapters 6 and 13, there is no universal recipe to deal with inter-class calls. Test designers must find a suitable balance between costs and benefits.

A possible approach to deal with inter-class calls consists in proceeding incrementally following the dependence relation, as we did for functional inter-class testing. The dependence relation that can be derived from code may differ from the dependence relation derived from specifications. However, we can still safely assume that well designed systems present at most a small number of easily breakable cycles. The dependencies of the implementation and specification of class `Model` are the same and are shown in Figure 15.11.

Leaf classes of the dependence hierarchy can be analyzed in isolation by identifying definitions and uses of instance variables, as just shown. The data flow information collected on leaf classes can be summarized by marking methods that access but do not modify the state as `Inspectors`; methods that modify, but do not otherwise access the state, as `Modifiers`; and methods that both access and modify the state as `Inspector/Modifiers`.
When identifying inspectors, modifiers and inspector/modifiers, we consider the whole object state. Thus, we mark a method as inspector/modifier even if it uses just one instance variable and modifies a different one. This simplification is crucial to scalability, since distinguishing uses and definitions of each individual variable would quickly lead to an unmanageable amount of information while climbing the dependence hierarchy.

If methods contain more than one execution path, we could summarize the whole method as an inspector, modifier, or inspector/modifier, or we could select a subset of paths to be considered independently. A single method might include Inspector, Modifier, and Inspector/Modifier paths.

Once the data flow information of leaf classes has been summarized, we can proceed with classes that only use or contain leaf classes. Invocations of modifier methods and inspector/modifiers of leaf classes are considered as definitions. Invocations of inspectors and inspector/modifiers are treated as uses. When approximating inspector/modifiers as uses, we assume that the method uses the values of the instance variables for computing the new state. This is a common way of designing methods, but some methods may fall outside this pattern. Again, we trade precision for scalability and reduced cost.

We can then proceed incrementally analyzing classes that depend on only classes already analyzed, until we reach the top of the hierarchy. In this way, each class is always considered in isolation and the summary of information at each step prevents exponential growth of information, thus allowing large classes to be analyzed, albeit at a cost in precision.

Figure 15.14 shows the summary information for classes Slot, ModelDB and Model. The summary information for classes Slot and ModelDB can be used for computing structural coverage of class Model without unfolding the method calls. The summary information for class Model can be used to compute structural coverage for class Order without knowing the structure of the classes used by class Order. Method checkConfiguration is not included in the summary information because it is private. The three paths in checkConfiguration are included in the summary information of the calling method isLegalConfiguration.

While summary information is usually derived from child classes, sometimes it is useful to provide the same information without actually performing the analysis, as we have done when analyzing class Model. This is useful when we cannot perform data flow analysis on the child classes, e.g., when child classes are delivered as a closed component without source code, or are not available yet because the development is still in progress.

### 15.8 Oracles for Classes

Unit (intra-class) and integration (inter-class) testing require suitable scaffolding to exercise the classes under test (drivers and stubs) and to inspect the test results (oracles). Constructing stubs and drivers for object-oriented software is essentially similar to the same task for procedural programs, and as in procedural programs, stubs can be avoided to the extent that the order of test execution is aligned with the build order of...
Check the paths

<table>
<thead>
<tr>
<th>Class Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot() modifier</td>
</tr>
<tr>
<td>bind() modifier</td>
</tr>
<tr>
<td>unbind() modifier</td>
</tr>
<tr>
<td>isbound() inspector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class ModelDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelDB() modifier</td>
</tr>
<tr>
<td>getModel() inspector</td>
</tr>
<tr>
<td>findModel() inspector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model()</td>
</tr>
<tr>
<td>selectModel() modifier</td>
</tr>
<tr>
<td>deselectModel() modifier</td>
</tr>
<tr>
<td>addComponent() [1,2,8,9,10]</td>
</tr>
<tr>
<td>addComponent() [1,2,3,4,5,6,10]</td>
</tr>
<tr>
<td>addComponent() [1,2,3,4,7,10]</td>
</tr>
<tr>
<td>removeComponent() [1,2,3,4,5]</td>
</tr>
<tr>
<td>removeComponent() [1,2,4,5]</td>
</tr>
<tr>
<td>isLegalConfiguration() [1,2,3,[1,2,3,4,9],4]</td>
</tr>
<tr>
<td>isLegalConfiguration() [1,2,3,[1,2,3,4,5,6,7,4,9],4]</td>
</tr>
<tr>
<td>isLegalConfiguration() [1,2,3,[1,2,3,4,5,6,7,8,4,9],4]</td>
</tr>
<tr>
<td>isLegalConfiguration() [1,2,4]</td>
</tr>
</tbody>
</table>

Figure 15.14: Summary information for structural inter-class testing for classes Slot, ModelDB and Model. Lists of CFG nodes in square brackets indicate different paths, when methods include more than one part.
the software system. Oracles, however, can be more difficult to construct, owing to
encapsulation of object state.

The effect of executing a method or a whole sequence of methods in a test case is
not only the outputs produced, but also the state of the objects after execution. For ex-
ample, if method `deselectModel` of class `Model` does not clear the array `slots`,
it is erroneous, even if it produces the expected visible outputs. Thus, oracles need to
check the validity of both output and state. Unfortunately for the oracle builder, though,
the state of objects may not be directly accessible. For example, variable `slots` is pri-
vate and thus cannot be directly accessed by an oracle outside the class under test.

One approach to building oracles is to break the encapsulation, e.g., by modifying
the source code to allow inspection of private variables. If we violate encapsulation by
modifying code just for the purpose of testing, rather than leaving the modifications in
the actual delivered code, then we risk differences in behavior between what is tested
and what is used. We may mask faults, or we may inadvertently insert faults not present
in the original code, particularly if we make modifications by hand. Even a small
difference in performance can be important in a real-time system or in a multi-threaded
system sensitive to scheduler decisions.

Modifications that remain in the code, or (better) design rules that require program-
ners to provide observability interfaces, avoid discrepancies between the production
code and the tested code. This is a particularly attractive option if the interface for
observing object state can be separated from the main class, as one can for example
do with a C++ friend class.\footnote{A “friend” class in C++ is permitted direct access to private variables in another class. There is no
direct equivalent in Java or SmallTalk, although in Java one could obtain a somewhat similar effect by using
package visibility for variables and placing oracles in the same package.} An observability interface can be a collection of observer
methods, or a single method to produce a representation of the full object state. Often
an interface that produces a readable, canonical representation of an object value will
be useful in debugging as well as in testing.

Add exercise: canonical representation as abstract values

A second alternative is not to reveal the internal state of an object per se, but to
provide a way of determining whether two objects are equivalent. Here “equivalent”
does not mean that the internal states of two objects are identical, but that they represent
the same abstract value. For example, we might consider the Java `Vector` class as
representing a sequence. If so, then not only might two vectors with different capacities
be considered equivalent, but we might even consider a vector object and a linked list
object to be equivalent if they contain the same elements in the same order.

An (abstract) check for equivalence can be used in a test oracle if test cases exer-
cise two sequences of method calls which should (or should not) produce the same
object state. Comparing objects using this `equivalent scenarios` approach is particu-
larly suitable when the classes being tested are an instance of a fairly simple abstract
data type, such as a dictionary structure (which includes hash tables, search trees, etc.),
or a sequence or collection.

Table 15.3 shows two sequences of method invocations, one equivalent and one
non-equivalent to test case `TC_E` in Table ??.

\begin{table}
\centering
\begin{tabular}{|c|c|}
\hline
Method & Sequence 1 \tabularnewline
\hline
\hline
Method A & \dots \text{method A} \dots \text{method A} \text{clears} \text{slots} \text{array} \text{at} \text{line} x \text{in} \text{class} \text{Model} \text{with} \text{private} \text{variable} \text{slots}\text{.} \text{However,} \text{method} \text{deselectModel} \\
\hline
Method B & \dots \text{method B} \dots \text{method B} \text{does} \text{not} \text{clear} \text{the} \text{array} \text{slots} \text{at} \text{line} y \text{in} \text{class} \text{Model}\text{.} \text{Result}\text{.} \text{The} \text{sequence} \text{is} \text{invalid}\text{.} \\
\hline
Method C & \dots \text{method C} \dots \text{method C} \text{does} \text{not} \text{clear} \text{the} \text{array} \text{slots} \text{at} \text{line} z \text{in} \text{class} \text{Model}\text{.} \text{Result}\text{.} \text{The} \text{sequence} \text{is} \text{invalid}\text{.} \\
\hline
\end{tabular}
\caption{Equivalent scenarios}
\end{table}

The equivalent sequence is obtained by
removing “redundant” method invocations, i.e., invocations that bring the system to a
polymorphism and dynamic binding

Test Case $TC_E$
- selectModel(M1)
- addComponent(S1,C1)
- addComponent(S2,C2)
- isLegalConfiguration()
- deselectModel()
- selectModel(M2)
- addComponent(S1,C1)
- isLegalConfiguration()

Scenario $TC_{E_1}$
- selectModel(M2)
- addComponent(S1,C1)
- isLegalConfiguration()

Scenario $TC_{E_2}$
- selectModel(M2)
- addComponent(S1,C1)
- addComponent(S2,C2)
- isLegalConfiguration()

EQUIVALENT

How can this refer to a single test case $TC_E$?

Table 15.3: Equivalent and non-equivalent scenarios (invocation sequences) for test case $TC_E$ of Table ??.

previous state. In the example, method deselectModel cancels the effect of previous invocations of method selectModel and addComponent. The non-equivalent sequence is obtained by selecting a legal subset of method invocations that bring the object to a different state.

Producing equivalent sequences is often quite simple. While finding non-equivalent sequences is even easier, choosing a few good ones is difficult. One approach is to hypothesize a fault in the method that “generated” the test case, and create a sequence that could be equivalent if the method contained that fault. For example, test case $TC_E$ was designed to test method deselectModel. The non-equivalent sequence of Table 15.3 leads to a state that could be produced if method deselectModel did not clear all slots, leaving component C2 bound to slot S2 in the final configuration.

One sequence of method invocations is equivalent to another if the two sequences lead to the same object state. This does not necessarily mean that their concrete representation is bit-for-bit equal. For example, method addComponent binds a component to a slot by creating a new Slot object (Figure 15.2). Starting from two identical Model objects, and calling addComponent on both with exactly the same parameters, would result in two objects which represent the same information but which nonetheless would contain references to distinct Slot objects. The default equals method inherited from class Object, which makes a bit-for-bit comparison, would consider them inequivalent. A good practice is to add a suitable observer method to a class (e.g., by overriding the default equals method in Java).

Resumed here on Friday

15.9 Polymorphism and Dynamic Binding

Limited use of polymorphism and dynamic binding is easily addressed by unfolding polymorphic calls, considering each method that can be dynamically bound to each polymorphic call. Complete unfolding is impractical when many references may each
Figure 15.15: A method call in which the method itself and two of its parameters can be dynamically bound to different classes.

Consider, for example, the code fragment in Figure 15.15. Object Account may be an instance of any of the classes USAccount, UKAccount, EUAccount, JPAccount, or OtherAccount. Method validateCredit can be dynamically bound to methods validateCredit of any of the classes EduCredit, BizCredit, or IndividualCredit, each implementing different credit policies. Parameter creditCard may be dynamically bound to VISA Card, AmExp Card, or Chipmunk Card, each with different characteristics. Even in this simple example, replacing the calls with all possible instances results in 45 different cases (5 possible types of account \* 3 possible types of credit \* 3 possible credit cards).

The explosion in possible combinations is essentially the same combinatorial explosion encountered if we try to cover all combinations of attributes in functional testing, and the same solutions are applicable. The combinatorial testing approach presented in Chapter 10 can be used choose a set of combinations that covers each pair of possible bindings (e.g., Business account in Japan, Education customer using Chipmunk Card), rather than all possible combinations (Japanese business customer using Chipmunk card). Table 15.4 shows 15 cases that cover all pairwise combinations of calls for the example of Figure 15.15.

The combinations above were of dynamic bindings in a single call. Bindings in a sequence of calls can also interact. Consider, for example, method getYTD Purchased of class Account shown in Figure 15.4 on page 330, which computes the total yearly purchase associated with one account to determine the applicable discount. Chipmunk offers tiered discounts to customers whose total yearly purchase reaches a threshold, considering all subsidiary accounts.

The total yearly purchase for an account is computed by method getYTD Purchased, which sums purchases by all customers using the account and all subsidiaries. Amounts are always recorded in the local currency of the account, but getYTD Purchased sums the purchases of subsidiaries even when they use different currencies, e.g., when some are bound to subclass USAccount and others to EUAccount. The intra- and inter-class testing techniques presented in the previous section may fail to reveal this type of fault. The problem can be addressed by selecting test cases that cover combinations of polymorphic calls and bindings. To identify sequential combinations of bindings, we must first identify individual polymorphic calls and binding sets, and then select possible sequences.

Let us consider for simplicity only the method getYTD Purchased. This method
Table 15.4: A set of test case specifications that cover all pairwise combinations of the possible polymorphic bindings of Account, Credit, and creditCard.

is called once for each customer and once for each subsidiary of the account and in both cases can be dynamically bound to methods belonging to any of the subclasses of Account (UKAccount, EUAccount, etc.) At each of these calls, variable totalPurchased is used and changed, and at the end of the method it is used twice more (to set an instance variable and to return a value from the method).

Data flow analysis may be used to identify potential interactions between possible bindings at a point where a variable is modified and points where the same value is used. Any of the standard data-flow testing criteria could be extended to consider each possible method binding at the point of definition and the point of use, i.e., a single Definition-Use pair becomes $n \times m$ pairs if the point of definition can be bound in $n$ ways and the point of use can be bound in $m$ ways. If this is impractical, a weaker but still useful alternative is to vary both bindings independently, which results in $m$ or $n$ pairs (whichever is greater) rather than their product. Note that this weaker criterion would be very likely to reveal the fault in getYTD Purchased, provided the choices of binding at each point are really independent rather than going through the same set of choices in lock-step. In many cases, binding sets are not mutually independent, so the selection of combinations is limited.

15.10 Inheritance

Inheritance does not introduce new classes of faults except insofar as it is associated with polymorphism and dynamic binding, which we have discussed above, and exception handling, which is discussed below. It does provide an opportunity for optimization by re-using test cases and even test executions. Subclasses share methods with ancestors. Identifying which methods do not need to be retested and which test cases
Testing Object-Oriented Software

356

can be reused may significantly reduce testing effort.

Methods of a subclass can be categorized as

**New:** if they are newly defined in the subclass, i.e., they do not occur in the ancestor.

New methods include those with the same name but different parameters than methods in ancestor classes.

**Recursive:** if they are inherited from the ancestor without change, i.e., they occur only in the ancestor.

**Redefined:** if they are overridden in the subclass, i.e., they occur both in the subclass.

**Abstract new:** if they are newly defined and abstract in the subclass.

**Abstract recursive:** if they are inherited from the ancestor, where they are abstract.

**Abstract redefined:** if they are re-defined in the subclass, and they are abstract in the ancestor.

When testing a base class, i.e., a class that does not specialize a previously tested class, we can summarize the testing information in a simple table that indicates the sets of generated and executed test cases. Such a table is called a **testing history**.

In general we will have four sets of test cases for a method: **intra-class functional**, **intra-class structural**, **inter-class functional**, and **inter-class structural**. For methods that do not call methods in other classes, we will have only **intra-class test cases**, since no integration test focuses on such methods. For abstract methods, we will only have **functional test cases**, since we do not have the code of the method. Each set of test cases is marked with a flag that indicates whether the test set can be executed.

Table 15.5 shows a testing history for class **LineItem**, whose code is shown in Figure 15.16. Methods **validItem**, **getWeightGm**, **getHeightCm**, **getWidthCm**, and **getDepthCm** are abstract and do not interact with external classes, thus we only have intra-class functional test cases that cannot be directly executed. Method **getUnitPrice** is abstract, but from the specifications (not shown here) we can infer that it interacts with other classes, thus we have both intra- and inter-class functional test cases. Both the constructor and method **getExtendedPrice** are implemented and interact with other classes (**Order** and **AccountType**, respectively) and thus we have all four sets of test cases.

New and abstract new methods need to be tested from scratch, thus we need to derive the needed test cases and execute them. We report the testing activity in the testing history of the new class by adding a new raw and new test cases. Recursive and abstract recursive methods do not need to be retested, thus the old test sets are copied into the new table and marked as not-to-be-executed. Redefined and abstract redefined methods must be retested, thus we add new test cases and we mark them to be executed.

Table 15.6 shows the testing history for class **CompositeItem** that specializes class **LineItem**. The code of class **CompositeItem** shown in Figure 15.17. Class **CompositeItem** adds a constructor, and thus we add a line to the testing history that indicates the four sets of test cases to be added and executed. It redefines method **getUnitPrice**, which was virtual in class **LineItem**: the functional test cases

Draft version produced February 22, 2006
/** One line item of a customer order (abstract). */
public abstract class LineItem{

/** The order this LineItem belongs to. */
protected Order order;

/** Constructor links item to owning order. Must call in subclasses. */
public LineItem(Order _order) { order = _order; }

/** Stock-keeping unit (sku) is unique key to all product databases. */
public String sku;

/** Number of identical units to be purchased. */
public int units=1;

/** Has this line item passed all validation tests? */
public abstract boolean validItem();

/** Price of a single item. */
public abstract int getUnitPrice(AccountType accountType);

/** Extended price for number of units */
public int getExtendedPrice(AccountType accountType)
{   return units * this.getUnitPrice(accountType);   }

// Dimensions for packing and shipping (required of all top-level items)
/** Weight in grams */
public abstract int getWeightGm();
/** Height in centimeters */
public abstract int getHeightCm();
/** Width in Centimeters. */
public abstract int getWidthCm();
/** Depth in Centimeters */
public abstract int getDepthCm();
}

Figure 15.16: Part of a Java implementation of the abstract class LineItem.
Table 15.5: Testing history for class LineItem

<table>
<thead>
<tr>
<th>method</th>
<th>Intra funct</th>
<th>Intra struct</th>
<th>Inter funct</th>
<th>Inter struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>LineItem</td>
<td>(TS_{L11} \cdot Y)</td>
<td>(TS_{L12} \cdot Y)</td>
<td>(TS_{L13} \cdot Y)</td>
<td>(TS_{L14} \cdot Y)</td>
</tr>
<tr>
<td>validItem</td>
<td>(TS_{vI1} \cdot N)</td>
<td>(TS_{vI2} \cdot N)</td>
<td>(TS_{vI3} \cdot N)</td>
<td>(TS_{vI4} \cdot N)</td>
</tr>
<tr>
<td>getUnitPrice</td>
<td>(TS_{gUP1} \cdot N)</td>
<td>(TS_{gUP2} \cdot Y)</td>
<td>(TS_{gUP3} \cdot N)</td>
<td>(TS_{gUP4} \cdot Y)</td>
</tr>
<tr>
<td>getExtendedPrice</td>
<td>(TS_{gXP1} \cdot Y)</td>
<td>(TS_{gXP2} \cdot Y)</td>
<td>(TS_{gXP3} \cdot Y)</td>
<td>(TS_{gXP4} \cdot Y)</td>
</tr>
<tr>
<td>getWeightGm</td>
<td>(TS_{gWG1} \cdot N)</td>
<td>(TS_{gWG2} \cdot N)</td>
<td>(TS_{gWG3} \cdot N)</td>
<td>(TS_{gWG4} \cdot N)</td>
</tr>
<tr>
<td>getHeightCm</td>
<td>(TS_{gHC1} \cdot N)</td>
<td>(TS_{gHC2} \cdot N)</td>
<td>(TS_{gHC3} \cdot N)</td>
<td>(TS_{gHC4} \cdot N)</td>
</tr>
<tr>
<td>getWidthCm</td>
<td>(TS_{gWC1} \cdot N)</td>
<td>(TS_{gWC2} \cdot N)</td>
<td>(TS_{gWC3} \cdot N)</td>
<td>(TS_{gWC4} \cdot N)</td>
</tr>
<tr>
<td>getDepthCm</td>
<td>(TS_{gDC1} \cdot N)</td>
<td>(TS_{gDC2} \cdot N)</td>
<td>(TS_{gDC3} \cdot N)</td>
<td>(TS_{gDC4} \cdot N)</td>
</tr>
</tbody>
</table>

Legend: \((TS_I, B)\) refers to test set \(I\), to be executed if \(B = Y\).
\((-\quad,-\quad)\) means no applicable tests.

Table 15.6: Testing history for class CompositeItem. New test sets are marked with a prime.

<table>
<thead>
<tr>
<th>method</th>
<th>Intra funct</th>
<th>Intra struct</th>
<th>Inter funct</th>
<th>Inter struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>LineItem</td>
<td>(TS_{L11} \cdot N)</td>
<td>(TS_{L12} \cdot N)</td>
<td>(TS_{L13} \cdot N)</td>
<td>(TS_{L14} \cdot N)</td>
</tr>
<tr>
<td>validItem</td>
<td>(TS_{vI1} \cdot N)</td>
<td>(TS_{vI2} \cdot N)</td>
<td>(TS_{vI3} \cdot N)</td>
<td>(TS_{vI4} \cdot N)</td>
</tr>
<tr>
<td>getUnitPrice</td>
<td>(TS_{gUP1} \cdot N)</td>
<td>(TS_{gUP2} \cdot N)</td>
<td>(TS_{gUP3} \cdot N)</td>
<td>(TS_{gUP4} \cdot N)</td>
</tr>
<tr>
<td>getExtendedPrice</td>
<td>(TS_{gXP1} \cdot N)</td>
<td>(TS_{gXP2} \cdot N)</td>
<td>(TS_{gXP3} \cdot N)</td>
<td>(TS_{gXP4} \cdot N)</td>
</tr>
<tr>
<td>getWeightGm</td>
<td>(TS_{gWG1} \cdot N)</td>
<td>(TS_{gWG2} \cdot N)</td>
<td>(TS_{gWG3} \cdot N)</td>
<td>(TS_{gWG4} \cdot N)</td>
</tr>
<tr>
<td>getHeightCm</td>
<td>(TS_{gHC1} \cdot N)</td>
<td>(TS_{gHC2} \cdot N)</td>
<td>(TS_{gHC3} \cdot N)</td>
<td>(TS_{gHC4} \cdot N)</td>
</tr>
<tr>
<td>getWidthCm</td>
<td>(TS_{gWC1} \cdot N)</td>
<td>(TS_{gWC2} \cdot N)</td>
<td>(TS_{gWC3} \cdot N)</td>
<td>(TS_{gWC4} \cdot N)</td>
</tr>
<tr>
<td>getDepthCm</td>
<td>(TS_{gDC1} \cdot N)</td>
<td>(TS_{gDC2} \cdot N)</td>
<td>(TS_{gDC3} \cdot N)</td>
<td>(TS_{gDC4} \cdot N)</td>
</tr>
</tbody>
</table>

The testing history approach reduces the amount of tests to be executed, but requires extra effort of keeping track of testing activities. Effort is repaid mainly when it is possible to avoid designing new test cases, but when the cost of executing test cases is high (e.g., because the test requires interaction with an external device or a human) the savings in test execution cost can also be significant. If the cost of executing test cases is negligible, it may be cheaper to simply retest all classes regardless of the tests executed on the ancestors.

### 15.11 Genericity

Generics, also known as parameterized types or (in C++) as templates, are an important tool for building re-usable components and libraries. A generic class (say,
package Orders;
import Accounts.AccountType;
import Prices.Pricelist;
import java.util.*;

/**
 * A composite line item includes a "wrapper" item for the whole
 * bundle and a set of zero or more component items.
 */
public abstract class CompositeItem extends LineItem {

/**
 * A composite item has some unifying name and base price
 * (which might be zero) and has zero or more additional parts,
 * which are themselves line items.
 */
private Vector parts = new Vector();

/**
 * Constructor from LineItem, links to an encompassing Order.
 */
public CompositeItem(Order _order) {
    super(_order);
}

public int getUnitPrice(AccountType accountType) {
    Pricelist prices = new Pricelist();
    int price = prices.getPrice(sku, accountType);
    for(Enumeration e = parts.elements(); e.hasMoreElements(); )
    {
        LineItem i = (LineItem) e.nextElement();
        price += i.getUnitPrice(accountType);
    }
    return price;
}

}

Figure 15.17: Part of a Java implementation of class CompositeItem.
linked lists) is designed to be instantiated with many different parameter types (e.g., `LinkedList<String>` and `LinkedList<Integer>`). We can test only instantiations, not the generic class itself, and we may not know in advance all the different ways a generic class might be instantiated.

A generic class is typically designed to behave consistently over some set of permitted parameter types. Therefore the testing (and analysis) job can be broken into two parts: Showing that some instantiation is correct, and showing that all permitted instantiations behave identically.

Testing a single instantiation raises no particular problems, provided we have source code for both the generic class and the parameter class. Roughly speaking, we can design test cases as if the parameter were copied textually into the body of the generic class.

Consider first the case of a generic class that does not make method calls on, nor access fields of, its parameters. Ascertaining this property is best done by inspecting the source code, not by testing it. If we can nonetheless conjecture some ways in which the generic and its parameter might interact (e.g., if the generic makes use of some service that a parameter type might also make use of, directly or indirectly), then we should design test cases aimed specifically at detecting such interaction.

Gaining confidence in an unknowable set of potential instantiations becomes more difficult when the generic class does interact with the parameter class. For example, Java (since version 1.5) has permitted a declaration like this:

```java
class PriorityQueue<E extends Comparable> { ... }
```

The generic `PriorityQueue` class will be able to make calls on the methods of interface `Comparable`. Now the behavior of `PriorityQueue<E>` is not independent of `E`, but it should be dependent only in certain very circumscribed ways, and in particular it should behave correctly whenever `E` obeys the requirements of the contract implied by `Comparable`.

The contract imposed on permitted parameters is a kind of specification, and specification-based (functional) test selection techniques are an appropriate way to select representative instantiations of the generic class. For example, if we read the interface specification for `java.lang.Comparable`, we learn that most but not all classes that implement `Comparable` also satisfy the rule

```
(x.compareTo(y) == 0) == (x.equals(y))
```

Explicit mention of this condition strongly suggests that test cases should include instantiations with classes that do obey this rule (class `String`, for example) and others that do not (e.g., class `BigDecimal` with two `BigDecimal` values 4.0 and 4.00).

### 15.12 Exceptions

Programs in modern object-oriented languages use exceptions to separate handling of error cases from the primary program logic, thereby simplifying normal control flow. Exceptions also greatly reduce a common class of faults in languages without...
exception-handling constructs. One of the most common faults in C programs, for example, is neglecting to check for the error indications returned by a C function. In a language like Java, an exception is certain to interrupt normal control flow.

The price of separating exception handling from the primary control flow logic is introduction of implicit control flows. The point at which an exception is caught and handled may be far from the point at which it is thrown, and moreover the association of exceptions with handlers is dynamic. In most object-oriented languages and procedural languages that provide exception handling, an exception propagates up the stack of calling methods until it reaches a matching handler.

Since exceptions introduce a kind of control flow, one might expect that it could be treated like other control flow in constructing program models and deriving test cases. However, treating every possible exception this way would create an unwieldy control flow graph accounting for potential exceptions at every array subscript reference, every memory allocation, every cast, etc., and these would be multiplied by matching them to every handler that could appear immediately above them on the call stack. Worse, many of these potential exceptions are actually impossible, so the burden would not be just in designing test cases for each of them but in deciding which can actually occur. It is more practical to consider exceptions separately from normal control flow in test design.

We can dismiss from consideration exceptions triggered by program errors signaled by the underlying system (subscript errors, bad casts, etc.), since exercising these exceptions adds nothing to other efforts to prevent or find the errors themselves. If a method A throws an exception that indicates a programming error, we can take almost the same approach. However, if there are exception handlers for these program error exceptions, such as we may find in fault-tolerant programs or in libraries that attempt to maintain data consistency despite errors in client code, then it is necessary to test the error recovery code (usually by executing it together with a stub class with the programming error). This is different and much less involved than testing the error recovery code coupled with every potential point at which the error might be present in actual code.

Exceptions that indicate abnormal cases but not necessarily program errors, e.g., exhaustion of memory or premature end-of-file, require special treatment. If the handler for these is local (e.g., a Java try block with an exception handler around a group of file operations), then the exception handler itself requires testing. Whether to test each individual point where exceptions bound to the same handler might be raised (e.g., each individual file operation within the same try block) is a matter of judgment.

The remaining exceptions are those that are allowed to propagate beyond the local context in which they are thrown. For example, suppose method A makes a call to method B, within a Java try block with an exception handler for exceptions of class E. Suppose B has no exception handler for E, and makes a call to method C which throws E. Now the exception will propagate up the chain of method calls until it reaches the handler in A. There could be many such chains, which depend in part on overriding of inherited methods, and it is difficult (sometimes even impossible) to determine all and only the possible pairings of points where an exception is thrown with handlers in other methods.
Since testing all chains through which exceptions can propagate is impractical, it is best to make it unnecessary. A reasonable design rule to enforce is that, if a method propagates an exception without catching it, the method call should have no other effect. If it is not possible to ensure that method execution interrupted by an exception has no effect, then an exception handler should be present (even if it propagates the same exception by throwing it again). Then, it should suffice to design test cases to exercise each point at which an exception is explicitly thrown by application code, and each handler in application code, but not necessarily all their combinations.

**Further Reading**

Many recent books on software testing and software engineering address object-oriented software to at least some degree. The most complete book-length account of current methods is Binder’s *Testing Object Oriented Systems* [Bin00].

Structural state-based testing is discussed in detail by Buy, Orso and Pezzè [BOP00]. The data flow approach to testing software with polymorphism and dynamic binding was initially proposed by Orso [Ors98]. Harrold, McGregor, and Fitzpatrick [HMF92] provide a detailed discussion of the use of testing histories for selecting test cases for subclasses.

Thévenod-Fosse and Waeselynck describe statistical testing using statechart specifications [TFW93]. An excellent paper by Doong and Frankl [DF94] introduces equivalent scenarios. Although Doong and Frankl discuss their application with algebraic specifications (which are not much used in practice), the value of the approach does not hinge on that detail.

**Related topics**

Basic functional and structural testing strategies are treated briefly here, and readers who have not already read Chapters 10, 11, and 12 will find there a more thorough presentation of the rationale and basic techniques for those approaches. Chapters 13 and 14 likewise present the basic data flow and model-based testing approaches in more detail. As integration testing progresses beyond small clusters of classes to major sub-systems and components, the inter-class testing techniques described in this chapter become less relevant, and component testing techniques presented in Chapter 21 become more important. The system and acceptance testing techniques described in Chapter 22 are as appropriate to object-oriented software as they are to mixed and purely procedural software systems.