telecommunications systems [Turner 93]. Statecharts were first presented in [Harel 87] and [Harel 88] and were reinterpreted for object-oriented development in [Harel+97]. They have been the subject of many research papers. Leveson presents design considerations to reduce risks and the use of state models to analyze safety properties [Leveson 95].

### State-based Software Testing

A pragmatic and complete introduction to state-based test design can be found in [Beizer 90] and [Beizer 95]. Considerations for test design with statecharts are developed in [Marick 95]. State-based testing has been extensively studied and applied for telecommunication software. The W-method and *n*-switch cover were developed to test software switching systems [Chow 78]. Empirical studies of the size and fault-finding ability of state-based testing strategies appear in [Sidhu+89] and [Fujiwara+91]. Holzmann provides a comprehensive and highly readable discussion of state-based verification [Holzmann 91]. The January 1992 issue of *IEEE Software* magazine focuses on verification of telecommunication protocols. Bernhard provides a concise comparative synopsis of formal state-based testing models [Bernhard 94]. The main results presented in this paper are three sequence selection algorithms that have the same fault-revealing power as the W-method but require significantly shorter signature sequences.

### State-based Hardware Testing

State models are a fundamental tool for the design and test of digital logic circuits. See, for example, [Kohavi 78, Fujiwara 85, Abramovici+90, Devadas+94, and Comer 95].

### Mathematical Theory

The Moore [Moore 56] and Mealy [Mealy 55] papers are seminal in automata theory. Gill's extensive application-oriented study of the formal properties of state machines remains useful [Gill 62]. Many computer science textbooks present finite state automata in the context of computing theory—for example, [Lewis+81, Hopcroft+79]. Extensions to formal models that represent product machines and guards are developed in [Arnold 94]. A formal analysis of the reduction in the state space in product machines owing to statechart abstractions is developed in [Drusinsky+94].

# Chapter 8 A Tester's Guide to the UML

No realization without representation.
*Robert Ashenhurst*

**Overview**

This chapter considers the Unified Modeling Language (UML) from a testing perspective. The notation and semantics of each UML diagram are summarized. Considerations for developing test-ready UML diagrams are discussed. For some diagrams, testability extensions are presented. The chapter also gives a generic test strategy for graphs and relations that is used to identify generic test requirements for each UML diagram.

## 8.1 Introduction

### 8.1.1 The UML as a Test Model

At its most abstract, test design is the identification, analysis, and demonstration of relationships that must hold for the system under test. A test model captures these relationships. This model should be unambiguous, related to probable faults, and support automated production of compact test suites. As the preceding chapters have shown, combinational logic and state machines are powerful and testable models that meet these requirements.

The Unified Modeling Language (UML) is a notational syntax for expressing object-oriented models. It merges the OOA/D notations developed by

Booch [Booch 91], Jacobson et al. [Jacobson+92], and Rumbaugh et al. [Rumbaugh+91]. It has become a de facto standard for object-oriented modeling. Hundreds of publications about developing UML models are available, so we need no discussion of modeling technique here.

The UML is not a methodology. It does not prescribe technique, results, or process. It purposely places few restrictions on usage of the notation. As with natural languages, this flexibility allows both precision and muddle. Developing a rigorous application model with the UML is possible. However, fragmentary, incomplete, inconsistent, and ambiguous models are easily produced without violating any of the UML's requirements.[1]

If a model is produced at all during object-oriented development, it is likely to be a collection of UML diagrams. UML models are an important source of information for test design that should not be ignored. This chapter presents three main test development techniques:

- It summarizes the elements of UML diagrams and explains how these elements can be used for test design.
- It shows how to develop UML models with sufficient information to produce test cases.
- It presents general test requirements for each diagram, which can be used to develop application-specific test cases.

The chapter concludes with an advanced section that explains the graph relation test design model used to generate the generic test requirements.

### 8.1.2 Relational Testing Strategy

Lines between UML symbols represent an instance of a general relationship recognized within the UML. For example, an association line between two class symbols can show that an instance of one class depends on the other. Application-specific models are constructed from these built-in relationships. For example, an association line between the class `Person` and the class `Dog` shows that an instance of `Dog` may not exist unless a corresponding instance of `Person` is also present.

1. There are general and specific elements of the UML that I believe represent fundamentally bad choices for a software engineering notation. Despite these limitations, the UML can support development of test-ready models. Work-arounds for most of these problems are suggested in this chapter and elsewhere.



Such built-in relationships have corresponding generic test requirements that can be identified by applying a relational test strategy to each UML diagram. The mathematics of this approach are discussed at the end of the chapter, which explains the columns labeled *R*, *S*, and *T* in each set of generic test requirements.

You can develop a test suite for an application modeled with UML diagrams in two ways. First, you can interpret the generic test requirements for your application and develop the indicated tests. For example, we would test implementation of the existence dependency by trying to establish it, and then trying to create inconsistent instances—for example, a `Dog` with no human owner. Second, you can apply the related test design pattern. Figure 8.1 summarizes the intersection of the UML and the test design patterns in this book.

The generic test requirements can also serve as a test design and execution checklist. You use the following criterion to compare your test suite and your application model:

> For each UML diagram representing requirements of the SUT and for each application-specific instance of a UML relationship type that appears in that diagram, at least one actual test achieves the generic test requirements for an implementation instance of that application-specific relationship.

If nothing else, this analysis will establish traceability from each element of your application model to the implementation under test and your test suite. This exercise nearly always reveals omissions and inconsistencies.

## 8.2 General-purpose Elements

### 8.2.1 Organization and Annotation

General-purpose elements of the UML are used to organize diagrams and express details.

#### Package Diagrams and Packages

A *package* is a group of UML diagrams and diagram elements of any kind, including other packages. A *package diagram* shows the organization of packages. A large model is typically organized as a hierarchy of package diagrams. A package does not necessarily correspond to an Ada 95 or Java package or to any other implementation scoping mechanism (although it may). The symbol for a package is a large rectangle with a small box in the upper-left corner,

| Test Design Pattern | | Use Case | Class | Sequence | Activity | State-chart | Collaboration | Component | Deployment |
|---|---|---|---|---|---|---|---|---|---|
| Method scope | Category-Partition | ☆ | | ☆ | ☆ | | ☆ | | |
| | Combinational Function | ☆ | | ☆ | ☆ | | | | |
| | Recursive Function | | | ☆ | | | ☆ | | |
| | Polymorphic Message | | ☆ | ☆ | | | ☆ | | |
| Class scope | Invariant Boundaries | ☆☆ | ☆☆ | ☆☆ | | | ☆ | | |
| | Nonmodal Class | | | ☆ | | | | | |
| | Modal Class | | | | | ★ | | | |
| | Quasi-modal Class | | | | | ☆ | | | |
| Class scope integration | Small Pop | | | | | | | | |
| | Alpha-Omega Cycle | | ☆☆ | | | | | | |
| Flattened class scope | Polymorphic Server | | ☆ | ☆ | | | ☆ | | |
| | Modal Hierarchy | | ☆ | ☆ | | ★ | ☆ | | |
| Reusable components | Abstract Class | | ☆☆ | | | | | | |
| | Generic Class | | ☆☆ | | | | | | |
| | New Framework | ☆☆ | ☆☆ | ☆ | ☆ | ☆ | ☆ | ☆ | ☆ |
| | Popular Framework | ☆☆ | ☆☆ | ☆ | ☆ | ☆ | ☆ | ☆ | ☆ |
| Subsystem | Class Associations | | ★ | | | | | | |
| | Round-trip Scenario | | | ★ | ☆ | ☆ | ☆ | | |
| | Controlled Exception | | | ☆ | ☆ | | ☆ | | |
| | Mode Machine | | | | ☆ | ★ | | | |
| Integration | Big Bang | ☆ | | ☆ | | | ☆ | ☆ | ☆ |
| | Bottom-up | | | | | | ☆ | ☆ | ☆ |
| | Top-down | | | ☆ | | | ☆ | ☆ | ☆ |
| | Collaborations | | | ☆ | | | ★ | ☆ | ☆ |
| | Backbone | ☆ | | ☆ | | | | ☆ | ☆ |
| | Layers | ☆ | | ☆ | | | | ☆ | ☆ |
| | Client/Server | ☆ | | ☆ | | | | ☆ | ☆ |
| | Distributed Services | ☆ | | ☆ | | | | ☆ | ☆ |
| | High-frequency | ☆ | | ☆ | | | | ☆ | ☆ |
| Application scope | Extended Use Cases | ★ | | ☆ | | | | ☆ | ☆ |
| | Covered in CRUD | ★ | | | | | | ☆ | ☆ |
| | Allocate Tests by Profile | ★ | | | | | | ☆ | |
| Regression test | Retest All | | | | | | | ☆ | ☆ |
| | Retest Risky Use Cases | ☆ | | ☆ | | | | ☆ | ☆ |
| | Retest by Profile | ☆ | | | | | | ☆ | ☆ |
| | Retest Changed Code | ☆ | | | | | | ☆ | ☆ |
| | Retest Within Firewall | | | | | | ☆ | ☆ | ☆ |

★ Pattern explicitly based on this diagram, ☆ Pattern may be useful for testing elements of this diagram, ☆ Diagram may be useful when using this pattern

FIGURE 8.1 UML diagrams and test design patterns.

suggesting a file folder. The package name appears in the tab (if the content is displayed) or in the content box if the content is suppressed.

Packages may be marked as public, protected, or private. A dashed arrow shows that one package depends on another package. A solid line with an unfilled triangle on the end indicates inheritance.

Each element of a model is "owned" by only one package. The ownership relationship is a tree. Packages, however, can refer to ("depend on") any other package. They are the basic unit of organization for configuration control, storage, and access control.

### Keywords and Stereotypes

UML *keywords* are labels that distinguish representation elements that use the same symbol. Keywords are reserved; that is, these terms are predefined for all UML models. For example, the classifier symbol—a rectangle with compartments—can be used to represent a type, a class, a meta-class, or an object. A label bracketed with guillemets defines the kind of classifier as a «type», «class», «metaclass», or «object». A *stereotype* is a user-defined keyword.

### Expressions, Constraints, and Comments

An *expression* is a string from an executable language that can be evaluated to produce a result. A *constraint* is a predicate expression on model elements. It is shown as text enclosed in braces—for example, `{x == y && !isEmpty}`. A constraint expresses a relationship within the system under study. Although the interpretation of predefined UML constraints is fixed, the syntax and interpretation of user-specified constraints are user-defined. Constraints may be written in natural language, mathematical expressions, code, or Object Constraint Language (OCL). A constraint written in natural language is a called a *comment*.

### Notes

A *note* is shown as a box with a dog-eared corner. It may be connected to another diagram element or it may be free-standing. The note contains a textual description.

### Element Properties

The *properties* of an element can be displayed enclosed in braces and designated by a *tag* expression—for example, `{ LastRevDate=1999021, VersionId=0.2 }`.




FIGURE 8.2 General UML symbols.

Many properties are suppressed in typical presentation diagrams. Figure 8.2 shows a package diagram and a note.

The *UML Specification* states that an edge between diagram elements may be composed of one or more segments and that "A connected sequence of segments is called a path" [OMG 98, 3-52]. That is, a UML "path" is simply a line between two graph elements. The term *path* has a long-established, well-defined, and different meaning in graph theory, testing, and computer science. In this book, the graph-theoretic sense of **path** is used: a set of nodes and edges for which an unambiguous property holds. The UML also uses "node" as a diagram element that represents a processor (see Section 8.10, Deployment Diagram). In the following discussion **node**, is used in the graph-theoretic sense: an element of a graph connected by edges. An **edge** is an abstract connection between abstract nodes, and a **line** refers to one or more contiguous line segments rendered on a diagram.

## 8.2.2 Object Constraint Language

The OCL is a modeling language that can express relationships and properties of modeled elements [Warmer+98]. It is based on Syntropy [Cook+94] and aims to make the power of specification languages like Z accessible to software developers who do not have a mathematical background. OCL uses keywords that have been chosen for their understandability without sacrificing any precision. The meta-model for the UML is expressed in OCL.

OCL can be used to define (as a constraint or expression) basic relationships that are not easily shown with UML graphs:

- Class and type invariants
- Type invariant for stereotypes
- Preconditions and postconditions on operations (methods)

- All forms of guards
- Specification of the result of a computation in a nonprocedural manner[2]

Perhaps the most interesting application of OCL is to express the navigation of a class model. Navigation expressions define how a network of class relationships is traversed. They are similar to SQL, not in syntax, but in concept.

OCL can be used to express all of the state invariants discussed in Chapter 7 and the method and class assertions covered in Chapter 17. A UML application model that correctly and consistently uses OCL to define class contracts is probably test-ready. OCL expressions can be used in the ***Invariant Boundary***, ***Nonmodal Class Test***, ***Modal Class Test***, and ***Mode Machine Test*** patterns. This language presents many intriguing possibilities for test automation.

## 8.3 Use Case Diagram

### 8.3.1 Notation and Semantics

A **use case** is an abstraction of a system response to external inputs. It accomplishes a task that is important *from a user's point of view.* "When a user uses the system, she or he will perform a behaviorally related sequence of transactions in a dialogue with the system. We call such a special sequence a use case" [Jacobson 95]. Use cases are the dominant form of system-level requirements specification in object-oriented development: "the collection of use cases is the complete functionality of the system" [Jacobson+92]. Use cases are said to "drive" the unified development process[3]—in that they are the primary representation of system requirements [Jacobson+99]. Use cases can represent many kinds of system requirements:

- Functional requirements
- Allocation of functionality to classes (objects)
- Object interaction and object interfaces

2. OCL is a declarative language; that is, none of its operations may change the state of the modeled system. Operations are modeled by what must be true after the operation is completed. The computation or transformation that achieves this condition cannot be expressed.

3. The unified development process is a development methodology proposed by the primary authors of the UML—Booch, Rumbaugh, and Jacobson.



- User interfaces
- User documentation

In addition, use cases can be applied to analyze and define related design problems:

- Determination of development increments
- Establishment of traceability
- Conceptualization and prototyping
- Determination of system boundaries
- Development resource and effort sizing
- Architectural partitioning

A use case focuses on only those features visible at the external interfaces of a system. It represents a dialog between the system and external **actors**. An actor need not be human. Use cases, for example, may describe interactions with other computer systems and electromechanical sensors and actuators. A **use case instance** defines particular input values and expected results. A use case is composed of **operations**. An operation is a particular sequence of messages exchanged among objects, initiated by an external input. An operation causes a particular path to be traced through a sequence diagram. An **event trace** is a path allowed by the structure of a **sequence diagram**. The UML use case model has several key elements and is a simplified form of the OOSE use case [Jacobson+92] (some OOSE use case elements dropped from the UML use case reappear in the UML Scenario Diagram).

- ***actor*** Any person or system submitting or receiving information to or from the system under test.
- ***system boundary*** An abstraction of the implementation interface that accepts and transports external inputs and system outputs. It includes sources producing messages accepted by blocks in the SUT and sinks that accept messages generated by blocks in the SUT. It may be a virtual machine, such as a GUI subsystem that is used by an application system.
- ***use case instance*** A use case with specific values specified for the use case parameters and the state of the SUT.

Figure 8.3 depicts the overall system relationships represented by a use case. The collaboration among components that support a use case may be

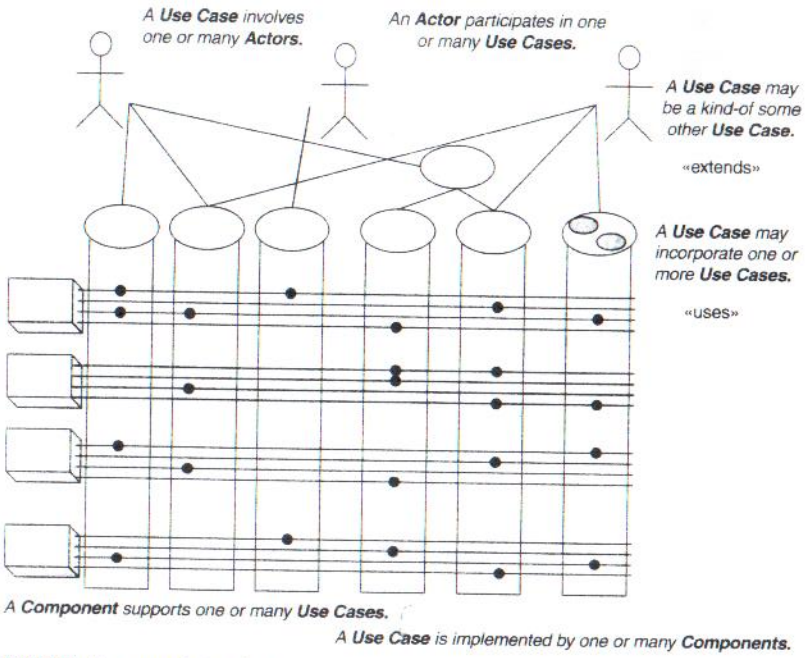


FIGURE 8.3 Use case relationships.

represented by a Sequence Diagram (see Section 8.5). Figure 8.4 shows a Use Case Diagram for the cash box example.

### 8.3.2 Generic Test Model

Table 8.1 lists the basic elements of the Use Case Diagram. Table 8.2 lists test criteria suggested by applying the relational test requirements (see Table 8.17 on page 312) to use case relationships. ***Extended Use Case Test*** is the primary test design pattern to develop application-specific tests. Figure 8.1 lists other test design patterns that use information presented in a Use Case Diagram and use case narratives.



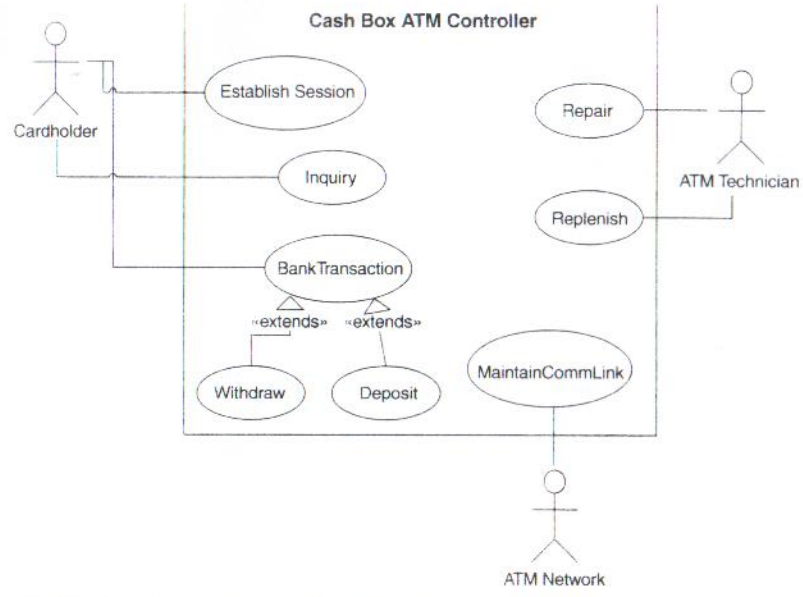


FIGURE 8.4 Use Case Diagram, cash box example.

### 8.3.3 Testability Extensions

A system specified with use cases provides some of the information necessary for system testing. Jacobson suggests four general kinds of tests that can be derived from use cases: (1) tests of basic courses, or "the expected flow of events," (2) tests of odd courses, or "all other flows of events," (3) tests of any line-item requirements traceable to each use case, and (4) tests of features described in user documentation traceable to each use case [Jacobson+92]. Equivalence class partitioning [Myers 79] is suggested for test design. Essentially the same test strategy is recited in [Jacobson+99]. From a testing perspective, this is good as far as it goes. But some key questions are unanswered.

TABLE 8.1 Use Case Diagram Elements

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Nodes | Actors (A) | Stick figure | Any external input source or response sink: external systems, human users, electromechanical sensors and actuators, and so on |
| | Use case (U) | Oval | A collection of interactions that accomplishes a user-defined task |
| | System | Box | A system that supports a use case |
| Edges | A communicates with U | Arrow | Links actors with use cases |
| | U1 extends U2 | Generalization | U1's behavior is added to U2 |
| | U1 uses U2 | Generalization | U2's behavior is used by U1 |

TABLE 8.2 Generic Use Case Test Requirements

| Relationship | R | S | T | Test | Test Requirement† |
|---|---|---|---|---|---|
| Actor communicates with Use Case | DC | Yes | DC | 24 | Every use case<br>Every actor's use case |
| Use case 1 extends Use Case 2 | No | No | Yes | 2 | Every fully expanded extension combination |
| Use case 1 uses Use Case 2 | No | DC | Yes | 8 | Every fully expanded uses combination |

†Entries to be read as "At least one test case that exercises <test requirement>."

### How do I choose test cases?

Equivalence class partitioning is a deprecated test model (see Chapter 10). Even if it were not, UML use cases lack two necessary elements of test design. First, domain definitions for input and output variables are not a part of the UML. (This problem is not limited to the UML—with the exception of Fusion [Coleman+94], no other OOA/D approach requires explicit domain definitions for use case variables.) Second, testable specification of input/output relationships and the conditions that determine the basic and alternate flows are likewise absent in the UML.

### In what order should I apply my tests?

Jacobson et al. are silent on the order in which to apply use case tests. Typical systems have hundreds of use cases from which tens of thousands of test cases can be developed. Sequential constraints and dependencies among the test cases are typically present and must be observed to conduct testing.



### How do I know when I'm done?

Jacobson et al. argue that all use cases and each of their equivalence classes should be tested at least once. This goal is ambiguous because a wide range of test suites could be produced to satisfy this test strategy, because a wide range of artifacts can satisfy the requirements of use case and equivalence class specifications.

A use case describes interactions but does not tell us how often the interaction occurs or specify what the content of the interaction will be. *Use cases, as defined in UML, OOSE, and other object-oriented methodologies, are necessary but not sufficient for system test design.* We must determine the following additional items:

- The domain of each variable that participates in a use case
- The required input/output relationships among use case variables
- The relative frequency of each use case
- Sequential dependencies among use cases

**Extended use cases** provide this additional information and may be prepared with any OOA/D technique that calls for use cases. Operational variables are analyzed to construct an operational relation. Relative frequencies are developed by constructing an operational profile. This approach allows systematic generation of test cases, ensuring that field reliability is maximized for a given testing budget.

### Operational Variables

The modeling of operational variables is illustrated with the Establish Session use case for an automatic teller machine (ATM) given in *Extended Use Case* (Chapter 14). Its instances differ by the amount of the transaction, the state of the customer's account, the state of the card, the number of times that the customer enters a wrong PIN, how much cash is contained in the ATM, the state of the ATM, and other factors. These factors are **operational variables**.

Operational variables may vary from one use case instance to the next use case instance and are used to construct specific test cases. They can be identified as follows.

1. *System boundary inventory.* Operational variables are typically visible at the system boundary—for example, the objects visible in the View component of an MVC application or the widgets constituting the GUI of the system under test. This inventory is not sufficient, however, because constraints or relationships among variables, key characteristics of the user or environment, and state information can be missed by a simple inventory.

2. *Use case instance inventory.* We could simply dream up (or observe) some arbitrary number of use case instances and then use them as test cases. This approach is weak from a testing point of view, because we cannot be sure that the resulting test suite will exercise all relations represented by the system's use cases.

3. *Operational variable definition.* Operational variables are inputs, outputs, and environmental conditions that (1) result in significantly different actor behavior; (2) abstract the state of the system under test (e.g., an ATM state of out of service, out of cash, ready, and so on); and (3) result in a significantly different system response.

### The Operational Relation

Once operational variables have been defined, logical relationships among operational variables are modeled with a decision table yielding the **operational relation** (see Chapter 6). This supports systematic generation of test cases. When all of the conditions in a row are true, the corresponding action is to be produced. The operational relation supports systematic selection of test cases. At minimum, every row should be made true once and false once. If any operational variable specifies a range, tests that probe the boundaries of the range should be added. Chapter 6 discusses strategies for developing test suites from combinational models.

"Robustness analysis" is suggested as an initial validation of use cases [Jacobson+92, Rosenberg+99] but does not require testable content. It has been my experience that the analysis required to produce extended use cases is very effective at finding omissions, inconsistencies, and other requirements bugs. Validation techniques applicable to the operational relation are discussed in Chapter 6.



## 8.4 Class Diagram

### 8.4.1 Notation and Semantics

A Class Diagram (also Object Diagram, Static Structure Diagram) represents relationships among *classifier elements*. Class Diagrams can serve many purposes. They typically represent the entities and relationships of interest for a particular application and document the structure of the classes in a library, framework, or application. Figure 8.5 shows a partial Class Diagram for the cash box example.

Most classifier elements are represented by a box with *compartments*. The kind-of classifier may be designated with a keyword (e.g., «interface»).

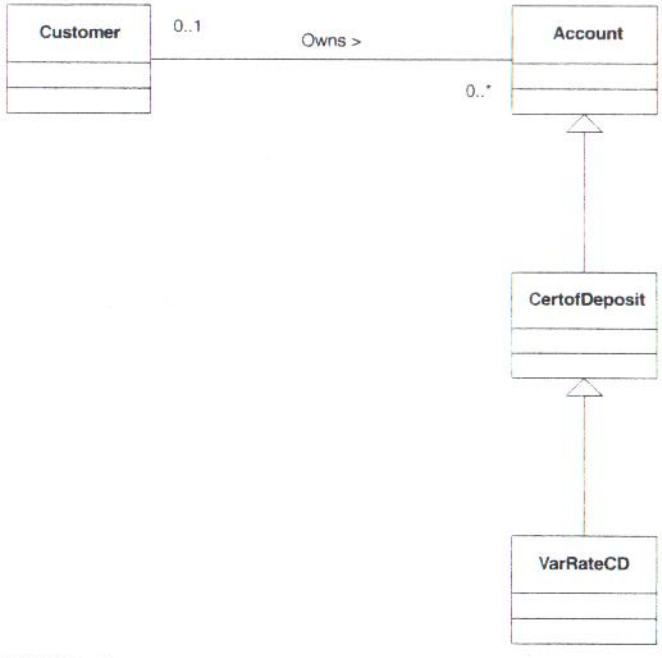


FIGURE 8.5 Class Diagram, cash box example.

TABLE 8.3 Class Diagram Elements: Nodes

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Nodes | Type | Class box «type» | A role or entity; may have attributes but no operations |
| | Class | Class box (name, operation, and attribute compartments) | An implementation class |
| | Interface | Class box or small circle | Abstract class, Java interface, Objective-C Protocol |
| | Parameterized class | Class box with parameter box | C++ template class, Eiffel generic, and so on |
| | Meta-class | Class box «metaclass» | An implementation meta-class |
| | Association class | Class box | An association as a class |
| | Object | Object box (name and attribute compartments); the name is underlined and of the form ClassName:ObjectName | An instance of a class |
| | Composite object | Object box with object boxes in the attribute compartment | The attributes of a composite object as a graph, allowing relationships to be shown |
| | Utility | Class box «utility» | A group of global variables and procedures |

Table 8.3 lists the various classifiers. Several basic relationships and their variants may be represented in a Class Diagram.

### Association

An **association** between two classes means that both must implement a mutual constraint or dependency. For example, in the cash box ATM controller, we need to represent accounts and customers. Figure 8.5 shows the Customer-Owns-Account relationship, including a classifier box for the Customer and Account classes and an association line to show the Owns relationship.

An association models a constraint that must be realized in the implementation. It does not necessarily represent the class or object structure of the implementation that realizes this constraint. The association name suggests the nature of the modeled relationship. Usually, this relationship can be read as a declarative sentence in either the active or passive voice. For example, the association name *owns* yields "customer owns account" (active voice) or "account is owned by customer" (passive voice).



*Multiplicity* parameters appear at the ends of the association line. These parameters define how many instances of one class may be associated with an instance of the other class. The multiplicity parameters in Figure 8.5 imply that (1) a customer is not required to own an account, (2) an account is owned by only one customer or has no owner, and (3) a customer may own many accounts.

Pair-wise multiplicity associations are common. In this situation, paired multiplicity parameters appear at both ends of the association line. Each pair defines the minimum and maximum number of instances for the class nearest this pair, defined with respect to any single instance of the class opposite this pair. A multiplicity parameter pair is interpreted by assuming that we have a single instance of one class and then reading the minimum and maximum number of instances for the other class. For example, reading from left to right, Figure 8.5 asserts that any given instance of a customer may own no account (0) or any number of accounts (*). Reading in the opposite direction (right to left), we see that an account may have no owner (0) or at most one owner.

### Aggregation

Aggregation relationships identify classifiers that are constituent parts of another classifier (a "container"). For example, the instance variables of classes are typically objects of other classes. The class of the instance variables may be shown by an aggregation relationship. Two kinds of aggregation may be represented.

- Simple *aggregation* is shown by a line with an unfilled diamond at the container end. This indicates that the constituent classifier is created and destroyed independently of the container classifier.
- *Composition* is shown by a line with a filled diamond at the using end. This indicates that the constituent classifier is created and destroyed with the container classifier.

Roles and multiplicity may be specified for aggregation relationships. Aggregation may be drawn as classifiers connected with an aggregation line or as a composite object (an object box enclosing other object boxes in the attribute compartment).

### Generalization

Generalization relationships identify classifiers that share common elements. For example, a superclass generalizes a subclass; that is, the subclass is a specialization of the superclass that shares the elements of the superclass. Generalization is shown by a line with an unfilled triangle whose apex points to the more general classifier.

Generalization may represent abstract generalization/specialization of types and implementation inheritance of a class hierarchy. When it is used to represent an implementation hierarchy, we assume the scoping and semantics of the implementation language.

## 8.4.2 Generic Test Model

Table 8.3 lists the basic elements included in a Class Diagram. Relationships are listed in Table 8.4. Table 8.5 on page 288 lists test criteria suggested by applying the relational test requirements (see Table 8.17 later in the chapter) to relationships. ***Class Association Test*** is the primary test design pattern employed to develop application-specific tests. Figure 8.1 lists other test design patterns that use information presented in a Class Diagram.

# 8.5 Sequence Diagram

## 8.5.1 Notation and Semantics

A Sequence Diagram represents how a sequence of messages exchanged among objects can accomplish some result of interest. Sequence Diagrams are typically used to design the collaboration necessary to implement a use case. Generally, vertical lines represent objects and horizontal lines represent messages sent from one object to another. Time progresses from top to bottom—if message `foo` appears above message `bar`, then `foo` is sent before `bar`. An end-to-end path is called an *interaction*.

The nodes in a Sequence Diagram represent runtime properties of executable components and provide some drafting convenience features:

- *Objects*. An object box designates an object that participates in an interaction. The object may be a composite; for example, it can represent a component, an entire subsystem, or an interface to an external system.



TABLE 8.4 Class Diagram Elements: Edges

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Edges | Association (binary) | Undirected line with ornaments | A required pair-wise property between instances of the classifier |
| | Association (*n*-ary) | A large diamond with association lines | An association among instances of three or more classifiers |
| | Association Path | Undirected dashed line | Connects an association class to an association line |
| | *Or* association | Undirected dashed line {or} | Indicates that exactly one of the spanned associations holds for any given set of classifier instances |
| | Association Qualifier | Box with attribute(s) names, between an object and an association line | The attributes that identify instance-pair subsets of an association; similar to the key of a database or file. |
| | Aggregation | Line with a small, unfilled diamond at the container end | The contained classifier is a part of the container classifier and may be created and destroyed independently of the container |
| | Composition | Line with a small, filled diamond at the container end | The contained classifier is a part of the container classifier and is created and destroyed along with the container |
| | Generalization | Line with an unfilled triangle at the general end | The specialized classifier is a kind of the general classifier (e.g., a subclass of a superclass) |
| | Dependency | Dashed arrow | A physical or logical reference such that changing the target classifier may require changing the source classifier |
| | Link | Line between two objects | A list of object references; an instance of an association |
| | Realizes | Dashed line with a solid triangular arrowhead | Indicates an implementation class for a type |

TABLE 8.5 Generic Test Requirements for Class Diagrams

| Relationship | R | S | T | Test | Test Requirement† |
|---|---|---|---|---|---|
| Association (binary) | | | | | Application-specific. See ***Class Association Test***. |
| Association (*n*-ary) | | | | | Application-specific. See ***Class Association Test***. |
| Association path | | | | | Tested as part of its association. |
| *Or* association | N | Y | N | 4 | Each association singularly (accepts).<br>Two or more associations (rejects). |
| Association qualifier | | | | | Tested as part of its association. |
| Aggregation | DC | N | N | 19 | Independent creation and destruction of the container and the component. |
| Composition | DC | N | Y | 20 | Sequential creation and destruction of the container and the component.<br>An attempt at independent creation and destruction of the container and the component fails. |
| Generalization | N | N | Y | 2 | Each superclass classifier is exercised in the context of every subclass classifier. |
| Dependency | DC | N | N | 10 | Application-specific. |
| Link | | | | | Tested as part of its association. |
| Realizes | N | N | N | 1 | Each behavior of the type in the implementation class. |

†Entries to be read as, "At least one test case that exercises <test requirement>."

- *Object lifeline.* The lifeline is a dashed vertical line that connects the object box to an activation interval for the object. An **X** at the bottom of a lifeline marks the explicit or implicit destruction of the object.
- *Object activation.* An activation is shown by a narrow rectangle overlaying the lifeline. At the object scope, the activation represents placing an executable instance of the object on the runtime stack. When an object is running (i.e., on the top of the runtime stack), the corresponding area of the activation box may be shaded. When the object has been context-swapped (i.e., not on the top of the runtime stack), the activation box is unfilled.
- *Conditional branch.* As a drawing convenience, a conditional segment within the scope of a method activation can be shown by a parallel activation box attached to the original activation by dashed lines. This does *not* represent a new activation record on the runtime stack.



- *Recursive activation.* An activation that results from a recursive call is shown by a single, short activation box that partially overlays the initial activation box. A looping arrow distinguishes this box from a loop box. Note that the sending method and the receiving method must be the same. The implementation semantics of recursion are language- and compiler-specific.

The edges in a Sequence Diagram represent generic properties of runtime *procedure calls:*

- *Suspend/resume procedure call.* A solid line with a filled arrowhead represents a message, function call, remote procedure call, interprocess communication call, and so on. Control is transferred from the currently active method to the receiver of the call. Recipient visibility to the call parameters is established. No return arrow from the receiver to the sender is needed, but a dashed arrow may be used.
- *Nonblocking procedure call.* A solid line with a half-open arrowhead represents a message, function call, remote procedure call, or interprocess communication call that does not suspend activation of the sender. The receiver of the call is activated when the message is received. Visibility is established for the call parameters. A dashed arrow must be used to show the return (if any) to the sender.
- *Conditional procedure calls.* Solid lines leaving the same point on the activation box indicate that only one of two or more possible procedure calls will be sent. The condition that selects each call may be written near the line, in brackets.
- *Procedure call loop.* Iteration is shown by a box that overlays the lifeline box. The predicate for a conditional or a loop appears in brackets, near the lower part of the loop box. A loop box does not have entry/exit arrows; a recursive activation box does.
- *Procedure call delay.* Flat procedure call arrows represent calls whose sending time is inconsequential to the model. If the transmit time of a call is significant, then the arrow is drawn with a downward slope.

Figure 8.6 shows a Sequence Diagram for the Start New Session use case in the cash box example.

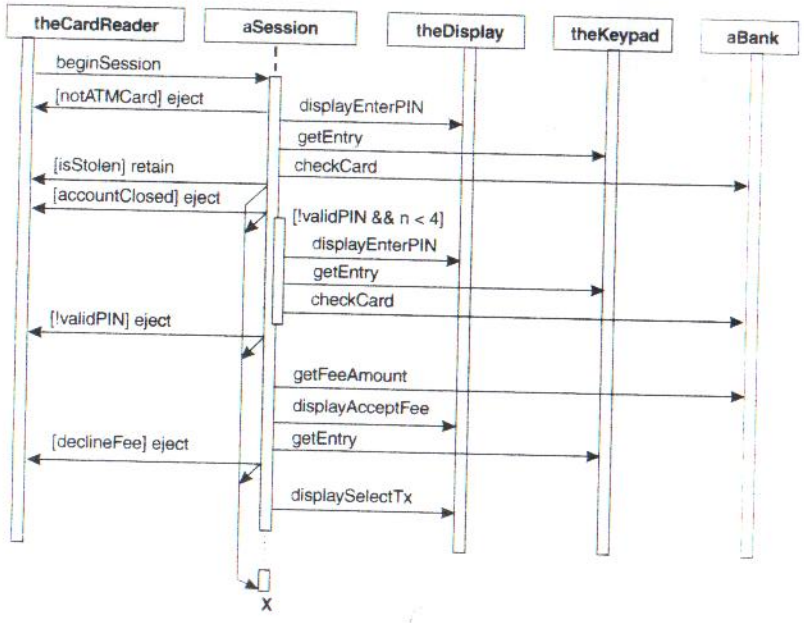


FIGURE 8.6 Sequence Diagram, cash box example.

### 8.5.2 Generic Test Requirements

The relationship between use case instances and Sequence Diagrams is shown in Figure 8.3. Table 8.6 lists the graph elements of Sequence Diagrams. Table 8.7 lists the generic test requirements suggested by applying the relational test strategy (see Table 8.17 later in the chapter). These test requirements confirm that basic testing practice should apply to Sequence Diagrams: all end-to-end paths should be identified and exercised. This is equivalent to the requirement to identify and exercise all transitive relations formed by the variations on client-sends-message-to-server. ***Round-trip Scenario Test*** is the primary test



TABLE 8.6 Sequence Diagram Elements

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Nodes | Object and Object Lifeline | Object Box over vertical dashed line | An object, component, or subsystem |
| | Object Activation | Narrow rectangle | The presence or absence of an object instance during a scenario |
| | Conditional Branch | Short Narrow Rectangle connected by dashed lines to an Activation box | Conditional code segment within a method; diagraming convenience |
| | Recursive Activation | Short Narrow Rectangle, overlaying an Object Activation box, pointed to by a looping arrow | Recursive activation |
| Edges | Suspend/Resume Procedure Call | Solid arrow | A message that suspends activation of the sender |
| | Nonblocking Procedure Call | A solid line with a half-open arrowhead | A message that does not suspend activation of the sender |
| | Conditional Procedure Calls | Solid lines leaving the same point on the activation box | Only one of the indicated procedure calls will be sent |
| | Procedure Call Loop | A narrow rectangle that partially overlays an Object Activation box | The scope of a loop. The loop termination condition may be noted next to the rectangle. |
| | Recursive Call | Loop arrow to a Recursive Activation box | Recursive call |
| | Procedure Call Delay | A downward-sloping arrow | Significant message transmit time |

TABLE 8.7 Sequence Diagram Test Requirements

| Relationship | R | S | T | Test | Test Requirements† |
|---|---|---|---|---|---|
| Client calls Server (suspend/resume) | Y | N | Y | 11 | Client calls Server and returns. |
| Client calls Server (nonblocking) | N | N | Y | 2 | Client calls Server and returns. Client continues execution. |
| Client may call Server 1, 2, . . . | N | N | Y | 2 | Client calls Server 1, 2, . . . |
| Client repeats call to Server | Y | N | Y | 11 | Client repeats call to Server. |
| Client recursively calls Client | Y | N | N | 10 | Client recursively calls Client. |
| Client call delayed to Server | N | N | N | 1 | Client calls Server. |

†Entries to be read as, "At least one test case that exercises <test requirement>."

design pattern to develop application-specific tests. Figure 8.1 lists other test design patterns that use information presented in a Sequence Diagram.

### 8.5.3 Testability Extensions

A Sequence Diagram unfolds message paths in the time dimension, which provides a useful representation for conceptualizing how a collaboration will be accomplished. However, it is a poor model for developing tests of their control flow. Where a properly constructed flow graph will show all possible activation sequences, a Sequence Diagram is *required* to show only a *single* collaboration. For example, the implementation of a use case may require several scenario diagrams. A complete model of the use case's implementation would require an overlay composed of all of these diagrams. This fragmentation creates opportunities for errors and makes it difficult to decide when a complete test suite has been developed.

Sequence Diagrams pose some additional problems:

- Representing complex control is difficult. The notation for selection and iteration is clumsy and supports possibly conflicting interpretations. This poses more problems for test design, as it will be difficult to decide when all paths in a Scenario Diagram have been covered.
- The distinction between a conditional message and a delayed message is weak (a line angles down for both; a conditional message has a predicate string). A conditional delayed message could be noted by showing the same time for both messages.
- Dynamic binding and unique superclass/subclass behavior cannot be represented directly. A Sequence Diagram depicts a single collaboration. The binding of a message to a different class (even within the same hierarchy) must be shown on a separate sheet or with an object lifeline for each level in the polymorphic server hierarchy.

A likely result is that a Sequence Diagram will not accurately represent a complex control implementation. Test design should not be done without reverifying the content of the model and analyzing the actual structure of the implementation to assess interfaces, paths, and other features. The ***Round-trip Scenario Test*** pattern shows how to derive a composite control flow graph from a Sequence Diagram. It allows established control flow test models to be used as test models.



## 8.6 Activity Diagram

### 8.6.1 Notation and Semantics

An Activity Diagram represents sequences in which an *activity* may occur. The model borrows ideas from flow charts, state transition diagrams, industrial engineering work flow graphs, and Petri nets.[4] It is described as a flow chart that can represent two or more threads of execution. Strangely, an Activity Diagram includes no activities—the entire diagram is considered to represent a single activity. Instead, it is composed of *action states*, each of which is a collection of happenings that achieves a result of interest. When an action state finishes, one or more successor action states begin. Several action states may run concurrently. In effect, an action state is a process, not a state.

The Activity Diagram is typically used for modeling human work flow and the interaction of this work flow with a software system. It may be associated with a use case or a class. Because it supports all of the elements of a basic flow graph, this type of diagram can be used to develop test models for control flow-based techniques. Developers who are limited to using a CASE tool that supports only UML can still produce these diagrams by using a subset of the Activity Diagram.

- The Activity Diagram may be used to represent decision tables (see Chapter 6). It can therefore support test development using the ***Combinational Function Test*** and ***Extended Use Case Test*** patterns.
- The Activity Diagram can be used to create a composite control flow graph for a collection of Sequence Diagrams. This use supports ***Round-trip Scenario Test*** (see Chapter 12).
- The Activity Diagram can be used to create a control flow graph at method scope. This application may be useful to analyze paths for coverage.

The nodes in an Activity Diagram represent processes and process control:

- *Action state.* A collection of happenings that are of interest. This is labeled with a descriptive phrase and represented by a horizontal capsule.

4. Activity Diagrams are remarkably similar to the SREM system diagrams [Alford 85], which also represent multiple threads of control.

- *Decision.* A diamond shows there are at least two successor nodes, but only one is to be reached after the antecedent action state terminates.
- *Synchronization point.* A thick bar shows that the successor action state(s) is activated only when all predecessor states (or state) terminate. It can be thought of as an *and* gate: its successor nodes are activated only when all of its predecessor nodes have terminated. A synchronization point with two or more predecessors is called a *merge;* one with two or more successors is called a *fork.*
- *Object box.* See the discussion of the object box in the Sequence Diagram discussion, Section 8.5.1.
- *Signal receiver.* It shows that the successor action state(s) is activated only when the designated signal is received. It is shown as a rectangle with a triangular notch in either side.
- *Signal sender.* It shows that the designated signal is emitted when the predecessor action state terminates. It is shown as a rectangle with a triangular point in either side.
- *Swim lanes.* These lanes indicate a group to which action states may belong. They are shown by vertical lines that form areas into which action states may be placed. Swim lanes do not represent behavior.

The edges in an Activity Diagram show which activities may follow one another and, optionally, which objects may participate in an activity.

- *Control flow.* A solid arrow shows a predecessor/successor relationship. Control flow may be omitted when messages (dashed arrows) link the same action states.
- *Message flow.* A dashed arrow shows that messages are sent between an action state and an object.
- *Signal flow.* A dashed arrow links signal receipt and signal sending nodes.

Figure 8.7 shows an Activity Diagram for the cash box example. This diagram shows a fragment of the user behavior that occurs when a customer makes a deposit. The user must prepare the deposit envelope and insert it into the open depository slot. The slot automatically closes if no deposit is inserted after a certain interval.



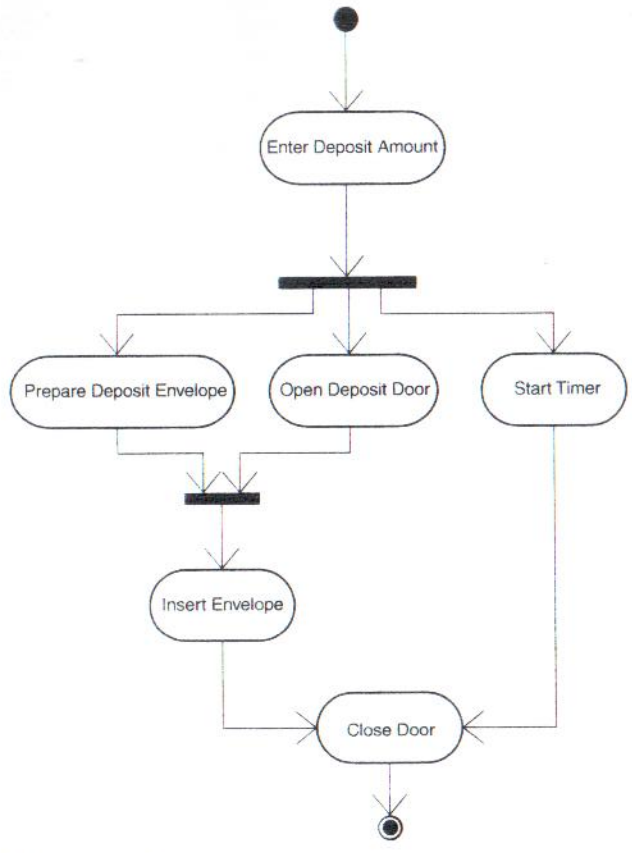


FIGURE 8.7 Activity Diagram, cash box example.

### 8.6.2 Generic Test Model

Table 8.8 lists the graph elements of Activity Diagrams. Table 8.9 lists test criteria suggested by applying the relational test requirements (see Table 8.17, page 312) to Activity Diagram relationships. The test requirements are applied to all pairs of nodes that can participate in a given relationship. None of the test design patterns in this book use the Activity Diagram as a basic test model. Figure 8.1 lists test design patterns that can use information presented in an Activity Diagram.

If an Activity Diagram supports concurrent action states (at least one "fork" synchronization bar exists), then the testing problem becomes more complex. Complete testing strategies for concurrent models are an advanced subject beyond the scope of this book. If the IUT is modeled by an Activity Diagram, the test strategies for product machines described in Chapter 7 may be useful.

TABLE 8.8 Activity Diagram Elements

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Nodes | Action State | Horizontal Capsule | A process |
| | Decision | Diamond | Only one of several successors may be next |
| | Swim Lane | Parallel vertical lines | A group of related processes |
| | Synchronization Point | Thick Bar | All predecessors must terminate before the successor can start |
| | Object | Object Box | An object, component, or subsystem |
| | Signal Receiver | Notched Rectangle | The successor cannot be started until the signal is received |
| | Signal Sender | Pointy Rectangle | A signal is sent before the successor starts |
| | Initial Action State | Filled Circle | Predecessor to first action state |
| | Final Action State | Bull's Eye | Final action state |
| Edges | Control Flow | Solid Arrow | Predecessor/successor relationship |
| | Message Flow | Dashed Arrow | Message sent to/from an object |
| | Signal Flow | Dashed Arrow | A pair of sender/receiver nodes |



TABLE 8.9 Activity Diagram Test Requirements

| Relationship | R | S | T | Test | Test Requirements* |
|---|---|---|---|---|---|
| Action 1 precedes Action 2 | DC | DC | Y | 26 | Action 1 is followed by Action 2. |
| Action depends on Synch Point | N | N | N | 1 | Synch Point is followed by Action. |
| Action 2, 3, . . . may follow Action 1 | DC | DC | N | 25 | Action 1 is followed by Action 2.<br>Action 1 is followed by Action 3.<br>. . . |
| Action depends on Signal | N | N | N | 1 | Action is reached after Signal. |
| Synch Point follows Action | N | N | N | 1 | Synch Point is reached after Action. |

*Key:* Action = Action State
*Entries to be read as, "At least one test case that exercises <test requirement>."

## 8.7 Statechart Diagram

A statechart represents sequential control requirements for a class, subsystem, or system. Statecharts are discussed in detail in Chapter 7. The UML definition of a statechart provides most of the information necessary for state-based testing, but permits the development of nontestable constructs and ambiguities. The FREE state model is consistent with all requirements of UML statecharts and remedies these problems. It is discussed in detail in Chapter 7. In addition, Chapter 7 provides the following model validation checklists:

- State name
- Guarded transition
- Flattened machine
- Robustness

As Figure 8.1 shows, FREE statecharts are the primary test model for *Modal Class Test*, *Modal Hierarchy Test*, and *Mode Machine Test*.

## 8.8 Collaboration Diagram

### 8.8.1 Notation and Semantics

A Collaboration Diagram represents interactions among object roles. The diagram shows *one* instance of an interaction, in which objects take on the *roles*

necessary to perform the computation. Although the diagram can be used to represent the structure of an implementation, this is not its primary purpose. The diagram is abstract (it does not require binding to specific objects) and is a slice of the modeled scope.[5] A Collaboration Diagram may be used to specify the implementation of a method, all methods in a class, or a use case. This type of diagram may also depict the participants in design patterns. Figure 8.8 shows a Collaboration Diagram for the cash box example.

A node is a *collaboration role*. Such a node can represent any object of a class that may fulfill the role. These nodes do not represent specific objects, however. They may be simple or composite. A *ClassifierRole* is depicted via a classifier box. An *AssociationRole* is shown as a line between two classifiers and may be ornamented with the items described in a Class Diagram. A *multiobject* is a collection. Links to a multiobject are indicated with a filled diamond ornament on the link. Messages to multiobjects assume an iterator to attach the message to a member of the collection. A multiobject is shown as two stacked rectangles. An *active object* is visible to the runtime environment as a task, process, or thread. It is designated by a classifier box with a heavy border.

Edges are *links*. They are shown with solid, undirected lines and represent that visibility is required from one role to another. The sending of messages and related control mechanisms is represented by a small arrow near the link. Because the links allow more than one message to be sent, execution paths cannot be traced directly from links.

An *interaction* is a call path within the scope of a collaboration. A *message flow* represents a message sent from one object to another and is shown by a short arrow parallel to the link that supports the message. Several types may be used:

- *Procedure call.* For example, a C++ function call or a Java message with typical suspend/resume activation semantics. It is shown by a solid line with a filled arrowhead, where the arrow points to the server object.
- *Flat flow of control.* Shows execution sequence at statement scope. It is shown by a solid line with an open arrowhead. The arrow points to the successor object.
- *Asynchronous flow of control.* Indicates an "asynchronous" call or other nonblocking mechanism (the recipient and the sender execute concurrently). The arrow points to the server object.

5. At the method scope, a slice is a subset of the method statements. It may be defined by many criteria—for example, all statements that are needed to support an entry–exit path or all statements that can influence a given statement [Tip 95].

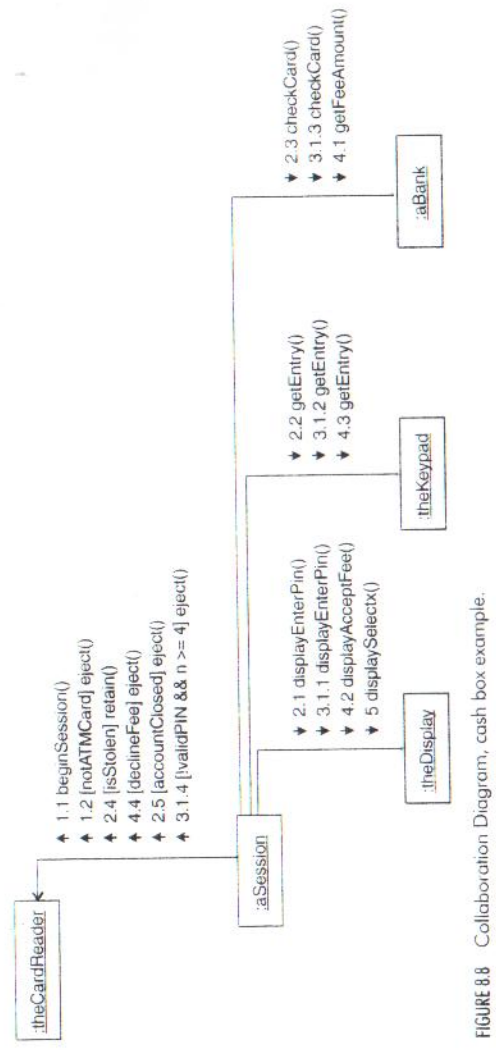


FIGURE 8.8 Collaboration Diagram, cash box example.

Each message may be labeled. Message labels can specify the predecessor call hierarchy, conditional execution, and iteration. All elements of the message are optional.

```
<message label> ::=
    <predecessor guard-condition>
    <sequence-expression>
    <<return-value> := <message-name> <argument-list>>
```

The <predecessor guard-condition> is a predicate evaluated within the scope of the predecessor object. If the predicate is true, then the message is sent. If not, then the message is not sent. The return expression specifies the sender's variable, which will become bound to the result of the designated message.

The <sequence-expression> denotes the predecessor messages using number-dot notation. The left most number denotes a step within the scope of the first method. A dot (period) separates the next level of control—that is, the scope of control for a second method called from the first method. Problems with this notation are discussed in Section 8.8.3.

Roles and links may be ornamented with *creation/destruction markers* that specify their scope of persistence.

### 8.8.2 Generic Test Model

Table 8.10 lists the graph elements of Collaboration Diagrams. Table 8.11 lists test criteria suggested by applying the relational test requirements (see Table 8.17) to Collaboration Diagram relationships. ***Collaboration Integration*** is the primary pattern for developing application-specific tests. Figure 8.1 lists other test design patterns that use information presented in a Collaboration Diagram.

### 8.8.3 Testability Extensions

The Collaboration Diagram is problematic as a test model for the following three reasons:

1. It represents only one slice of the IUT and can support test design of only this slice. In most circumstances, however, we are interested in testing and covering the entire implementation. A composite Collaboration Diagram would



TABLE 8.10 Collaboration Diagram Elements

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Nodes | Object Role | Classifier Box | An object, component, or subsystem |
| | Multiobject | Stacked Classifier Box | A collection of the same kind of objects |
| | Active Object | Classifier Box with heavy border | An object that runs as an independent process, task, or thread |
| Edges | Link | Solid line | Visibility required between two object roles |
| | Procedure Call | Line with filled arrowhead | Suspend/resume message sent to a role |
| | Flat Control Flow | Line with open arrowhead | Predecessor/successor relationship within a method |
| | Asynchronous Control Flow | Line with open half-arrowhead | Nonblocking message sent to a role |

TABLE 8.11 Collaboration Diagram Test Requirements

| Relationship | R | S | T | Test | Test Requirements† |
|---|---|---|---|---|---|
| Role is linked to Role | D | D | D | NA | Not testable per se. |
| Statement 1 precedes Statement 2 (Flat Control Flow ) | N | N | Y | 2 | Statement 1 and Statement 2. |
| Client sends message to Server (Procedure Call) | D | N | Y | 20 | Client sends message to Server and returns. |
| Client may send message to Server (Conditional Procedure Call) | D | N | Y | 20 | Client sends message to Server and returns. Client does not send message to Server. |
| Client repeats message to Server (Iterative Procedure Call) | D | N | Y | 20 | Client repeats message to Server and returns. |
| Client sends recursive message to self (Recursive Procedure Call) | Y | N | N | 10 | Client recursively calls self. |
| Client makes Nonblocking Procedure Call to Server (Asynchronous Control Flow) | N | N | Y | 2 | Client sends message to Server and returns. Server receives message. |

†Entries to be read as "At least one test case that exercises <test requirement>."

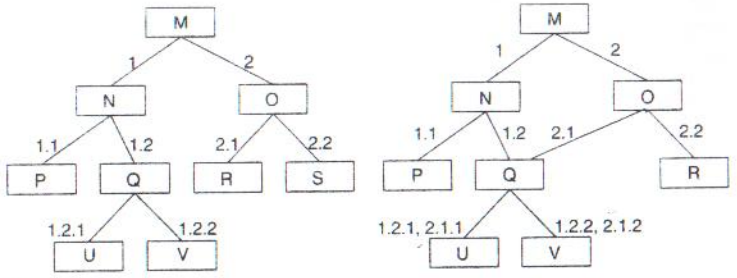


FIGURE 8.9 Overlapping sequence expressions in a Collaboration Diagram.

be necessary to develop a complete test suite for the implementation. Testing only the slices represented by one or several Collaboration Diagrams does not accomplish anything for the untested slices.

2. Although [OMG 98] states that a sequence expression has "an implicit predecessor [which] need not be explicitly listed," unless the objects visited by an interaction form a tree,[6] each message must explicitly designate its prefix. Consider the call paths in Figure 8.9 (boxes represent methods, lines are messages.) The hierarchy on the left is a tree, so only one root-to-leaf path exists for every leaf level node in the graph. If we label the tree according to the `<sequence-expression>` scheme, we can trace all possible call paths: 1, 1.1, 1.2, 1.2.1, 1.2.2, 2, 2.1. Now consider the hierarchy on the right. It differs only in that object `O` also sends a message to object `Q`. It is not a tree, however, because more than one root-to-leaf path exists through `O`. The links `Q-U` and `Q-V` reveal the problem. If `N` sends a message to `Q`, these links should be labeled 1.2.1 and 1.2.2. If `O` sends a message to `Q`, these links should be labeled 2.1.1 and 2.1.2. So, unless the call hierarchy of collaborations is restricted to a tree, all of the predecessor prefixes must appear on each link. If they are not shown, the predecessor is ambiguous.

3. One dot level must be added for each object that participates in an interaction. For example, if a dozen objects participate, a complete `<sequence-expression>` list will look something like this:

6. A tree is a graph that has no cycles and for which the number of nodes equals the number of edges plus 1.



4, 4.2, 4.2.18, 4.2.18.6, 4.2.18.6.8, 4.2.18.6.8.5, 4.2.18.6.8.5.13, 4.2.18.6.8.5.13.2, 4.2.18.6.8.5.13.2.5, 4.2.18.6.8.5.13.2.5.5, 4.2.18.6.8.5.13.2.5.5.3, 4.2.18.6.8.5.13.2.5.5.3.9.

The addition of iteration, exceptions, and recursion complicates the picture.

The Collaboration Diagram presents many design bug hazards and is, by definition, an incomplete representation of its implementation. Even if all tests in a test suite that has been developed from several Collaboration Diagrams pass, the tester should still ask, "Is that all there is?" Covering a collaboration typically will not achieve coverage of the IUT, unless the Collaboration Diagrams provide a complete representation of all slices.

The Collaboration Diagram represents highly abstract, implementation-independent requirements (roles) and implementation-specific details (call paths, statement level flow, return values, and so on). A likely result is a confusion about abstraction and implementation, leading to poor design decisions or detail design omissions. Tests for implementations developed from and documented by Collaboration Diagrams should be designed using some other model, such as a class or cluster state model.

## 8.9 Component Diagram

### 8.9.1 Notation and Semantics

A Component Diagram shows the dependency relationships among components, physical containment of components, and interfaces and calling components, using dashed arrows from components to interfaces on other components. A *component* is an implementation entity, including source code units, binary code (the output of a translator), and linked, loadable executable files.

- The symbol for a component is a rectangle with two small boxes on one side. The component name and type are underlined and placed either within the component symbol or near to it.
- The symbol for an interface is a small, unfilled circle attached to the component box.
- Compiler dependencies are shown by dashed arrows. A language-specific mechanism may be designated with a stereotype.

- The usage of interfaces and call dependencies is indicated with dashed arrows between components and the interfaces they use.

Figure 8.10 shows a Component Diagram for the cash box example.

### 8.9.2 Generic Test Model

Table 8.12 lists the graph elements of Component Diagrams. Table 8.13 lists test criteria suggested by applying the relational test requirements (see Table 8.17 on page 312). The generic test requirements confirm that basic testing practice should apply to Component Diagrams: all call paths should be identified and exercised.

Component Diagrams are useful for integration planning. The dependency analysis outlined in Chapter 13, for example, can be developed from a sufficiently detailed Component Diagram. Figure 8.1 lists other test design patterns that can use the information presented in Component Diagrams.

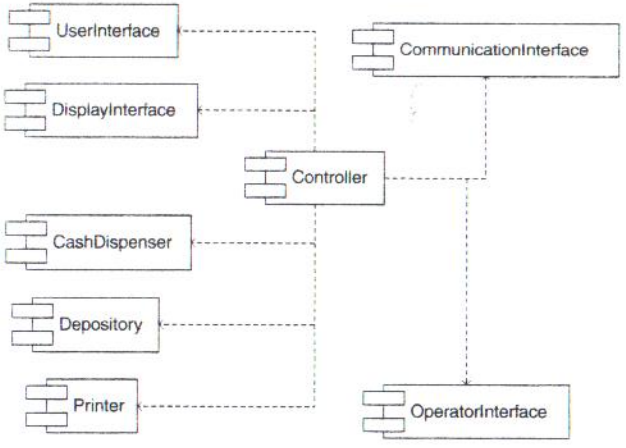


FIGURE 8.10 Component Diagram, cash box example.



TABLE 8.12 Component Diagram Elements

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Nodes | Component | Rectangle with two small box extensions on one side | A software entity visible to the target environment |
| | Interface | Small unfilled circle | A symbolic identifier visible to target environment that supports activation and parameter passing |
| Edges | Message Flow | Dashed arrow | Message sent to/from a component |
| | Source Code Dependency | Dashed arrow | Components cannot operate without one another |

TABLE 8.13 Component Diagram Test Requirements

| Relationship | R | S | T | Test | Test Requirements† |
|---|---|---|---|---|---|
| Component sends message to interface | D | N | Y | 20 | Client sends message to server and returns |
| Component depends on component | D | D | D | 27 | Application-specific |

†Entries to be read as "At least one test case that exercises <test requirement>."

## 8.10 Deployment Diagram

### 8.10.1 Notation and Semantics

A Deployment Diagram represents hardware, software, and network architecture.

- A Deployment Diagram *node* is a processor; it is not a node in the graph-theoretic sense. Nodes include all kinds of computer systems and may represent manual or mechanical processors. The symbol for a node is a box with dropping sides, suggesting a perspective drawing. The name and type of the node may appear as underlined text.
- Nodes are connected by *communication associations*, indicated by a solid line (no arrow). A stereotype may be used to designate particulars of the interface and channel.
- Component instances and objects that run on a node are shown by placing them in the node box. Software components in a Deployment Diagram must be executable units (entities that are visible to the

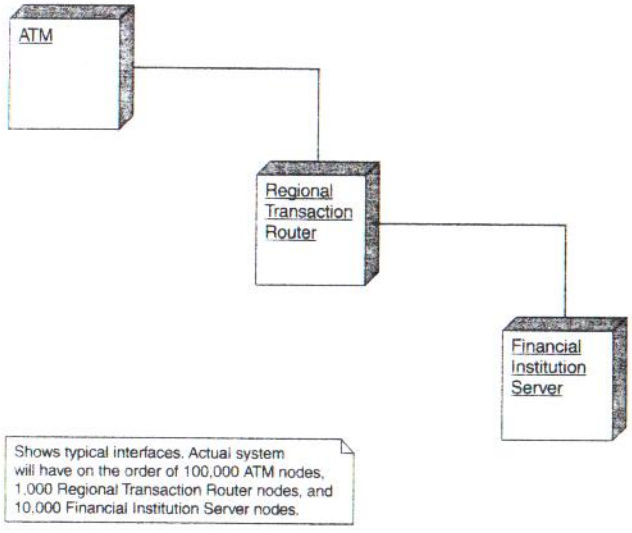


FIGURE 8.11 Deployment Diagram, cash box example.

execution and scheduling mechanisms of the target environment). Source code entities do not appear. The Component Diagram rules for components are also applied in Deployment Diagrams.

- A dashed-arrow dependency indicates that a node (or node type) can support a component (or component type). Stereotypes specify additional information; for example, the «becomes» stereotype shows that a component moves from one node to another.

Figure 8.11 shows a Deployment Diagram for the cash box example.

### 8.10.2 Generic Test Model

Table 8.14 lists the graph elements of Deployment Diagrams. Table 8.15 lists test criteria suggested by applying the relational test requirements (see Table 8.17). Deployment Diagrams are useful for integration planning. The dependency analysis outlined in Chapter 13, for example, can be developed from a



TABLE 8.14 Deployment Diagram Elements

| | Diagram Element | Symbol | Represents |
|---|---|---|---|
| Nodes | Node | Rectangle with dropped sides | A processor |
| | Component | Rectangle with two small box extensions on one side | A software entity visible to the target environment |
| | Interface | Small unfilled circle | A symbolic identifier visible to the target environment that supports activation and parameter passing |
| Edges | Communication Association | Solid line | A communication channel |
| | Component Dependency | Dashed arrow | The node(s) on which a component may run |

TABLE 8.15 Deployment Diagram Test Requirements

| Relationship | R | S | T | Test | Test Requirements† |
|---|---|---|---|---|---|
| Component runs on Node | D | D | D | 27 | Component can be loaded and run on each designated host Node |
| Node communicates with Node (Communication Association) | D | D | D | 27 | Open, transmit, and close communication for each remote component |

†Entries to be read as "At least one test case that exercises <test requirement>."

sufficiently detailed Deployment Diagram. Figure 8.1 lists other test design patterns that can use Deployment Diagrams.

## ◆ 8.11 Graphs, Relations, and Testing

A **graph** represents relationships among things by using two simple constructs: **nodes** and **edges**. Nodes are connected to other nodes by edges. This idea supports a wide range of mathematically rigorous models.[7] The nodes and edges of

7. The extensive body of mathematical knowledge about graphs is called *graph theory*. See the Bibliographic Notes section for some introductory sources. Formally, a graph is a set of vertices $V$ and a set of edges $E$. An edge is a pair of vertices. I use *node* instead of *vertex*, because the content of a node matters in software engineering models. A vertex is simply an undifferentiated dot. A graph is a set of pairs of nodes (two nodes connected by an edge). Binary (pair-wise) relations on sets can be represented and analyzed with graphs, and vice versa.

a graph represent a relation, which is another simple but powerful mathematical concept. UML diagrams can be viewed as graphs (in this mathematical sense) that represent many significant relationships and therefore provide a rich source of information for test design. The basic mathematical properties of graphs and relations can yield tests to check whether the IUT is consistent in its implementation of these relationships [Beizer 90, Beizer 95]. This strategy allows the same analysis to be applied to all UML diagrams and thereby simplifies the job of testing from these representations. Although it is applied only to UML diagrams here, the same analysis can be developed for any other graphic modeling technique and, with the necessary changes, for any quantitative modeling formalism.

A **relation**, in the mathematical sense, is a set such that some property holds among members of that set. It is a "set-theoretic formalization of a practical notion" [Mac Lane 86, 131], which is the basis of the relational data model and relational databases [Date 82]. In the UML, relations are represented with associations. Because the UML meta-model defines associations among UML elements, two levels of relations exist in any UML model: intrinsic relations among UML elements and application-specific relations among classifiers that represent the system under development.

For example, the classes of a framework that send messages to themselves can be enumerated by a relation defined by the requirement that at least one statement in the class has the form `this.foo( )`. Classes that are clients and servers can also be defined. For example, if class `A` declares an instance variable of type `B`, and class `C` declares one of type `D`, then the relation has two instances: `A, B` and `C, D`.

Any property can define a relation: whole-part, is-a-kind-of, adjacent-to, person-knows-system, and so on. A relation can be defined for any number of elements of a set. A *binary relation* is defined based on pairs. The mathematical shorthand for a binary relation is $xRy$, which is read "relation $R$ holds for all pairs of $x$ and $y$." For example, the mathematical expression $x = y$ asserts that the equality relationship holds for all pairs of $x$ and $y$. The programming predicate expression `(x == y)` can be used to identify $x,y$ pairs that do (and do not) satisfy this relation. If a relationship is not permitted, we write $\sim xRy$. A binary relation can be thought of as a table with two columns. Each row in this table is a member of the set defined by the relation.

Certain general kinds of relations are useful for test design:

- A **reflexive relation** is one where the property of interest holds when applied to each individual entity. For example, consider the relation *person-recognizes-person*. This relation is reflexive, assuming that a person can recognize himself. Jill recognizes Jill—that is she recognizes herself. In a graph, an edge that exits and enters the same node (a loop on a single node) often represents a reflexive relation. If the members of a reflexive relation are depicted as a table, every cell in the diagonal must be part of the relation (see Figure 8.12). A relation that does not allow self-reference is **nonreflexive**.



- A **symmetric relation** is one where for any pair of entities, the right and left sides can be exchanged and the property of interest still holds. For example, consider *person-recognizes-person*. If all the persons in question know one another, the relationship is symmetric. That is, if Jack recognizes Jill, then Jill recognizes Jack. In a graph, a pair of arrows going in opposite directions, a bidirectional arrow, or an undirected edge is often used to depict a symmetric relation. If the members of a symmetric relation are shown as a table, the upper and lower triangles will be mirror images (see Figure 8.12). A relation that does not allow transposition is **nonsymmetric**.

- A **transitive relation** is one where the property of interest holds when, for any two pairs of entities that share a common entity, the relationship also holds for both pairs. For example, consider the relation *person-isTallerThan-person*. If Jack is taller than Jill, and Jill is taller than Jane, then Jack must also be taller than Jane. In a graph, a path formed by several successive arrows may represent a transitive relation. A relation that does not allow transitivity is **nontransitive**.

Table 8.16 lists the corresponding set expressions. The modeler can define a relation so that any or all of these properties hold, do not hold, or are irrelevant. For example, the relation $x$ *precedes* $y$ is nonreflexive (an $x$ cannot precede itself), nonsymmetric (if $x$ precedes $y$ then $y$ cannot also precede $x$), and transitive (if $x$ precedes $y$ and $y$ precedes $z$, then $x$ must also precede $z$). If a property is irrelevant, then instances that would satisfy either case must be allowed. That is, suppose that we do not care whether an $x$ is reflexive: a class may or may not send messages to itself; a state may or may not have a transition back to itself, and so on. In other words, either situation is permitted.

For any given relation, we can ask which of the three properties is required, excluded, or irrelevant. The answers lead directly to a simple but effective test suite. That is, the general requirement implies a test requirement. A correct implementation will accept inputs (or states) that are consistent with the relation; it will reject inputs or states that are inconsistent with it. The test requirement

| a Reflexive Relation | | a | b | c |
|---|---|---|---|---|
| | a | ✓ | | |
| | b | | ✓ | |
| a Symmetric Relation | b | a | b | ✓ c |
| | a | | ✓ | ✓ |
| | b | ✓ | | ✓ |
| a Transitive Relation | c | ✓ a | ✓ b | c |
| | a | | ✓ | ✓ |
| | b | | | ✓ |
| | c | | | |

FIGURE 8.12 Tableau for reflexive, symmetric, and transitive relations.

TABLE 8.16 Formal Properties of Relations

| Property | Requirement |
|---|---|
| Reflexive | $(x,x) \in R$ for every $x \in S$ |
| Nonreflexive | $(x,x) \notin R$ for every $x \in S$ |
| Symmetric | If $(x,y) \in R$ then $(y,x) \in R$ for every $x,y \in S$ |
| Nonsymmetric | If $(x,y) \in R$ then $(y,x) \notin R$ for every $x,y \in S$ |
| Transitive | If $(x,y) \in R$ and $(y,z) \in R$, then $(x,z) \in R$ for every $x,y,z \in S$ |
| Nontransitive | If $(x,y) \in R$ and $(y,z) \in R$, then $(x,z) \notin R$ for every $x,y,z \in S$ |

A binary relation $R$ is a subset of the Cartesian product (cross-product) of some set $S$, $R \subseteq S \times S$. Variables $x$, $y$, and $z$ are elements of $S$, denoted $x \in S$, and so on. A pair of elements is denoted $(x,y)$.

is simple: Identify test inputs to show that the acceptable case is allowed and that the unacceptable case is rejected.

- When reflexivity is required, any $x$ that would make $xRx$ true should be accepted.
- When reflexivity is excluded, any $x$ that would make $xRx$ true should be rejected.
- When reflexivity is irrelevant, some $x$ should be accepted such that $xRx$ is true and some $x$ should be accepted such that $xRx$ is false.



- When symmetry is required, any $x$ and $y$ that would make ($xRy$ and $yRx$) true should be accepted.
- When symmetry is excluded, any $x$ and $y$ that would make ($xRy$ and $yRx$) true should be rejected.
- When symmetry is irrelevant, some $x$ and $y$ should be accepted such that ($xRy$ and $yRx$) is true. Also, some $x$ and $y$ for which ($xRy$ and $yRx$) does not hold should be accepted.
- When transitivity is required, any $x$, $y$, and $z$ that would make ($xRy$ and $yRz$ and $xRz$) true should be accepted.
- When transitivity is excluded, any $x$, $y$, and $z$ that would make ($xRy$ and $yRz$ and $xRz$) true should be rejected.
- When transitivity is irrelevant, some $x$, $y$, and $z$ that would make ($xRy$ and $yRz$ and $xRz$) true should be accepted. Also, some $x$, $y$, and $z$ for which ($xRy$ and $yRz$ and $xRz$) does not hold should be accepted.

Table 8.17 summarizes the test requirements for combinations of these relationships. Entries in the table are interpreted as follows:

- *Yes.* The relationship requires this property for all entity pairs.
- *No.* The relationship excludes this property for any entity pair.
- *DC (don't care).* When Reflexivity is a don't care, we should be able to find at least two different inputs (or states) such that one is reflexive and accepted and the second is reflexive and is rejected. That is, reflexivity does not determine the response of the IUT.
- *Accept.* When the IUT is presented with a test case that makes the test condition true, it is expected to accept the test inputs.
- *Reject.* When the IUT is presented with a test case that makes the test condition true, it is expected to reject the test inputs: an error message should be produced, an exception is thrown, the inputs are ignored, and so on.

The generic test model developed for each diagram type is also a basis for traceability (see Chapter 9). For example, one generic test requirement for use cases is that at least one test case is developed for each use case. This is also a traceability requirement. If a test strategy calls for this test, then traceability should be established and monitored between these items. ■

TABLE 8.17 General Test Conditions for Relations in a Graph

| | General Requirements | | | Test Conditions | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Reflexive? | Symmetric? | Transitive? | Reflexivity | | Symmetry | | Transitivity | |
| | *xRx* | *xRy* & *yRx* | *xRy* & *yRz* & *xRz* | *aRa* | *bRb* | *aRb* & *bRa* | *cRd* & *!dRc* | *ARB* & *bRb* & *aRc* | *cRd* & *dRf* & *!cRf* |
| 1 | No | No | No | reject | reject | reject | accept | reject | accept |
| 2 | No | No | Yes | reject | reject | reject | accept | accept | reject |
| 3 | No | No | DC | reject | reject | reject | accept | accept | accept |
| 4 | No | Yes | No | reject | reject | accept | reject | accept | accept |
| 5 | No | Yes | Yes | reject | reject | accept | reject | accept | reject |
| 6 | No | Yes | DC | reject | reject | accept | reject | accept | accept |
| 7 | No | DC | No | reject | reject | accept | accept | reject | accept |
| 8 | No | DC | Yes | reject | reject | accept | accept | accept | reject |
| 9 | No | DC | DC | reject | reject | accept | accept | accept | accept |
| 10 | Yes | No | No | accept | accept | reject | accept | reject | accept |
| 11 | Yes | No | Yes | accept | accept | reject | accept | accept | reject |
| 12 | Yes | No | DC | accept | accept | reject | accept | accept | accept |
| 13 | Yes | Yes | No | accept | accept | accept | reject | reject | accept |
| 14 | Yes | Yes | Yes | accept | accept | accept | reject | reject | reject |
| 15 | Yes | Yes | DC | accept | accept | accept | reject | reject | accept |
| 16 | Yes | DC | No | accept | accept | accept | reject | accept | accept |
| 17 | Yes | DC | Yes | accept | accept | accept | accept | accept | reject |
| 18 | Yes | DC | DC | accept | accept | accept | accept | accept | accept |
| 19 | DC | No | No | accept | accept | reject | accept | reject | accept |
| 20 | DC | No | Yes | accept | accept | reject | accept | accept | reject |
| 21 | DC | No | DC | accept | accept | reject | accept | accept | accept |
| 22 | DC | Yes | No | accept | accept | accept | reject | reject | accept |
| 23 | DC | Yes | Yes | accept | accept | accept | reject | reject | reject |
| 24 | DC | Yes | DC | accept | accept | accept | reject | reject | accept |
| 25 | DC | DC | No | accept | accept | accept | accept | reject | accept |
| 26 | DC | DC | Yes | accept | accept | accept | accept | accept | reject |
| 27 | DC | DC | DC | accept | accept | accept | accept | accept | accept |

*Note*: *x*, *y*, and *z* denote variables for the relation; *a*, *b*, *c*, *d*, *e*, and *f* denote specific test case values; *!* denotes logical *not*; & denotes logical *and*



## 8.12 Bibliographic Notes

Hundreds of articles and dozens of books interpret the UML and provide guidance for its use. The primary source for the UML is the *OMG Unified Modeling Language Specification* [OMG 98]. This document can be downloaded from the OMG Web site. The authors of the UML explain its concepts and organization in [Rumbaugh+98]. Fowler's *UML Distilled* [Fowler 97] provides a concise introduction. D'Souza and Wills show how to use the UML to support a comprehensive pattern–based modeling strategy that targets implementation with frameworks and components [D'Souza+99]. Douglass presents embedded system design techniques with UML [Douglass 98].

The use of relations and graphs for test design is discussed in [Beizer 90] with a useful reprise in [Beizer 95]. Most introductory texts on discrete mathematics outline the notion of a relation and the reflexive, symmetric, and transitive properties—for example, [Hopcroft+79] and [Lewis+81]. A rigorous analysis (double black) of the role of relations in testing is developed in [Gourlay 83]. Graph theory has been studied extensively. West's *Introduction to Graph Theory* [West 96] is a popular textbook that provides an extensive bibliography. It is limited to undirected graphs, however. Directed graphs and their applications are covered in [Deo 76].

# Part III Patterns

Part III presents test design patterns for results-oriented testing of classes, subsystems, and application systems. Taken as a whole, these patterns are a broad solution to the problem of test design in object-oriented development. Taken at any particular scope, they offer stand-alone solutions for developing and executing scope-specific test plans.

- Chapter 9, Results-oriented Test Strategy, provides the necessary background for understanding the use of test design patterns.
- Chapter 10, Classes, presents test design patterns for methods, classes, and flattened classes. Although these techniques are applicable to class clusters, each focuses on the behavior of a single class in the cluster. Code coverage and domain analysis for object-oriented systems is presented.
- Chapter 11, Reusable Components, presents test design patterns for object-oriented software artifacts designed for reuse. Object-oriented software is unique in its potential for reuse. Chapter 11 therefore focuses on testing of reusable components. The testing issues unique to several kinds of reusable artifacts are considered and resolved, including class, subsystem, and system scope components.
- Chapter 12, Subsystems, presents test design patterns for subsystems. Several distinct kinds of subsystems occur in object-oriented systems. Clusters, build groups, and process groups are considered.

- Chapter 13, Integration, presents test design patterns for integration. These patterns show how to choose the order and scope of component testing to achieve a stabilized system.
- Chapter 14, Application Systems, presents test design patterns for application systems. An application system is a collection of subsystems that cooperate to support a feature set intended for a user or customer.
- Chapter 15, Regression Testing, presents test design patterns for choosing how much of an existing test suite should be rerun after changing part of a system. Regression testing at class, subsystem, and system scope is considered.

# Chapter 9 Results-oriented Test Strategy

It doesn't matter whether a cat is black or white
so long as it catches mice.
*Deng Xiaoping*

**Overview**

This chapter provides a topical introduction to Part III. It introduces ideas used in all the test strategies of Part III: results-oriented testing and the test design pattern. Considerations for producing professional test plan documentation are discussed.

## 9.1 Results-oriented Testing

Results-oriented testing is a point of view:

Design test cases using scope-appropriate responsibility-based patterns. Develop *efficient* test suites: try to exercise many responsibilities and component interfaces with just a few test cases. If the components of the implementation under test are not trusted, choose an integration cycle based on code dependencies to control introduction of untrusted parts. Develop and execute the test suite according to this cycle. Determine responsibility coverage by analysis and implementation coverage by instrumentation. Stop testing when the modeled responsibilities and part interfaces have been covered.