CHAPTER 1

THE CHALLENGE

The lakes and rivers of New Zealand are a fisherman's paradise. But one threat to the quality of streams and fishing in New Zealand is the koi carp, which was introduced accidentally in the 1960s as part of a goldfish consignment and is now classified as a noxious species. Koi carp resemble huge goldfish, but they destroy native plants and fish habitats, eat indiscriminately like vacuum cleaners, and grow to about 75 cm long. They are found mostly in the Auckland/Waikato region, and the goal is to prevent them from spreading further.

Imagine that you are a regional manager of the Fish and Game Council and you been given the job of eliminating carp from a stream, so that rainbow trout and other fish can thrive there. How would you do it? Would you take any or all of the following actions?

- Employ hundreds of amateur fishermen to fish with rods and hooks and offer a bounty payment for each koi carp caught.
- Place nets at strategic places, with regular inspections to kill all koi carp and release all other fish.

Note: The photo here is of the Waikato River in Hamilton, New Zealand.

FIGURE 1.1 New Zealand's first electrofishing boat looks like something from a science fiction movie. But Waikato University's *Te Waka Hiko Hi Ika* is the first successful electrofishing boat. It has spectacular electronic prongs in front that dangle under the water and generate a 5 kilowatt pulsating electronic fishing field. Fish are temporarily stunned by the field and float to the surface. Pest fish, such as koi carp, can be scooped out with a net, while other fish are left unharmed. Some advantages of electrofishing over conventional fish-capturing techniques, such as netting, are that it captures fish of all sizes and from all locations, including bottom-dwelling ones. *Source*: Centre for Biodiversity and Ecology Research, University of Waikato, Hamilton, NZ. Used with permission.

- Use an advanced technology solution, such as an electrofishing boat (see Figure 1.1) that attracts all fish and allows pest fish like koi carp to be killed while other fish can be returned to the water unharmed [HOL05, H^+05].

Now imagine that you are the validation manager of a software development company that is finalizing the development of a new smart card payment system for car parking. To thoroughly test your system, which of the following actions would you take?

- Employ a dozen full-time testers to manually design tests, record the tests on paper, and then manually perform the tests each time the system changes.
- Manually design a set of tests and then use automated test execution tools to rerun them after every change and report tests that fail.
- Use state-of-the-art tools that can automatically generate tests from a model of your requirements, can regenerate updated test suites each time the requirements change, and can report exactly which requirements have been tested and which have not.

In both cases, the third solution takes advantage of new technology to get faster results with lower costs than traditional methods, and it ensures a more systematic, less ad hoc, coverage (of the fish in the stream, or the failures in the program).

This book will show you how to test your software systems using the third approach. That is, it will explain how a new breed of test generation tools, called *model-based testing* tools, can improve your testing practices while reducing the cost of that testing.

1.1 WHAT DO WE MEAN BY TESTING?

> *Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.*

This definition of testing, from the IEEE *Software Engineering Body of Knowledge* (SWEBOK 2004),[1] describes the top-level goals of testing. It goes on to give more detail:

> Software testing consists of the *dynamic* verification of the behavior of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behavior.

We've emphasized in italics the words that capture the key features of software testing; these are their definitions as they relate to this book.

Dynamic: This means that we execute the program with specific input values to find failures in its behavior. In contrast, *static* techniques (e.g., inspections, walkthroughs, and static analysis tools) do not require execution of the program. One of the big advantages of (dynamic) testing

[1] The SWEBOK can be downloaded from *http://www.swebok.org* or purchased from the IEEE.

is that we are executing the actual program either in its real environment or in an environment with simulated interfaces as close as possible to the real environment. So we are not only testing that the design and code are correct, but we are also testing the compiler, the libraries, the operating system and network support, and so on.

Finite: Exhaustive testing is not possible or practical for most real programs. They usually have a large number of allowable inputs to each operation, plus even more invalid or unexpected inputs, and the possible sequences of operations is usually infinite as well. So we must choose a smallish number of tests so that we can run the tests in the available time. For example, if we want to perform nightly regression testing, our tests should take less than 12 hours!

Selected: Since we have a huge or infinite set of possible tests but can afford to run only a small fraction of them, the *key challenge* of testing is how to select the tests that are most likely to expose failures in the system. This is where the expertise of a skilled tester is important—he or she must use knowledge about the system to guess which sets of inputs are likely to produce the same behavior (this is called the *uniformity assumption*) and which are likely to produce different behavior. There are many informal strategies, such as equivalence class and boundary value testing,[2] that can help in deciding which tests are likely to be more effective. Some of these strategies are the basis of the test selection algorithms in the model-based testing tools that we use in later chapters.

Expected: After each test execution, we must decide whether the observed behavior of the system was a failure or not. This is called the *oracle* problem. The oracle problem is often solved via manual inspection of the test output; but for efficient and repeatable testing, it must be automated. Model-based testing automates the generation of oracles, as well as the choice of test inputs.

Before describing the various kinds of testing, we briefly review some basic terms according to standard IEEE software engineering terminology.

A *failure* is an undesired behavior. Failures are typically observed during the execution of the system being tested.

A *fault* is the *cause* of the failure. It is an error in the software, usually caused by human error in the specification, design, or coding process. It is the execution of the faults in the software that causes failures. Once we have

[2] See Lee Copeland's book [Cop04] for a comprehensive overview of the most popular informal test design techniques.

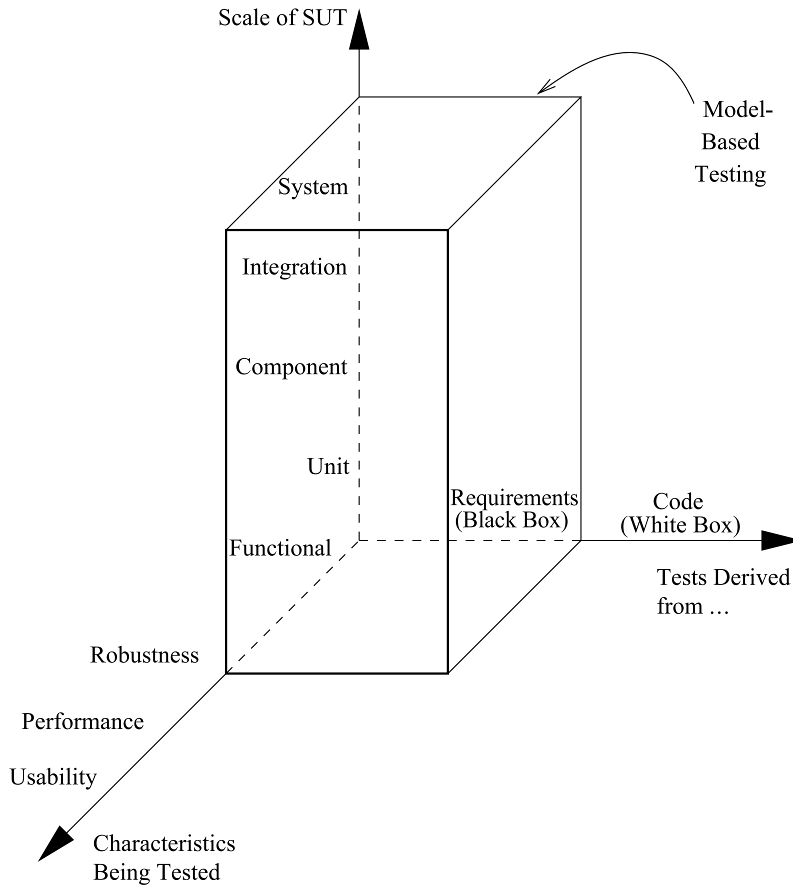

FIGURE 1.2 Different kinds of testing. *Source*: From Tretmans [Tre04]. Used with permission.

observed a failure, we can investigate to find the fault that caused it and correct that fault.

So *testing* is the activity of executing a system in order to detect failures. It is different from, and complementary to, other quality improvement techniques such as static verification, inspections, and reviews. It is also distinct from the debugging and error-correction process that happens after testing has detected a failure.

In fact, there are many kinds of testing. Figure 1.2 shows one way to classify various kinds of testing along three dimensions (adapted from [Tre04]). One axis shows the *scale* of the system under test (SUT), ranging from small units up to the whole system. *Unit testing* involves testing a single unit at

a time, such as a single procedure or a single class. *Component testing* tests each component/subsystem separately, and *integration testing* aims at testing to ensure that several components work together correctly. *System testing* involves testing the system as a whole. Model-based testing can be applied to any of these levels.

Another axis shows the different *characteristics* that we may want to test. The most common kind of testing is *functional testing* (also known as *behavioral testing*), where we aim to find errors in the functionality of the system—for example, testing that the correct outputs are produced for given inputs. *Robustness testing* aims at finding errors in the system under invalid conditions, such as unexpected inputs, unavailability of dependent applications, and hardware or network failures. *Performance testing* tests the throughput of the system under heavy load. *Usability testing* focuses on finding user interface problems, which may make the software difficult to use or may cause users to misinterpret the output.

The main use of model-based testing is to generate functional tests, but it can also be used for some kinds of robustness testing such as testing the system with invalid inputs. It is not yet widely used for performance testing, but this is an area under development.

The third axis shows the kind of information we use to design the tests. *Black-box testing* means that we treat the SUT as a "black box," so we do not use information about its internal structure. Instead, the tests are designed from the system requirements, which describe the expected behavior of that black box. On the other hand, *white-box testing* uses the implementation code as the basis for designing tests. For example, we might design a set of tests to ensure *statement coverage* of a procedure, meaning that each statement will be executed by at least one of the tests.

Much has been written about the pros and cons of black-box and white-box testing. However, the most common practice is to use black-box testing techniques to design functional and robustness tests. Some testers then use white-box coverage metrics to check which parts of the implementation have not been tested well so that extra tests can be designed for those cases. Model-based testing is a form of black-box testing because tests are generated from a model, which is derived from the requirements documentation. The next section describes model-based testing in more detail.

1.2 WHAT IS MODEL-BASED TESTING?

Model-based testing has become a bit of a buzzword in recent years, and we have noticed that people are using the term for a wide variety of test gen-

eration techniques. The following are the four main approaches known as model-based testing.

1. Generation of test input data from a domain model
2. Generation of test cases from an environment model
3. Generation of test cases with oracles from a behavior model
4. Generation of test scripts from abstract tests

We will briefly describe these approaches and then explain why this book focuses mostly on the third meaning of model-based testing and covers the other meanings more briefly.

When model-based testing is used to mean the generation of test input data, the *model* is the information about the domains of the input values and the test generation involves clever selection and combination of a subset of those values to produce test input data. For example, if we are testing a procedure that has three inputs, $A : \{red, green, yellow\}$, $B : 1..4$, and $C : \{car, truck, bike\}$, then we might use a *pairwise* algorithm[3] to generate a minimal set of tests that exercise all possible pairs of input values. For this example, a good pairwise algorithm would generate just 12 tests[4] rather than the $3 \times 4 \times 3 = 36$ tests that we would need if we tried all possible combinations. The automatic generation of test inputs is obviously of great practical importance, but it does not solve the complete test design problem because it does not help us to know whether a test has passed or failed.

The second meaning of model-based testing uses a different kind of model, which describes the expected *environment* of the SUT. For example, it might be a statistical model of the expected usage of the SUT [Pro03] (operation frequencies, data value distributions, etc.). From these environment models it is possible to generate sequences of calls to the SUT. However, like the previous approach, the generated sequences do not specify the expected *outputs* of the SUT. It is not possible to predict the output values because the environment model does not model the behavior of the SUT. So it is difficult to determine accurately whether a test has passed or failed—a crash/no-crash verdict may be all that is possible.

[3] See Chapter 4 for further discussion of pairwise testing, and the Pairwise website, *http://www.pairwise.org*, for tools, articles, and case studies on pairwise testing.

[4] For example, the 12 triples $(red, 1, car)$, $(red, 2, truck)$, $(red, 3, bike)$, $(red, 4, car)$, $(green, 1, truck)$, $(green, 2, car)$, $(green, 3, truck)$, $(green, 4, bike)$, $(yellow, 1, bike)$, $(yellow, 2, bike)$, $(yellow, 3, car)$, $(yellow, 4, truck)$ cover all pairs of input values. That is, all 12 combinations of color and number appear; so do all 12 combinations of number and vehicle and all 9 combinations of color and vehicle.

The third meaning of model-based testing is the generation of executable test cases that include *oracle* information, such as the expected output values of the SUT, or some automated check on the actual output values to see if they are correct. This is obviously a more challenging task than just generating test input data or test sequences that call the SUT but do not check the results. To generate tests with oracles, the test generator must know enough about the expected behavior of the SUT to be able to predict or check the SUT output values. In other words, with this definition of model-based testing, the model must describe the expected *behavior* of the SUT, such as the relationship between its inputs and outputs. But the advantage of this approach is that it is the only one of the four that addresses the whole test design problem from choosing input values and generating sequences of operation calls to generating executable test cases that include verdict information.

The fourth meaning of model-based testing is quite different: it assumes that we are given a very abstract description of a test case, such as a UML sequence diagram or a sequence of high-level procedure calls, and it focuses on transforming that abstract test case into a low-level test script that is executable. With this approach, the model is the information about the structure and API (application programming interface) of the SUT and the details of how to transform a high-level call into executable test scripts. We discuss this process in more detail in Chapter 8.

The main focus of this book (Chapters 3 to 7 and 9 and 10) is the third meaning of model-based testing: the generation of executable test cases that include oracle information, based on models of the SUT behavior. This generation process includes the generation of input values and the sequencing of calls into test sequences, but it also includes the generation of oracles that check the SUT outputs. This kind of model-based testing is more sophisticated and complex than the other meanings, but it has greater potential paybacks. It can automate the complete test design process, given a suitable model, and produces complete test sequences that can be transformed into executable test scripts.

With this view of model-based testing, we define model-based testing as *the automation of the design of black-box tests*. The difference from the usual black-box testing is that rather than manually writing tests based on the requirements documentation, we instead create a *model* of the expected SUT behavior, which captures some of the requirements. Then the model-based testing tools are used to automatically generate tests from that model.

Key Point Model-based testing is the automation of the design of black-box tests.

That leads us to two questions: What is a model? What notation should we use to write models? Here are two illuminating definitions of the word *model*, from the *American Heritage Dictionary* [Ame00]:

- A small object, usually built to scale, that represents in detail another, often larger object.
- A schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics.

These definitions show the two most important characteristics of models that we want for model-based testing: the models must be small in relation to the size of the system that we are testing so that they are not too costly to produce, but they must be detailed enough to accurately describe the characteristics that we want to test. A UML class diagram or an informal use case diagram by itself is not precise or detailed enough for model-based testing; some description of the dynamic behavior of the system is needed. Yes, these two goals (small, detailed) can be in conflict at times. This is why it is an important engineering task to decide which characteristics of the system should be modeled to satisfy the test objectives, how much detail is useful, and which modeling notation can express those characteristics most naturally. Chapter 3 gives an introduction to various kinds of modeling notations and discusses guidelines for writing effective models for testing purposes.

Once we have a model of the system we want to test, we can then use one of the model-based testing tools to automatically generate a test suite from the model. There are quite a few commercial and academic model-based testing tools available now, based on a variety of methods and notations. Many of the tools allow the test engineer to guide the test generation process to control the number of tests produced or to focus the testing effort on certain areas of the model.

The output of the test case generator will be a set of *abstract test cases*, each of which is a sequence of operations with the associated input values and the expected output values (the *oracle*). That is, the generated test cases will be expressed in terms of the abstract operations and values used by the model.

The next step is to *transform* (concretize) these abstract test cases into executable test scripts. This may be done by the model-based testing tool, using some templates and translation tables supplied by the test engineer. The resulting executable tests may be produced directly in some programming language, such as JUnit tests in Java, or in a dynamic language such as Tcl or Python, or in a dedicated test scripting language. These executable

test scripts can then be executed to try to detect failures in the SUT. The execution of the tests may be controlled and monitored by a *test execution* tool—different varieties of these tools are available for various types of SUT. The process of transforming the abstract test cases into executable tests and executing them is covered in Chapter 8.

In the next chapter, we will discuss the benefits and limitations of model-based testing and its impact on the software life cycle. But before that, let us look at a realistic example of model-based testing to get a clearer picture of what it involves.

1.3 A SMART CARD EXAMPLE

To give a good overview of what model-based testing can do, we will show how we can use it to test a simplified smart card system. Figure 1.3 shows a UML class diagram for this system. We will not describe the system in detail or show the entire model, but we note that it is sophisticated enough to allow each smart card to be used for a variety of applications, such as banks and retailers, and it even supports loyalty programs.

The `Smartcard` class in Figure 1.3 is annotated with ≪SUT≫ to indicate that it is the SUT. A `Smartcard` instance can contain several `Applications` and one of those `Applications` may be the `selectedApplication`. Each `Application` can be either an `EPurse` application, which stores real money, or a `Loyalty` application, which keeps track of reward points and prizes such as free trips. Finally, each `Application` has one or more `Profiles` associated with it that define the PINs (personal identification numbers) needed to use that `Application` and indicate whether those PINs have been entered correctly.

This class diagram by itself is not enough. It tells us the classes that make up the system, their data fields, and the signatures of the methods, but it says nothing about the behavior of those methods. To get a UML model that is suitable for model-based testing, we need to add details about the *behavior* of the methods shown in the class diagram.

UML offers many ways and notations for specifying behavior. The two that we will illustrate in this example are *OCL postconditions* and *state machine diagrams*. We will explain these notations more fully in Chapter 3, so here we have just a brief introduction.

OCL (Object Constraint Language) is a textual notation, somewhat similar to the expression part of a programming language, that can be used to define constraints within class diagrams or specify preconditions or postconditions for operations. For example, here is an OCL postcondition to specify

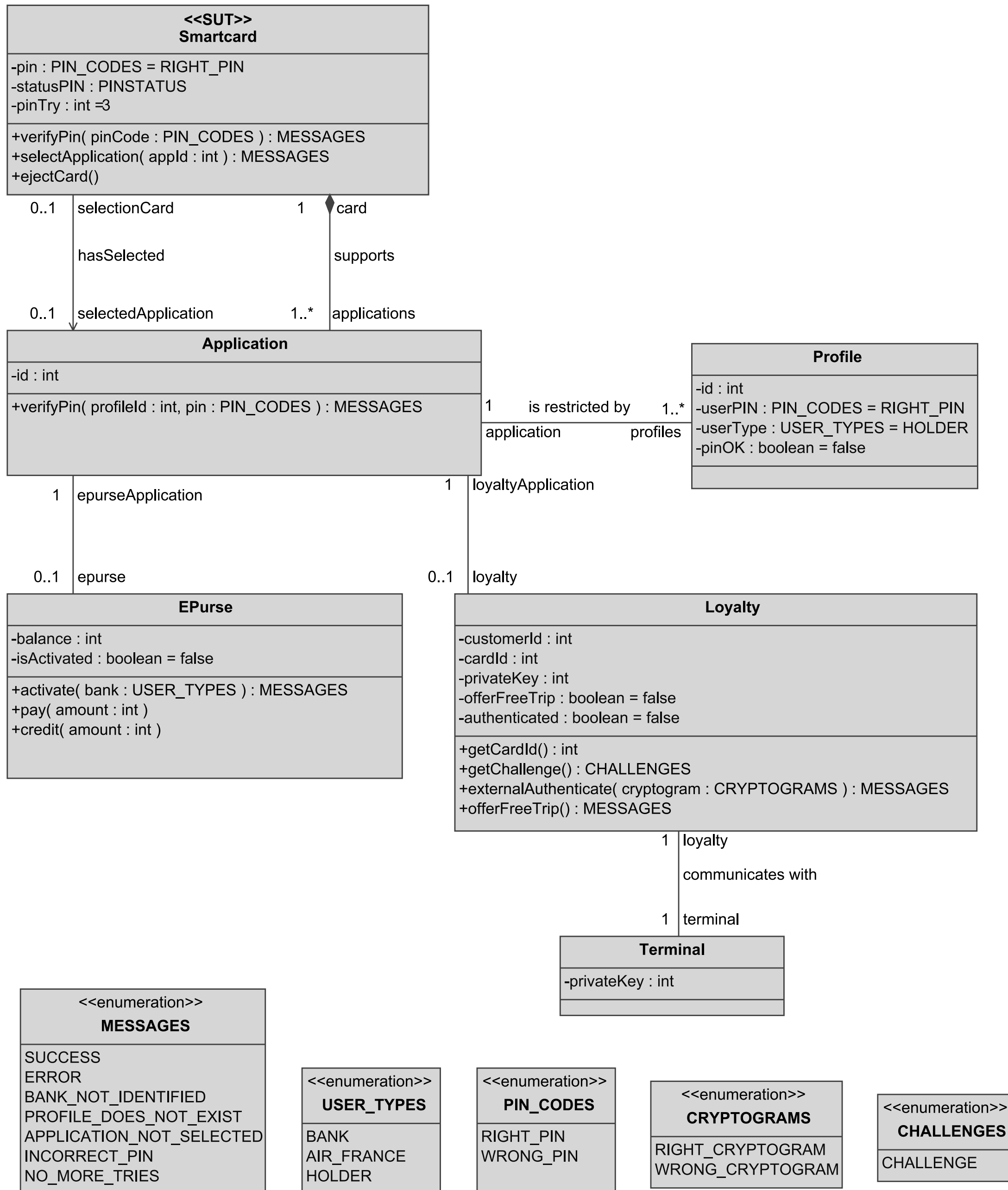

FIGURE 1.3 UML class diagram for the smart card system.

the behavior of the *credit* operation of the `EPurse` class. Note that the expression `self.balance@pre` gives the value of the `balance` field before the method executes, so this postcondition says that the balance is increased by `amount` if this EPurse is activated, but is left unchanged otherwise.

```
post: if (self.isActivated) then
          self.balance = self.balance@pre + amount
      else
          self.balance = self.balance@pre
      endif
```

Here is a more complex example, for the *activate* operation of the `EPurse` class. It illustrates some of the OCL operators for collections, which allow us to write expressive but concise conditions. The expression `self.epurseApplication.profiles` navigates through the class diagram, starting from the current EPurse object and returning its collection of Profile objects. Then the `.userType->excludes(bank)` part checks that none of those Profile objects are BANK profiles. The second `if` generates the same set of Profile objects and then checks that there is a BANK profile among them and that a valid PIN has been entered for that BANK profile.

```
post: if (self.epurseApplication.profiles.userType
              ->excludes(USER_TYPES::BANK)) then
          result = MESSAGES::ERROR
      else
          if (self.epurseApplication.profiles->
                 exists(p:Profile | p.userType
                 = USER_TYPES::BANK and p.pinOK)) then
              result = MESSAGES::SUCCESS and self.isActivated
          else
              result = MESSAGES::BANK_NOT_IDENTIFIED
          endif
      endif
```

Some class behaviors are better specified with *state machine diagrams*. UML state machine diagrams represent the various states that an object may be in and the transitions between those states. Figure 1.4 shows a state machine for the `Smartcard` class. Note that some transitions have labels in the form *Event*[*Guard*]/*Action*, which means that the transition is triggered by *Event* but is taken only if *Guard* evaluates to true. When the transition is taken, then *Action* (which is written in OCL in this example) can specify how the instance variables of the class are modified. The *Guard* and *Action* parts are optional, so the `ejectCard()` event is always enabled and has an

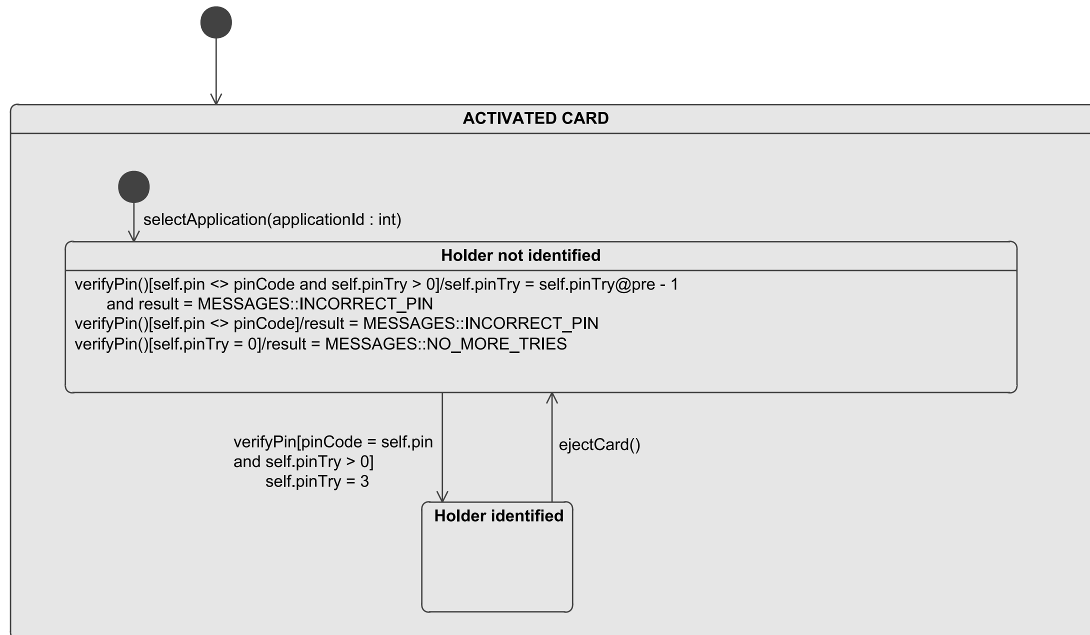

FIGURE 1.4 UML state machine for the `Smartcard` class.

empty action that does not change any variables. The transitions within the "Holder not identified" state are *self-transitions* that can change state variables and produce outputs, but do not move to a different state.

In addition to these details about the behavior of objects, it is useful to give a concrete scenario for testing purposes. Figure 1.5 shows a UML object diagram that specifies a single smart card that can interact with a bank and an airline.

After specifying all these elements, we can finally use a model-based testing tool to automatically generate some tests. We can choose various criteria to determine how many tests we want. For example, we might choose *all-transitions* coverage for a state machine to get a basic set of tests or *all-transition-pairs* coverage to get a larger set of tests that tests interactions between adjacent pairs of transitions. For the methods specified with OCL postconditions, we could choose basic *branch* coverage to make sure that all branches of if-then-else constructs are tested or something like *Modified Condition/Decision Coverage* (MC/DC) to give a larger set of tests that tests each condition more thoroughly and independently. Most tools offer a variety of coverage criteria or allow you to specify manually exactly which class or method or sequence of events you want to test.

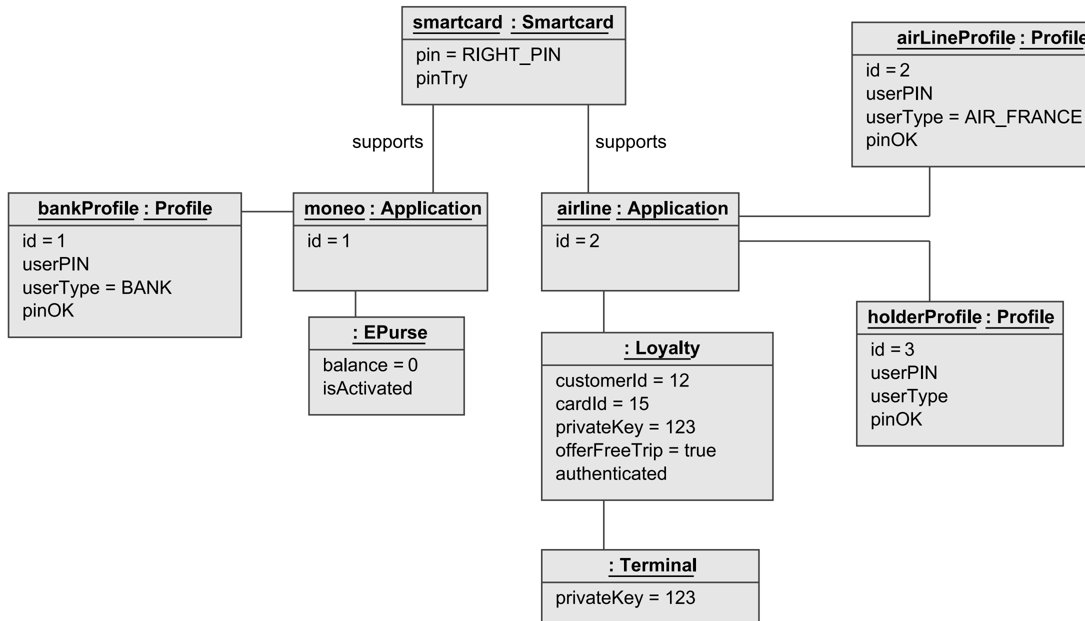

FIGURE 1.5 UML object diagram that defines an initial state.

For example, if we use the LTG/UML tool from LEIRIOS[5] Technologies, with its default settings of branch coverage and transition coverage, we get 29 abstract tests. Each abstract test is a short sequence of operation calls with appropriate input values, plus the expected output value of the method if it has an output. In addition, some oracle information is included (shown in italics in the following sample test), saying how the internal state of each object should have changed after each method call.

Here is one of the abstract tests that is generated, showing a successful withdrawal of one cent from the `moneo` application within the card. We write the actual test invocations in bold, where $Obj.M(\ldots) = R$ means that method M of object Obj should be called and its expected return value is R.

```
test001 =
    moneo.verifyPin(pin=RIGHT_PIN, profileId=1)=SUCCESS;
    bankProfile={id=1, userPIN=RIGHT_PIN, pinOK=true,
                        userType=BANK}
```

[5]LEIRIOS Technologies is a software company that produces model-based testing tools and other model-based testing services. LTG/UML is the LEIROS Test Generator tool for UML.

```
moneo.epurse.activate(bank=BANK)=SUCCESS;
moneo.epurse={isActivated=true, balance=0}
moneo.epurse.credit(amount=1);
moneo.epurse={isActivated=true, balance=1}
moneo.epurse.pay(amount=1);
moneo.epurse={isActivated=true, balance=0}
```

Why does this test do *credit*(1) before the *pay*(1) call? Because in this model, the precondition of *pay* is that *balance* >= amount, so the model-based test generation determined that it was necessary to increase the initial balance (which was 0) before the *pay* method could be tested. Similarly, it generated the necessary setup code to verify that the PIN was correct and to activate the bank's EPurse.

We say that this is an *abstract* test because it is written using the concepts, classes, and values of the model rather than the real SUT. The actual concrete API of the SUT may be quite different from the abstract model. So we need to define how each abstract method maps to the concrete operations of the system, how the abstract data values should be transformed into concrete data, and whether we can use query methods to observe some of the internal states of the objects. Most model-based testing tools provide some way to define these mappings, either by writing an *adaptation layer* that gives the SUT an interface similar to the model or by writing some templates in the target language for your tests so that the model-based testing tool can translate all the abstract tests into executable code.

For example, the preceding abstract test could be translated into a scripting language, such as Tcl, or into C++ or Java. Listing 1.1 shows how it might look if we translated it into Java code that uses the JUnit testing framework[6] (we assume that the `RIGHT_PIN` constant is mapped to 3042 and that we have query methods for accessing the `pinOK` and `id` fields of the `Profile` class and the `balance` field of the `EPurse` class, but not for accessing other data fields).

This example illustrates some of the tradeoffs of model-based testing. The tests do not come for free; there is the cost of writing the model or

[6]JUnit is a widely used Java library for unit testing of Java applications. See *http://www.junit.org* for details and downloads.

LISTING 1.1 The JUnit test script generated from `test001`.

```
public class  extends junit.framework.TestCase
{
    private Smartcard smartcard;
    private Application moneo;
    private Application airline;

    public setUp()
    {
        /* Code to set up the initial network of objects */
        ... reinitializes smartcard, moneo, airline, etc.
    }

    public void test001()
    {
        MESSAGES result = moneo.verifyPin(3042,1);
        assertEqual(result, MESSAGES.SUCCESS);
        assertTrue(moneo.getProfile().pinOk());
        assertTrue(moneo.getProfile().getId() == 1);

        result = moneo.getEPurse().activate(USER_TYPES.BANK);
        assertEqual(result, MESSAGES.SUCCESS);
        assertEqual(moneo.getEPurse.getBalance() == 0);

        moneo.epurse.credit(1);
        assertEqual(moneo.getEPurse.getBalance() == 1);

        moneo.epurse.pay(1);
        assertEqual(moneo.getEPurse.getBalance() == 0);
    }

    // ...the other 28 tests...
}
```

at least of making an existing model precise enough so that it can be used for model-based testing. In this example, it is likely that the class diagram already existed but that the OCL postconditions for eight methods, the state machine diagram, and the object diagram were developed specifically for the model-based testing. So the cost of using model-based testing is mostly the

time to write these 2 or 3 pages of model details. The benefit is that we can then automatically obtain a comprehensive set of 29 executable tests that cover all the different behaviors of each method and class.

The cost of writing these 10 to 15 pages of tests manually would have been greater than the additions we made to the model, and the coverage of the manually written test set would probably be less systematic. Furthermore, with model-based testing we can easily generate a larger test suite from the same model or regenerate our test suite each time we change the system requirements and therefore the model.

Other advantages of the model-based approach are discussed in the next chapter. We will also see there how model-based testing changes the software life cycle and the software development process.

1.4 SUMMARY

Model-based testing is the automation of black-box test design. A model-based testing tool uses various test generation algorithms and strategies to generate tests from a behavioral model of the SUT.

The model must be concise and precise: concise so that it does not take too long to write and so that it is easy to validate with respect to the requirements but precise enough to describe the behavior that is to be tested.

Test cases (including test data and oracles) can be automatically generated from the model using a model-based testing tool. The test engineer can also control the tool to focus the testing effort and manage the number of tests that are generated.

The tests produced from the model are abstract tests, so they must be transformed into executable tests. This also requires some input from the test engineer, but most model-based testing tools provide assistance with this process.

1.5 FURTHER READING

To understand model-based testing, it helps to have a good knowledge of general testing practices and techniques, so we will start by recommending a few good testing books.

Myers's book, *The Art of Software Testing* [Mye79], is a classic in the testing area—some sections are now a little out of date, but the first few chapters are well worth reading, and it has good descriptions of some widely used

test design strategies, such as cause-effect and boundary testing.[7] Beizer's book [Bei90] covers a range of manual test design techniques from models, including path testing, data-flow testing, logic-based testing, graph-based testing, and state-machine-based testing. Whittaker's book, *How to Break Software* [Whi03], contains a series of testing techniques, called "attacks," that target common software errors. The list is based on an empirical analysis of a large number of bugs found in commercial software. Copeland's book [Cop04] gives an overview of classic and recent test design techniques. He does not explicitly introduce model-based testing, but many of the test design techniques that he discusses are used by model-based testing tools to automate test case generation.

Binder's 1200-page book on testing object-oriented systems [Bin99] is the biggest and best guide to designing and automating tests for object-oriented applications. He explains that, in one sense, all testing must be model-based, whether the model is just in the tester's mind or is a sketch on paper or is a formal model that can be analyzed by tools (which is the focus of our book). He covers a wide variety of techniques for designing tests from state machine models, combinational logic and the Unified Modeling Language (UML), plus a large library of patterns for writing tests. The scope of his book is much broader than ours because we focus more on *tool-supported* model-based testing and on the kinds of models and test generation techniques that we have found to work best with automated tools.

For more information about model-based testing, we suggest that you go online to the website associated with this book (*http://www.cs.waikato.ac.nz/~markut.mbt*). There you will find up-to-date links to other model-based testing resources and lists of model-based testing tools and examples.

If you want to know more about the theory behind model-based testing, the book *Model-Based Testing of Reactive Systems* (BJK+05) is a valuable collection of survey papers on testing of finite state machines, testing of labeled transition systems, model-based test case generation, tools and case studies, and so forth.

[7] A second edition (2004) updates some chapters and covers new kinds of testing such as extreme testing and Internet testing.