Producing high-quality software means providing users with products that meet their needs and expectations. The essence of quality is customer satisfaction. In this part, different dimensions of software quality and testing are discussed. Testing may be conducted for different reasons. Quality assurance testing looks for potential problems in a proposed design. Usability testing, on the other hand, tests how well the interface fits user needs and expectations. To ensure user satisfaction with the finished product, user satisfaction must be measured along the way as the design takes form. In this part, we also study how to develop test plans for quality assurance, user satisfaction, and software usability. Part V consists of Chapters 13 and 14.
Chapter Objectives
You should be able to define and understand

- Testing strategies.
- The impact of an object orientation on testing.
- How to develop test cases.
- How to develop test plans.

13.1 INTRODUCTION
To develop and deliver robust systems, we need a high level of confidence that [2]

- Each component will behave correctly.
- Collective behavior is correct.
- No incorrect collective behavior will be produced.

Not only do we need to test components or the individual objects, we also must examine collective behaviors to ensure maximum operational reliability. Verifying components in isolation is necessary but not sufficient to meet this end [2].

In the early history of computers, live bugs could be a problem (see Bugs and Debugging). Moths and other forms of natural insect life no longer trouble digital computers. However, bugs and the need to debug programs remain.

In a 1966 article in Scientific American, computer scientist Christopher Strachey wrote,

Although programming techniques have improved immensely since the early years, the process of finding and correcting errors in programming—"debugging" still remains a most difficult, confused and unsatisfactory operation. The chief impact of this state of affairs is psychological. Although we are happy to pay lip service to the adage that to err is human, most of us like to make a small private reservation about our own perfor-
Bugs and Debugging

The use of the term bug in computing has been traced to Grace Murray Hopper during the final days of World War II. On September 9, 1945, she was part of a team at Harvard University, working to build the Mark II, a large relay computer (actually a room-size electronic calculator). It was a hot summer evening and the Mark II's developers had the window open. Suddenly, the device stopped its calculations. The trouble turned out to involve a flip-flop switch (a relay). When the defective relay was located, the team found a moth in it (the first case of a "bug"). "We got a pair of tweezers," wrote programmer Hopper. "Very carefully we took the moth out of the relay, and put it in the logbook, and put scotch tape over it."

After that, whenever Howard Aiken asked if a team was "making any numbers," negative responses were given with explanation "we are debugging the computer."

mance on special occasions when we really try. It is somewhat deflating to be shown publicly and incontrovertibly by a machine that even when we do try, we in fact make just as many mistakes as other people. If your pride cannot recover from this blow, you will never make a programmer.

Although three decades have elapsed since these lines were written, they still capture the essence and mystique of debugging.

Precisely speaking, the elimination of the syntactical bug is the process of debugging, whereas the detection and elimination of the logical bug is the process of testing. Gruenberger writes,

The logical bugs can be extremely subtle and may need a great deal of effort to eliminate them. It is commonly accepted that all large software systems (operating or application) have bugs remaining in them. The number of possible paths through a large computer program is enormous, and it is physically impossible to explore all of them. The single path containing a bug may not be followed in actual production runs for a long time (if ever) after the program has been certified as correct by its author or others. [10]

In this chapter, we look at the testing strategies, the impact of an object orientation on software quality, and some guidelines for developing comprehensive test cases and plans that can detect and identify potential problems before delivering the software to its users. After all, defects can affect how well the software satisfies the users' requirements. Chapter 14 addresses usability and user satisfaction tests.

13.2 QUALITY ASSURANCE TESTS

One reason why quality assurance is needed is because computers are infamous for doing what you tell them to do, not necessarily what you want them to do. To close this gap, the code must be free of errors or bugs that cause unexpected re-
sults, a process called debugging. **Debugging** is the process of finding out where something went wrong and correcting the code to eliminate the errors or bugs that cause unexpected results. For example, if an incorrect result was produced at the end of a long series of computations, perhaps you forgot to assign the correct value to a variable, chose the wrong operator, or used an incorrect formula.

Testing and searching for bugs is a balance of science, art, and luck. Sometimes, the error is obvious: The bug shows its hideous head the first time you run the application. Other bugs are stealthy and might not surface until a method receives a certain value or until you take a closer look at the output and find out that the results are off by a factor of a certain fraction or the middle initials in a list of names are wrong. There are no magic tricks to debugging; however, by selecting appropriate testing strategies and a sound test plan, you can locate the errors in your system and fix them using readily available debugging tools. A software debugging system can provide tools for finding errors in programs and correcting them. Let us take a look at the kinds of errors you might encounter when you run your program:

- **Language (syntax) errors** result from incorrectly constructed code, such as an incorrectly typed keyword or some necessary punctuation omitted. These are the easiest types of errors to detect; for the most part, you need no debugging tools to detect them. The very first time you run your program the system will report the existence of these errors.

- **Run-time errors** occur and are detected as the program is running, when a statement attempts an operation that is impossible to carry out. For example, if the program tries to access a nonexistent object (say, a file), a run-time error will occur.

- **Logic errors** occur when code does not perform the way you intended. The code might be syntactically valid and run without performing any invalid operations and yet produce incorrect results. Only by testing the code and analyzing the results can you verify that the code performs as intended. Logic errors also can produce run-time errors.

The elimination of the syntactical bug is the process of debugging, whereas the detection and elimination of the logical bug is the process of testing. As you might have experienced by now, logical errors are the hardest type of error to find.

**Quality assurance testing** can be divided into two major categories: error-based testing and scenario-based testing. **Error-based testing** techniques search a given class's method for particular clues of interests, then describe how these clues should be tested. For example, say we want to test the payrollComputation method of an Employee class: `Employee.computePayroll(hours)`. To test this method, we must try different values for `hours` (say, 40, 0, 100, and -10) to see if the program can handle them (this also is known as testing the boundary conditions). The method should be able to handle any value; if not, the error must be recorded and reported. Similarly, the technique can be used to perform integration testing by testing the object that processes a message and not the object that sends the message.
Scenario-based testing, also called usage-based testing, concentrates on what the user does, not what the product does. This means capturing use cases and the tasks users perform, then performing them and their variants as tests. These scenarios also can identify interaction bugs. They often are more complex and realistic than error-based tests. Scenario-based tests tend to exercise multiple subsystems in a single test, because that is what users do. The tests will not find everything, but they will cover at least the higher visibility system interaction bugs [12].

13.3 TESTING STRATEGIES

The extent of testing a system is controlled by many factors, such as the risks involved, limitations on resources, and deadlines. In light of these issues, we must deploy a testing strategy that does the “best” job of finding defects in a product within the given constraints. There are many testing strategies, but most testing uses a combination of these: black box testing, white box testing, top-down testing, and bottom-up testing. However, no strategy or combination of strategies truly can prove the correctness of a system; it can establish only its “acceptability.”

13.3.1 Black Box Testing

The concept of the black box is used to represent a system whose inside workings are not available for inspection [16]. In a black box, the test item is treated as “black,” since its logic is unknown; all that is known is what goes in and what comes out, or the input and output (see Figure 13–1). Weinberg describes writing a user manual as an example of a black box approach to requirements. The user manual does not show the internal logic, because the users of the system do not care about what is inside the system.

In black box testing, you try various inputs and examine the resulting output; you can learn what the box does but nothing about how this conversion is implemented [15]. Black box testing works very nicely in testing objects in an object-oriented environment. The black box testing technique also can be used for scenario-based tests, where the system’s inside may not be available for inspection but the input and output are defined through use cases or other analysis information.

**FIGURE 13–1**
The black box is an imaginary box that hides its internal workings.
13.3.2 White Box Testing

White box testing assumes that the specific logic is important and must be tested to guarantee the system’s proper functioning. The main use of the white box is in error-based testing, when you already have tested all objects of an application and all external or public methods of an object that you believe to be of greater importance (see Figure 13–2). In white box testing, you are looking for bugs that have a low probability of execution, have been carelessly implemented, or were overlooked previously [3].

One form of white box testing, called path testing, makes certain that each path in a object’s method is executed at least once during testing. Two types of path testing are statement testing coverage and branch testing coverage [3]:

- **Statement testing coverage.** The main idea of statement testing coverage is to test every statement in the object’s method by executing it at least once. Murray states, “Testing less than this for new software is unconscionable and should be criminalized” [quoted in 2]. However, realistically, it is impossible to test a program on every single input, so you never can be sure that a program will not fail on some input.

- **Branch testing coverage.** The main idea behind branch testing coverage is to perform enough tests to ensure that every branch alternative has been executed at least once under some test [3]. As in statement testing coverage, it is unfeasible to fully test any program of considerable size.

Most debugging tools are excellent in statement and branch testing coverage. White box testing is useful for error-based testing.

13.3.3 Top-Down Testing

Top-down testing assumes that the main logic or object interactions and systems messages of the application need more testing than an individual object’s methods or supporting logic. A top-down strategy can detect the serious design flaws early in the implementation.

In theory, top-down testing should find critical design errors early in the testing process and significantly improve the quality of the delivered software because of

**FIGURE 13–2**

In a white-box testing strategy, the internal workings are known.
the iterative nature of the test [6]. A top-down strategy supports testing the user interface and event-driven systems.

Testing the user interface using a top-down approach means testing interface navigation. This serves two purposes, according to Conger. First, the top-down approach can test the navigation through screens and verify that it matches the requirements. Second, users can see, at an early stage, how the final application will look and feel [6]. This approach also is useful for scenario-based testing. Top-down testing is useful to test subsystem and system integration.

**13.3.4 Bottom-Up Testing**

**Bottom-up testing** starts with the details of the system and proceeds to higher levels by a progressive aggregation of details until they collectively fit the requirements for the system. This approach is more appropriate for testing the individual objects in a system. Here, you test each object, then combine them and test their interaction and the messages passed among objects by utilizing the top-down approach.

In bottom-up testing, you start with the methods and classes that call or rely on no others. You test them thoroughly. Then you progress to the next level up: those methods and classes that use only the bottom level ones already tested. Next, you test combinations of the bottom two layers. Proceed until you are testing the entire program. This strategy makes sense because you are checking the behavior of a piece of code before it is used by another. Bottom-up testing leads to integration testing, which leads to systems testing.

**13.4 IMPACT OF OBJECT ORIENTATION ON TESTING**

The impact of an object orientation on testing is summarized by the following [12]:

- Some types of errors could become less plausible (not worth testing for).
- Some types of errors could become more plausible (worth testing for now).
- Some new types of errors might appear.

For example, when you invoke a method, it may be hard to tell exactly which method gets executed. The method may belong to one of many classes. It can be hard to tell the exact class of the method. When the code accesses it, it may get an unexpected value. In a non-object-oriented system, when you looked at

\[
x = \text{computePayroll}();
\]

you had to think about the behaviors of a single function. In an object-oriented environment, you may have to think about the behaviors of base::computePayroll(), of derived::computePayroll, and so on. For a single message, you need to explore (or at least think about) the union of all-distinct behaviors. The problem can be complicated if you have multiple inheritance. However, by applying the design axioms and corollaries of object-oriented design (OOD; see Chapter 9), you can limit the differences in behavior between base and derived classes.

The testing approach is essentially the same in both environments. The problem of testing messages in an object orientation is the same as testing code that takes
a function as a parameter and then invokes it. Marick argues that the process of testing variable uses in OOD essentially does not change, but you have to look in more places to decide what needs testing. Has the plausibility of faults changed? Are some types of fault now more plausible or less plausible? Since object-oriented methods generally are smaller, these are easier to test. At the same time, there are more opportunities for integration faults. They become more likely, more plausible.

13.4.1 Impact of Inheritance in Testing

Suppose you have this situation [12]: The base class contains methods inherited() and redefined() and the derived class redefines the redefined() method.

The derived::redefined has to be tested afresh since it is a new code. Does derived::inherited() have to be retested? If it uses redefined() and the redefined() has changed, the derived::inherited() may mishandle the new behavior. So, it needs new tests even though the derived::inherited() itself has not changed.

If the base::inherited has been changed, the derived::inherited() may not have to be completely tested. Whether it does depends on the base methods; otherwise, it must be tested again. The point here is that, if you do not follow the OOD guidelines, especially if you don’t test incrementally, you will end up with objects that are extremely hard to debug and maintain.

13.4.2 Reusability of Tests

If base::redefined() and derived::redefined() are two different methods with different protocols and implementations, each needs a different set of test requirements derived from the use cases, analysis, design, or implementation. But the methods are likely to be similar. Their sets of test requirements will overlap. The better the OOD, the greater is the overlap. You need to write new tests only for those derived::redefined requirements not satisfied by the base::redefined tests. If you have to apply the base::redefined tests to objects of the class “derived,” the test inputs may be appropriate for both classes but the expected results might differ in the derived class [12].

Marick argues that the simpler a test, the more likely it is to be reusable in subclasses. But simple tests tend to find only the faults you specifically target; complex tests are better at both finding those faults and stumbling across others by sheer luck. There is a trade-off here, one of many between simple and complex tests.

The models developed for analysis and design should be used for testing as well. For example, the class diagram describes relationships between objects; that is, an object of one class may use or contain an object of another class, which is useful information for testing. Furthermore, the use-case diagram and the highest level class diagrams can benefit the scenario-based testing. Since a class diagram shows the inheritance structure, which is important information for error-based testing, it can be used not only during analysis but also during testing.

13.5 TEST CASES

To have a comprehensive testing scheme, the test must cover all methods or a good majority of them. All the services of your system must be checked by at least one
test. To test a system, you must construct some test input cases, then describe how
the output will look. Next, perform the tests and compare the outcome with the
expected output. The good news is that the use cases developed during analysis
can be used to describe the usage test cases. After all, tests always should be
designed from specifications and not by looking at the product! Myers describes
the objective of testing as follows [13]:

- Testing is the process of executing a program with the intent of finding errors.
- A good test case is the one that has a high probability of detecting an as-yet
  undiscovered error.
- A successful test case is the one that detects an as-yet undiscovered error.

13.5.1 Guidelines for Developing Quality Assurance Test Cases

Gause and Weinberg provide a wonderful example to highlight the essence of a
test case. Say, we want to test our new and improved “Superchalk”:

Writing a geometry lesson on a blackboard is clearly normal use for Superchalk. Drawing
on clothing is not normal, but is quite reasonable to expect. Eating Superchalk may
be unreasonable, but the design will have to deal with this issue in some way, in order
to prevent lawsuits. No single failure of requirements work leads to more lawsuits than
the confident declaration. [8, p. 252]

Basically, a test case is a set of what-if questions. Freedman [Weinberg][7] and
Thomas [14] have developed guidelines that have been adapted for the UA:

- Describe which feature or service (external or internal) your test attempts to
cover.
- If the test case is based on a use case (i.e., this is a usage test), it is a good idea
to refer to the use-case name. Remember that the use cases are the source of test
cases. In theory, the software is supposed to match the use cases, not the reverse.
As soon as you have enough of use cases, go ahead and write the test plan for
that piece.
- Specify what you are testing and which particular feature (methods). Then, specify
what you are going to do to test the feature and what you expect to happen.
- Test the normal use of the object’s methods.
- Test the abnormal but reasonable use of the object’s methods.
- Test the abnormal and unreasonable use of the object’s methods.
- Test the boundary conditions. For example, if an edit control accepts 32 charac-
ters, try 33, then try 40. Also specify when you expect error dialog boxes, when
you expect some default event, and when functionality still is being defined.
- Test objects’ interactions and the messages sent among them. If you have de-
veloped sequence diagrams, they can assist you in this process.
- When the revisions have been made, document the cases so they become the
starting basis for the follow-up test.
- Attempting to reach agreement on answers generally will raise other what-if
questions. Add these to the list and answer them, repeat the process until the list
is stabilized, then you need not add any more questions.
• The internal quality of the software, such as its reusability and extendability, should be assessed as well. Although the reusability and extendability are more difficult to test, nevertheless they are extremely important. Software reusability rarely is practiced effectively. The organizations that will survive in the 21st century will be those that have achieved high levels of reusability—anywhere from 70–80 percent or more [5]. Griss [9] argues that, although reuse is widely desired and often the benefit of utilizing object technology, many object-oriented reuse efforts fail because of too narrow a focus on technology rather than the policies set forth by an organization. He recommends an institutionalized approach to software development, in which software assets intentionally are created or acquired to be reusable. These assets then are consistently used and maintained to obtain high levels of reuse, thereby optimizing the organization’s ability to produce high-quality software products rapidly and effectively. Your test case may measure what percentage of the system has been reused, say, measured in terms of reused lines of code as opposed to new lines of code written.

Specifying results is crucial in developing test cases. You should test cases that are supposed to fail. During such tests, it is a good idea to alert the person running them that failure is expected. Say, we are testing a File Open feature. We need to specify the result as follows:

1. Drop down the File menu and select Open.
2. Try opening the following types of files:
   • A file that is there (should work).
   • A file that is not there (should get an Error message).
   • A file name with international characters (should work).
   • A file type that the program does not open (should get a message or conversion dialog box).

### 13.6 TEST PLAN

On paper, it may seem that everything will fall into place with no preparation and a bug-free product will be shipped. However, in the real world, it may be a good idea to use a test plan to find bugs and remove them. A dreaded and frequently overlooked activity in software development is writing the test plan. A test plan is developed to detect and identify potential problems before delivering the software to its users. Your users might demand a test plan as one item delivered with the program. In other cases, no test plan is required, but that does not mean you should not have one. A test plan offers a road map for testing activities, whether usability, user satisfaction, or quality assurance tests. It should state the test objectives and how to meet them.

The test plan need not be very large; in fact, devoting too much time to the plan can be counterproductive. The following steps are needed to create a test plan:

1. **Objectives of the test.** Create the objectives and describe how to achieve them. For example, if the objective is usability of the system, that must be stated and also how to realize it. (Usability testing will be covered in Chapter 14.)
2. **Development of a test case.** Develop test data, both input and expected output, based on the domain of the data and the expected behaviors that must be tested (more on this in the next section).

3. **Test analysis.** This step involves the examination of the test output and the documentation of the test results. If bugs are detected, then this is reported and the activity centers on debugging. After debugging, steps 1 through 3 must be repeated until no bugs can be detected.

All passed tests should be repeated with the revised program, called *regression testing*, which can discover errors introduced during the debugging process. When sufficient testing is believed to have been conducted, this fact should be reported, and testing for this specific product is complete [3].

According to Tamara Thomas [14], the test planner at Microsoft, a good test plan is one of the strongest tools you might have. It gives you the chance to be clear with other groups or departments about what will be tested, how it will be tested, and the intended schedule. Thomas explains that, with a good, clear test plan, you can assign testing features to other people in an efficient manner. You then can use the plan to track what has been tested, who did the testing, and how the testing was done. You also can use your plan as a checklist, to make sure that you do not forget to test any features.

Who should do the testing? For a small application, the designer or the design team usually will develop the test plan and test cases and, in some situations, actually will perform the tests. However, many organizations have a separate team, such as a quality assurance group, that works closely with the design team and is responsible for these activities (such as developing the test plans and actually performing the tests). Most software companies also use *beta testing*, a popular, inexpensive, and effective way to test software on a select group of the actual users of the system. This is in contrast to *alpha testing*, where testing is done by in-house testers, such as programmers, software engineers, and internal users. If you are going to perform beta testing, make sure to include it in your plan, since it needs to be communicated to your users well in advance of the availability of your application in a beta version.

### 13.6.1 Guidelines for Developing Test Plans

As software gains complexity and interaction among programs is more tightly coupled, planning becomes essential. A good test plan not only prevents overlooking a feature (or features), it also helps divide the work load among other people, explains Thomas.

The following guidelines have been developed by Thomas for writing test plans [14]:

- You may have requirements that dictate a specific appearance or format for your test plan. These requirements may be generated by the users. Whatever the appearance of your test plan, try to include as much detail as possible about the tests.
- The test plan should contain a schedule and a list of required resources. List how many people will be needed, when the testing will be done, and what equipment will be required.
• After you have determined what types of testing are necessary (such as black box, white box, top-down, or bottom-up testing), you need to document specifically what you are going to do. Document every type of test you plan to complete. The level of detail in your plan may be driven by several factors, such as the following: How much test time do you have? Will you use the test plan as a training tool for newer team members?

• A configuration control system provides a way of tracking the changes to the code. At a minimum, every time the code changes, a record should be kept that tracks which module has been changed, who changed it, and when it was altered, with a comment about why the change was made. Without configuration control, you may have difficulty keeping the testing in line with the changes, since frequent changes may occur without being communicated to the testers.

• A well-thought-out design tends to produce better code and result in more complete testing, so it is a good idea to try to keep the plan up to date. Generally, the older a plan gets, the less useful it becomes. If a test plan is so old that it has become part of the fossil record, it is not terribly useful. As you approach the end of a project, you will have less time to create plans. If you do not take the time to document the work that needs to be done, you risk forgetting something in the mad dash to the finish line. Try to develop a habit of routinely bringing the test plan in sync with the product or product specification.

• At the end of each month or as you reach each milestone, take time to complete routine updates. This will help you avoid being overwhelmed by being so out-of-date that you need to rewrite the whole plan. Keep configuration information on your plan, too. Notes about who made which updates and when can be very helpful down the road.

13.7 CONTINUOUS TESTING

Software is tested to determine whether it conforms to the specifications of requirements. Software is maintained when errors are found and corrected, and software is extended when new functionality is added to an already existing program. There can be different reasons for testing, such as to test for potential problems in a proposed design or to compare two or more designs to determine which is better, given a specific task or set of tasks.

A common practice among developers is to turn over applications to a quality assurance (QA) group for testing only after development is completed. Since it is not involved in the initial plan, the testing team usually has no clear picture of the system and therefore cannot develop an effective test plan. Furthermore, testing the whole system and detecting bugs is more difficult than testing smaller pieces of the application as it is being developed. The practice of waiting until after the development to test an application for bugs and performance could waste thousands of dollars and hours of time.

Testing often uncovers design weaknesses or, at least, provides information you will want to use. Then, you can repeat the entire process, taking what you have learned and reworking your design, or move onto reprototyping and retesting. Testing must take place on a continuous basis, and this refining cycle must continue.
Real-World Issues on the Agenda

CONTINUOUS TESTING CURES THE LAST-MINUTE CRUNCH

Erin Callaway

Remember cramming for exams in college? As the clock ticked to 4 A.M. and the coffee ran cold, a nagging voice inside your head jeered, "If you studied each day, you'd be asleep right now."

Well, the same is true for testing applications. If you wait until after development to test an application for bugs and performance, you could be wasting thousands of dollars and hours of time. Fortunately, more corporate developers are starting to pay attention to that little voice that's telling them after-the-fact testing just doesn't make sense.

That's what happened at Bankers Trust in 1992. "Our testing was very complete and good, but it was costing a lot of money and would add months onto a project," says Glenn Shimamoto, vice president of technology and strategic planning at the New York bank. In one case, testing added nearly six months to the development of a funds transfer application. The problem: Developers would turn applications over to a quality assurance group for testing only after development was completed. Since the QA group wasn't included in the initial plan, it didn't have a clear picture of its rules until it came time to test.

Software Quality Engineering Corp., a training and consulting company in Jacksonville, Fla., helped Bankers adopt better testing practices. Now, all of Bankers' in-house application-development efforts integrate sophisticated testing plans from day one of the project. They also use automated testing tools.

Why don't more companies incorporate continuous testing in their application development? At Bankers, it was simply the fact that the accounting frameworks did not separate testing from total development costs. Neglect, oversight, and pressure to deliver a product while the business need is hot are some of the other culprits. But, says Ed Weller, a fellow of software processes at Bull HN Information Systems Inc., in Phoenix, "It comes down to a reluctance on the part of a lot of people [and organizations] to take a hard look at failure data."

Weller, a software and hardware process improvement veteran, emphasizes the benefits of formally applying lessons learned from past development mistakes to future projects.

At Bankers, one clear benefit is improved customer satisfaction: Users now say application roll out is "quiet" because the quality of the product is so much better.

But like digging for gold, improving the testing process requires hard work. And results won't be obvious overnight. In fact, it may take two, even three development cycles before the payback really becomes clear.

So if you decide to tackle testing yourself, be prepared to make a long-term commitment. Says Bull's Weller: "This is something you commit to for the rest of your life."


throughout the development process until you are satisfied with the results. During this iterative process, your prototypes will be transformed incrementally into the actual application. The use cases and usage scenarios can become test scenarios and, therefore, will drive the test plans. Here are the steps to successful testing:

- **Understand and communicate the business case** for improved testing.
- **Develop an internal infrastructure** to support continuous testing.
- **Look for leaders** who will commit to and own the process.
- **Measure** and document your findings in a defect recording system.
- **Publicize improvements** as they are made and let people know what they are doing better.
13.8 MYERS'S DEBUGGING PRINCIPLES

I conclude this discussion with the Myers's bug location and debugging principles [13]:

1. Bug Locating Principles
   - Think.
   - If you reach an impasse, sleep on it.
   - If the impasse remains, describe the problem to someone else.
   - Use debugging tools (this is slightly different from Myers's suggestion).
   - Experimentation should be done as a last resort (this is slightly different from Myers's suggestion).

2. Debugging Principles
   - Where there is one bug, there is likely to be another.
   - Fix the error, not just the symptom of it.
   - The probability of the solution being correct drops as the size of the program increases.
   - Beware of the possibility that an error correction will create a new error (this is less of a problem in an object-oriented environment).

13.9 CASE STUDY: DEVELOPING TEST CASES FOR THE VIANET BANK ATM SYSTEM

In Chapter 6, we identified the scenarios or use cases for the ViaNet bank ATM system. The ViaNet bank ATM system has scenarios involving Checking Account, Savings Account, and general Bank Transaction (see Figures 6-9, 6-10, and 6-11). Here again is a list of the use cases that drive many object-oriented activities, including the usability testing:

- Bank Transaction (see Figure 6-9).
- Checking Transaction History (see Figure 6-9).
- Deposit Checking (see Figure 6-10).
- Deposit Savings (see Figure 6-11).
- Savings Transaction History (see Figure 6-9).
- Withdraw Checking (see Figure 6-10).
- Withdraw Savings (see Figure 6-11).
- Valid/Invalid PIN (see Figure 7-4).

The activity diagrams and sequence/collaboration diagrams created for these use cases are used to develop the usability test cases. For example, you can draw activity and sequence diagrams to model each scenario that exists when a bank client withdraws, deposits, or needs information on an account. Walking through the steps can assist you in developing a usage test case.

Let us develop a test case for the activities involved in the ATM transaction based on the use cases identified so far. (See the activity diagram in Figure 6-8 and the sequence diagram of Figure 7-4 to refresh your memory.)
A test case general format is this: *If it receives certain input, it produces certain output.*

1. If a bank client inserts his or her card, the system will respond by requesting the password.
2. If the password is incorrect the system must display the bad password message, eject the card, and request the client to take the card.
3. After the transaction is completed, the system should show the main screen.
4. If a bank client selected an incorrect command when entering the transaction, the system will respond immediately by indicating that some error has been made. The system should allow the customer to correct the mistake and not force him or her to make a withdrawal if the intention was to make a deposit.
5. If an unauthorized customer attempts access to someone else’s account, the system should record the time, date, and if possible the identity of the person (not based on the original use cases).
6. If the cash is low, the system will notify the bank for additional money (not based on the original use cases).
7. If an act of vandalism is committed, the system will sound an alarm and call security (not based on the original use cases).
8. If the customer enters a wrong PIN number, the system will give the customer three more chances to correct it; otherwise, it will abort the operation.

And so on.

This is an iterative process; at every iteration, a new issue will be exposed that will help you refine the system. A positive side effect of a test plan is that it might raise some new questions. As the test cases are performed, new test cases will come up, and you should repeat the test. Gause and Weinberg argue that, even if you never attempt to answer the test plans, just developing them might lead you to other, overlooked issues.

### 13.10 SUMMARY

Testing may be conducted for different reasons. Quality assurance testing looks for potential problems in a proposed design. In this chapter, we looked at guidelines and the basic concept of test plans and saw that, for the most part, use cases can be used to describe the usage test cases. Also, some of the techniques, strategies, and approaches for quality assurance testing and the impact of object orientation on testing are discussed.

Testing is a balance of art, science, and luck. It may seem that everything will fall into place without any preparation and a bug-free product will be shipped. However, in the real world, we must develop a test plan for locating and removing bugs. A test plan offers a road map for testing activity; it should state test objectives and how to meet them. The plan need not be very large; in fact, devoting too much time to the plan can be counterproductive.

There are no magic tricks to debugging; however, by selecting appropriate testing strategies and a sound test plan, you can locate the errors in your system and fix them by utilizing debugging tools.
Once you have created fully tested and debugged classes of objects, you can put them into a library for use or reuse. The essence of an object-oriented system is that you can take for granted that these fully tested objects will perform their desired functions and seal them off in your mind like black boxes.

Testing must take place on a continuous basis, and this refining cycle must continue throughout the development process until you are satisfied with the results.

**KEY TERMS**

Alpha testing (p. 334)
Beta testing (p. 334)
Black box testing (p. 328)
Bottom-up testing (p. 330)
Branch testing coverage (p. 329)
Configuration control system (p. 335)
Debugging (p. 327)
Error-based testing (p. 327)
Language errors (p. 327)
Path testing (p. 329)
Regression testing (p. 334)
Scenario-based testing (p. 328)
Statement testing coverage (p. 329)
Test plan (p. 333)
Top-down testing (p. 329)
Usage-based testing (p. 328)
White box testing (p. 329)

**REVIEW QUESTIONS**

1. What is a successful test when you are looking for bugs?
2. Describe the different testing strategies.
3. What is black box testing?
4. When would you use white box testing?
5. If you do not know what is inside the box, how would you test it?
6. In black box testing, you have to test the normal use of functions, the abnormal but reasonable use of functions, and the abnormal and unreasonable use of functions—why?
7. What is the importance of developing a test case?
8. What is white box testing?
9. What is path testing?
10. What is statement testing coverage?
11. What is branch testing coverage?
12. Describe top-down and bottom-up strategies. Which do you prefer and why?
13. What is regression testing?
14. What is a test plan? What steps are followed in developing a test plan?
15. Why are debugging tools important?
16. What basic activities are performed in using debugging tools?
PROBLEMS

1. How would you decide whether a product you are buying is bug free?
2. Research on the WWW or your local library to write a paper on when software must be tested; make sure to include some real-world examples in your report.
3. Consult the WWW or the library to obtain an article that describes beta testing. What are the advantages and disadvantages of this form of testing? Write a paper based on your findings.
4. Do research on the McCabe theory of software complexity and also the McCabe test tool.

REFERENCES

System Usability and Measuring User Satisfaction

Ninety percent of product development efforts fail. About thirty percent fail to produce anything at all, but most of the failures don’t have that problem. They do produce a product, but people don’t like it. They do not use it at all, or if they do, they may grumble endlessly.

—Donald C. Gause and Gerald Weinberg

Chapter Objectives

You should be able to define and understand

- Usability testing.
- User satisfaction testing.

14.1 INTRODUCTION

Quality refers to the ability of products to meet the users’ needs and expectations. The task of satisfying user requirements is the basic motivation for quality. Quality also means striving to do the things right the first time, while always looking to improve how things are being done. Sometimes, this even means spending more time in the initial phases of a project—such as analysis and design—making sure that you are doing the right things. Having to correct fewer problems means significantly less wasted time and capital. When all the losses caused by poor quality are considered, high quality usually costs less than poor quality.

Two main issues in software quality are validation or user satisfaction and verification or quality assurance (see Chapter 3). There are different reasons for testing. You can use testing to look for potential problems in a proposed design. You can focus on comparing two or more designs to determine which is better, given a specific task or set of tasks. Usability testing is different from quality assurance testing in that, rather than finding programming defects, you assess how well the interface or the software fits the use cases, which are the reflections of users’ needs and expectations. To ensure user satisfaction, we must measure it throughout the system development with user satisfaction tests. Furthermore, these tests can be used as a communication vehicle between designers and end users [3]. In the next section, we look at user satisfaction tests that can be invaluable in developing high-
quality software. In Chapter 12, we learned that the process of designing view layer classes consists of the following steps (see Figure 14–1):

1. **Macro-level UI Design Process**—Identifying view layer objects.
2. **Micro-level UI Design Activities**.
3. Testing usability and user satisfaction.
4. Refining and iterating the design.

In spite of the great effort involved in developing user interfaces and the potential costs of bad ones, many user interfaces are not evaluated for their usability or acceptability to users, risking failure in the field. Bad user interfaces can contribute to human error, possibly resulting in personal and financial damages [2]. Usability should be a subset of software quality characteristics. This means that usability must be placed at the same level as other characteristics, such as reliability, correctness, and maintainability [1]. Usability is an important characteristic in determining the quality of the product. In this chapter, we look at usability testing and issues regarding user satisfaction and its measurement. We study how to develop user satisfaction and usability tests that are based on the use cases. The use cases identified during the analysis phase can be used in testing the design.
Once the design is complete, you can walk users through the steps of the scenarios to determine if the design enables the scenarios to occur as planned.

14.2 USABILITY TESTING

The International Organization for Standardization (ISO) defines *usability* as the effectiveness, efficiency, and satisfaction with which a specified set of users can achieve a specified set of tasks in particular environments. The ISO definition requires

- *Defining tasks.* What are the tasks?
- *Defining users.* Who are the users?
- *A means for measuring effectiveness, efficiency, and satisfaction.* How do we measure usability?

The phrase *two sides of the same coin* is helpful for describing the relationship between the usability and functionality of a system. Both are essential for the development of high-quality software [4]. *Usability testing* measures the ease of use as well as the degree of comfort and satisfaction users have with the software. Products with poor usability can be difficult to learn, complicated to operate, and misused or not used at all. Therefore, low product usability leads to high costs for users and a bad reputation for the developers.

Usability is one of the most crucial factors in the design and development of a product, especially the user interface. Therefore, usability testing must be a key part of the UI design process. Usability testing should begin in the early stages of product development; for example, it can be used to gather information about how users do their work and find out their tasks, which can complement use cases. You can incorporate your findings into the usability test plan and test cases. As the design progresses, usability testing continues to provide valuable input for analyzing initial design concepts and, in the later stages of product development, can be used to test specific product tasks, especially the UI.

Usability test cases begin with the identification of use cases that can specify the target audience, tasks, and test goals. When designing a test, focus on use cases or tasks, not features. Even if your goal is testing specific features, remember that your users will use them within the context of particular tasks. It also is a good idea to run a pilot test to work the bugs out of the tasks to be tested and make certain the task scenarios, prototype, and test equipment work smoothly.

Test cases must include all use cases identified so far. Recall from Chapter 4 that the use case can be used through most activities of software development. Furthermore, by following Jacobson’s life cycle model, you can produce designs that are traceable across requirements, analysis, design, implementation, and testing. The main advantage is that all design traces directly back to the user requirements. Use cases and usage scenarios can become test scenarios; and therefore, the use case will drive the usability, user satisfaction, and quality assurance test cases (see Figure 14-2).
14.2.1 Guidelines for Developing Usability Testing

Many techniques can be used to gather usability information. In addition to use cases, focus groups can be helpful for generating initial ideas or trying out new ideas. A focus group requires a moderator who directs the discussion about aspects of a task or design but allows participants to freely express their opinions.

Usability tests can be conducted in a one-on-one fashion, as a demonstration, or as a “walk through,” in which you take the users through a set of sample scenarios and ask about their impressions along the way. In a technique called the Wizard of Oz, a testing specialist simulates the interaction of an interface. Although these latter techniques can be valuable, they often require a trained, experienced test coordinator [5]. Let us take a look at some guidelines for developing usability testing:

- The usability testing should include all of a software’s components.
- Usability testing need not be very expensive or elaborate, such as including trained specialists working in a soundproof lab with one-way mirrors and sophisticated recording equipment. Even the small investment of tape recorder, stopwatch, and notepad in an office or conference room can produce excellent results.
- Similarly, all tests need not involve many subjects. More typically, quick, iterative tests with a small, well-targeted sample of 6 to 10 participants can identify 80–90 percent of most design problems.
- Consider the user’s experience as part of your software usability. You can study 80–90 percent of most design problems with as few as three or four users if you target only a single skill level of users, such as novices or intermediate level users.
- Apply usability testing early and often.
14.2.2 Recording the Usability Test

When conducting the usability test, provide an environment comparable to the target setting; usually a quiet location, free from distractions is best. Make participants feel comfortable. It often helps to emphasize that you are testing the software, not the participants. If the participants become confused or frustrated, it is no reflection on them. Unless you have participated yourself, you may be surprised by the pressure many test participants feel. You can alleviate some of the pressure by explaining the testing process and equipment.

Tandy Trower, director of the Advanced User Interface group at Microsoft, explains that the users must have reasonable time to try to work through any difficult situation they encounter. Although it generally is best not to interrupt participants during a test, they may get stuck or end up in situations that require intervention. This need not disqualify the test data, as long as the test coordinator carefully guides or hints around a problem. Begin with general hints before moving to specific advice. For more difficult situations, you may need to stop the test and make adjustments. Keep in mind that less intervention usually yields better results. Always record the techniques and search patterns users employ when attempting to work through a difficulty and the number and type of hints you have to provide them.

Ask subjects to think aloud as they work, so you can hear what assumptions and inferences they are making. As the participants work, record the time they take to perform a task as well as any problems they encounter. You may want to follow up the session with the user satisfaction test (more on this in the next section) and a questionnaire that asks the participants to evaluate the product or tasks they performed.

Record the test results using a portable tape recorder or, better, a video camera. Since even the best observer can miss details, reviewing the data later will prove invaluable. Recorded data also allows more direct comparison among multiple participants. It usually is risky to base conclusions on observing a single subject. Recorded data allows the design team to review and evaluate the results.

Whenever possible, involve all members of the design team in observing the test and reviewing the results. This ensures a common reference point and better design solutions as team members apply their own insights to what they observe. If direct observation is not possible, make the recorded results available to the entire team.

To ensure user satisfaction and therefore high-quality software, measure user satisfaction along the way as the design takes form [3]. In the next section, we look at the user satisfaction test, which can be an invaluable tool in developing high-quality software.

14.3 USER SATISFACTION TEST

User satisfaction testing is the process of quantifying the usability test with some measurable attributes of the test, such as functionality, cost, or ease of use. Usability can be assessed by defining measurable goals, such as

- 95 percent of users should be able to find how to withdraw money from the ATM machine without error and with no formal training.
• 70 percent of all users should experience the new function as “a clear improvement over the previous one.”
• 90 percent of consumers should be able to operate the VCR within 30 minutes.

Furthermore, if the product is being built incrementally, the best measure of user satisfaction is the product itself, since you can observe how users are using it—or avoiding it [3]. A positive side effect of testing with a prototype is that you can observe how people actually use the software. In addition to prototyping and usability testing, another tool that can assist us in developing high-quality software is measuring and monitoring user satisfaction during software development, especially during the design and development of the user interface. Gause and Weinberg have developed a user satisfaction test that can be used along with usability testing. Here are the principal objectives of the user satisfaction test [3]:

• As a communication vehicle between designers, as well as between users and designers.
• To detect and evaluate changes during the design process.
• To provide a periodic indication of divergence of opinion about the current design.
• To enable pinpointing specific areas of dissatisfaction for remedy.
• To provide a clear understanding of just how the completed design is to be evaluated.

Even if the results are never summarized and no one fills out a questionnaire, the process of creating the test itself will provide useful information. Additionally, the test is inexpensive, easy to use, and it is educational to both those who administer it and those who take it.

14.3.1 Guidelines for Developing a User Satisfaction Test

The format of every user satisfaction test is basically the same, but its content is different for each project. Once again, the use cases can provide you with an excellent source of information throughout this process. Furthermore, you must work with the users or clients to find out what attributes should be included in the test. Ask the users to select a limited number (5 to 10) of attributes by which the final product can be evaluated. For example, the user might select the following attributes for a customer tracking system: ease of use, functionality, cost, intuitiveness of user interface, and reliability.

A test based on these attributes is shown in Figure 14–3. Once these attributes have been identified, they can play a crucial role in the evaluation of the final product. Keep these attributes in the foreground, rather than make assumptions about how the design will be evaluated [3]. The user must use his or her judgment to answer each question by selecting a number between 1 and 10, with 10 as the most favorable and 1 as the least. Comments often are the most significant part of the test. Gause and Weinberg raise the following important point in conducting a user satisfaction test [3, p. 239]: “When the design of the test has been drafted, show it to the clients and ask, ‘If you fill this out monthly (or at whatever interval), will it enable you to express what you like and don’t like?’ If they answer negatively then find out what attributes would enable them to express themselves and revise the test.”
14.4 A TOOL FOR ANALYZING USER SATISFACTION: THE USER SATISFACTION TEST TEMPLATE

Commercial off-the-shelf (COTS) software tools are already written and a few are available for analyzing and conducting user satisfaction tests. However, here, I have selected an electronic spreadsheet to demonstrate how it can be used to record and analyze the user satisfaction test. The user satisfaction test spreadsheet (USTS) automates many bookkeeping tasks and can assist in analyzing the user satisfaction test results. Furthermore, it offers a quick start for creating a user satisfaction test for a particular project.

Recall from the previous section that the tests need not involve many subjects. More typically, quick, iterative tests with a small, well-targeted sample of 6 to 10 participants can identify 80-90 percent of most design problems. The spreadsheet should be designed to record responses from up to 10 users. However, if there are inputs from more than 10 users, it must allow for that (see Figures 14–4 and 14–5).

One use of a tool like this is that it shows patterns in user satisfaction level. For example, a shift in the user satisfaction rating indicates that something is happening (see Figure 14–6). Gause and Weinberg explain that this shift is sufficient cause to follow up with an interview. The user satisfaction test can be a tool for
### Measuring User Satisfaction

**Project Name:** Customer Tracking System

<table>
<thead>
<tr>
<th>User</th>
<th>Ease of Use</th>
<th>Functionality</th>
<th>Intuitiveness</th>
<th>Cost</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**Comments:**

**General Comments:**

---

**FIGURE 14-4**
A user satisfaction test for a customer tracking system.

---

**FIGURE 14-5**
A user satisfaction summary sheet. The main thing to look for in the responses is changes over time. A shift in the user satisfaction rating indicates that something is happening.

### Measuring User Satisfaction

**Project Name:** Customer Tracking System

<table>
<thead>
<tr>
<th>Test</th>
<th>Period 1</th>
<th>Period 2</th>
<th>Period 3</th>
<th>Period 4</th>
<th>Period 5</th>
<th>Period 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Average</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Overall High</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Overall Low</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Changes</th>
<th>Improvements</th>
<th>Improvements</th>
<th>Problems</th>
<th>Improvements</th>
<th>Improvements</th>
</tr>
</thead>
</table>

**Comments:**
finding out what attributes are important or unimportant. An interesting side effect of developing a user satisfaction test is that you benefit from it even if the test is never administered to anyone; it still provides useful information. However, performing the test regularly helps to keep the user involved in the system. It also helps you focus on user wishes. Here is the user satisfaction cycle that has been suggested by Gause and Weinberg:

1. Create a user satisfaction test for your own project. Create a custom form that fits the project’s needs and the culture of your organization. Use cases are a great source of information; however, make sure to involve the user in creation of the test.

2. Conduct the test regularly and frequently.

3. Read the comments very carefully, especially if they express a strong feeling. Never forget that feelings are facts, the most important facts you have about the users of the system.

4. Use the information from user satisfaction test, usability test, reactions to prototypes, interviews recorded, and other comments to improve the product.

Another benefit of the user satisfaction test is that you can continue using it even after the product is delivered. The results then become a measure of how well users are learning to use the product and how well it is being maintained. They also provide a starting point for initiating follow-up projects [3].

**FIGURE 14-6**
Periodical plotting can reveal shifts in user satisfaction, which can pinpoint a problem. Plotting the high and low responses indicates where to go for maximum information (Gause and Weinberg (3)).
14.5 CASE STUDY: DEVELOPING USABILITY TEST PLANS AND TEST CASES FOR THE VIANET BANK ATM SYSTEM

In Chapter 13, we learned that test plans need not be very large; in fact, devoting too much time to the plans can be counterproductive. Having this in mind let us develop a usability test plan for the ViaNet ATM kiosk by going through the following steps (see Chapter 13).

14.5.1 Develop Test Objectives

The first step is to develop objectives for the test plan. Generally, test objectives are based on the requirements, use cases, or current or desired system usage. In this case, ease of use is the most important requirement, since the ViaNet bank customers should be able to perform their tasks with basically no training and are not expected to read a user manual before withdrawing money from their checking accounts.

Here are the objectives to test the usability of the ViaNet bank ATM kiosk and its user interface:

- 95 percent of users should be able to find out how to withdraw money from the ATM machine without error or any formal training.
- 90 percent of consumers should be able to operate the ATM within 90 seconds.

14.5.2 Develop Test Cases

Test cases for usability testing are slightly different from test cases for quality assurance. Basically, here, we are not testing the input and expected output but how users interact with the system. Once again, the use cases created during analysis can be used to develop scenarios for the usability test. The usability test scenarios are based on the following use cases:

Deposit Checking (see Figure 6–10).
Withdraw Checking (see Figure 6–10).
Deposit Savings (see Figure 6–11).
Withdraw Savings (see Figure 6–11).
Savings Transaction History (see Figure 6–9).
Checking Transaction History (see Figure 6–9).

Next we need to select a small number of test participants (6 to 10) who have never before used the kiosk and ask them to perform the following scenarios based on the use case:

1. Deposit $1056.65 to your checking account.
2. Withdraw $40 from your checking account.
3. Deposit $200 to your savings account.
4. Withdraw $55 from savings account.
5. Get your savings account transaction history.
6. Get your checking account transaction history.

Start by explaining the testing process and equipment to the participants to ease the pressure. Remember to make participants feel comfortable by emphasizing that
you are testing the software, not them. If they become confused or frustrated, it is no reflection on them but the poor usability of the system. Make sure to ask them to think aloud as they work, so you can hear what assumptions and inferences they are making. After all, if they cannot perform these tasks with ease, then the system is not useful.

As the participants work, record the time they take to perform a task as well as any problems they encounter. In this case, we used the kiosk video camera to record the test results along with a tape recorder. This allowed the design team to review and evaluate how the participants interacted with the user interface, like those developed in Chapter 12 (see Chapter 12, Figures 12–18 to 12–21). For example, look for things such as whether they are finding the appropriate buttons easily and the buttons are the right size.

Once the test subjects complete their tasks, conduct a user satisfaction test to measure their level of satisfaction with the kiosk. The format of the user satisfaction test is basically the same as the one we studied earlier in this chapter (see Figure 14–3), but its content is different for the ViaNet bank. The users, uses cases, and test objects should provide the attributes to be included in the test. Here, the following attributes have been selected, since the ease of use is the main issue of the user interface:

- Is easy to operate.
- Buttons are the right size and easily located.
- Is efficient to use.
- Is fun to use.
- Is visually pleasing.
- Provides easy recovery from errors.

Based on these attributes, the test shown in Figure 14–7 can be performed. Remember, as explained earlier, these attributes can play a crucial role in the evaluation of the final product.

14.5.3 Analyze the Tests

The final step is to analyze the tests and document the test results. Here, we need to answer questions such as these: What percentage were able to operate the ATM within 90 seconds or without error? Were the participants able to find out how to withdraw money from the ATM machine with no help? The results of the analysis must be examined.

We also need to analyze the results of user satisfaction tests. The USTS described earlier or a tool similar to it can be used to record and graph the results of user satisfaction tests. As we learned earlier, a shift in user satisfaction pattern indicates that something is happening and a follow-up interview is needed to find out the reasons for the changes. The user satisfaction test can be used as a tool for finding out what attributes are important or unimportant. For example, based on the user satisfaction test, we might find that the users do not agree that the system “is efficient to use,” and it got a low score. After the follow-up interviews, it became apparent that participants wanted, in addition to entering the amount for withdrawal, to be able to select from a list with predefined values (say, $20, $40,
$60, $80, $100, and $200). This would speed up the process at the ATM kiosk.

Based on the result of the test, the UI was modified to reflect the wishes of the users (see Figure 14–8).

You need also to pay close attention to comments, especially if they express a strong feeling. Remember, feelings are facts, the most important facts you have about the users of the system.

14.6 SUMMARY

In this chapter, we looked at different dimensions of quality in software development. We discussed the importance of the usability and user satisfaction tests. The main point here is that you must focus on the users' perception. Many systems that are adequate technically have failed because of poor user perception. This can be prevented, or at least minimized, by utilizing usability and user satisfaction tests as part of your UI design process.

Usability testing begins with defining the target audience and test goals. When designing a test, focus on tasks, not features. Even if your goal is testing specific features, remember that your customers will use them within the context of particular tasks. The use cases identified during analysis can be used in testing your
FIGURE 14-8
The CheckingAccountUI and SavingsAccountUI interface objects for withdrawal modified based on the usability and user satisfaction tests. The users have the option of selecting quick withdrawal or selecting other option and then entering the amount they want to withdraw.

design. Once the design is complete, you can walk users through the steps of the scenarios to determine if the design enables the scenarios to occur as planned.

An interesting side effect of developing user satisfaction tests is that you benefit from it even if the test is never administered to anyone; it still provides useful information. However, performing the test regularly helps keep the user actively involved in the system development. It also helps us stay focused on the users' wishes.
KEY TERMS

Commercial off-the-shelf software (p. 347)
Usability testing (p. 343)
User satisfaction testing (p. 345)

REVIEW QUESTIONS

1. What is validation?
2. What is verification?
3. What is quality?
4. Why do we need usability testing? When do we use usability testing?
5. Why do we need to apply usability testing early on? Why should we apply it often?
6. Why should usability testing include all components of the software?
7. What are some other techniques to gather usability information?
8. Describe the "Wizard of Oz" technique.
9. What is a user satisfaction test?
10. Why do we need to measure user satisfaction?
11. How do you develop a custom form for a user satisfaction test?
12. What is the significance of comments in a user satisfaction form?

PROBLEMS

1. Create a user satisfaction test for a bank system application.
2. Following an elaborate effort on your part to do things right the first time, you decide to develop the world’s best user satisfaction test form (in your opinion!). After testing the forms with the users, you come across the following response, “This form stinks!” What should you do at this point?

REFERENCES

Aiken, Howard, 326
Akenhead, Scott, 256
Alexander, Christopher, 72, 73u–74, 86
Alter, Steven, 87
Anderson, Michael, 12, 56, 87, 149, 176
Appleton, Brad, 87, 216
Atkinson, M.P., 238, 239, 253, 279
B
Bailey, P.J., 279
Bancilhon, F., 279
Beck, Kent, 62, 154, 169, 171n, 172, 176, 188, 196
Beizer, Boris, 340
Bell, Paula, 139, 149, APA:1, APA:25
Bergstrand, John, 12, 56, 87, 149, 176
Berson, Alex, 279
Binder, Robert V., 56, 340
Blaha, Michael, 38, 87, 196, 216
Blum, Bruce I., 43, 44, 56, 139, 149, 216, 340
Bobrow, D., 151, 176
Boehm, B.W., 43, 56, 354
Booch, Grady, 6, 8, 12, 38, 56, 62, 71, 78, 80u–81, 87, 92u–93, 103, 122, 152, 154, 176, 177, 196
Brown, A.L., 279
Brown, J.R., 354
Burnett, Margaret, 12
Buschmann, Frank, 72, 87
C
Callaway, Erin, 56, 336
Capucciati, Maria, 289, 321
Card, D., 57
Carroll, John M., 149
Cattell, Rick, 280
Chin, J.P., 354
Chow, Tszun S., 340
Christo, K.J., 279
Coad, Peter, 31, 38, 57, 62, 71, 87, 162, 176, 179, 196, 207, 210, 216, 340
Cockshott, W.P., 279
Conger, Sue, 330, 340
Coplien, James O., 73, 87
Coronel, Carlos, 280
Cox, B.J., 216
Cunningham, Ward, 62, 154, 169, 171, 171n, 172, 176, 188, 196
D
Dahl, O.J., 38
DeWitt, D., 279
Diehl, V.A., 354
Dittrich, Klaus R., 279
Dollas, A., 57
E
Eddy, Frederick, 38, 87, 196, 216
Edwards, John, 12
Ericsson, Maria, 44u–45, 57, 87, 131, 149, 286, 321
Evans, Charlotte, 139, 149, APA:1, APA:25
F
Finkelstein, Shel, 280
Forsslund, David, 279
Fowler, Martin, 87, 122, 134, 137, 149
Freedman, Daniel P., 332, 340
G
Gabriel, R., 151, 176
Gamma, E., 72, 77, 87
Garfinkel, Simson L., 38
Gause, Donald, 125, 134u–135, 149, 235, 332, 340, 346, 349, 354
Gingrich, P., 103, 122
Goldberg, Adele, 12
Gonzalez, R.C., 153, 176
Graham, Ian, 12
Griss, M.L., 207, 216, 340
Gruenberger, F., 326, 340
H
Harmon, Paul, 122, 176, 196
Helm, R., 72, 87
Hopcroft, John E., 340
Hopper, Grace Murray, 326, 329
Hunter, Richard, 52

I
Irving, Rick, 52
Ishikawa, Sara, 86

J
Jacobson, Agneta, 57, 87, 131, 149, 286, 321
Jacobson, Ivar, 6, 8, 12, 44u–45, 46, 57, 62u–63, 71, 78, 80u–81, 87, 93, 122, 131, 135, 137, 149, 286, 321
Johnson, R., 72, 87

K
Kalakota, Ravi, 247u–248, 279
Kaspar, H., 354
Keller, Marilyn, 149
Kim, Won, 38, 279
King, Gary Warren, 12, 38
Kleyan, M., 103, 122
Kurotsuchi, Brian T., 212, 216

L
LaLonde, Wilf R., 38, 196
Lassesen, Kenneth M., 12
Latta, John, 297
Lee, Juhnyoung, 279
Lewis, Ted, 12
Lipow, M., 354
Lorenson, Bill, 38, 87, 196, 216

M
McClure, Carma, 52
Machiavelli, Niccolo, 39
Macleod, G., 354
Mahoney, Michael K., 38
Maier, D., 279
Marick, Brian, 331, 340
Martin, James, 122, 149, 152, 153, 176, 196, 216
Mellor, Stephen J., 62, 87, 122, 162, 176
Merrit, M., 354
Meunier, Regine, 72, 87
Meyer, Bertrand, 87
Morrison, R., 279
Myers, G.J., 332, 337, 340

N
Norman, K.L., 354
Norman, Ronald, 126, 149, 235
North, Ken, 279
Nygard, K., 38

O
Occam, William of, 203
Odell, James, 122, 149, 152, 153, 176, 196, 216

P
Perkins, S., 57
Premerlani, William, 38, 87, 196, 216
Pugh, John R., 38, 196

R
Rafii, F., 57
Riehle, D., 72, 87
Rob, Peter, 280
Robertson-Dunn, Bernard, 248, 280
Rohrert, Hans, 72, 87
Ross, R., 162, 176
Rumbaugh, James, 6, 8, 12, 38, 62u–63, 78, 80u–81, 87, 93, 122, 179, 196, 216

S
Schmidt, Douglas C., 87
Scott, Kendall, 87, 122, 134, 137, 149
Shaw, M., 155, 176
Shein, Esther, 256
Shimamoto, Glenn, 53, 336
Shlaer, Sally, 62, 87, 122, 162, 176
Short, Keith, 57, 87
Shumate, Ken, 149
Silverstein, Murray, 86
Sommerlad, Peter, 72, 87
Stal, Michael, 72, 87
Strachey, Christopher, 325u–326
Strunk, William, 139
Suh, Nam, 216
Sulaiman, Suziah, 321, 354

T
Taylor, Lloyd, 280
Texel, Putnam, 235
Thomas, Tamara, 332, 337, 340, 354
Tou, J.T., 153, 176
Tristram, Clair, 52
Trower, Tandy, 282, 321, 345
Turban, Efraim, 89, 90, 122

U
Ullman, Jeffrey, 340

V
Veblen, Thorstein, 135
Vlissides, J., 72, 87

W
Watson, Mark, 122, 176, 196
Weinberg, Gerald M., 125, 134u–135, 149, 235, 332, 340, 346, 349, 354
Weller, Ed, 336
Whinston, Andrew, 279
White, J., 139, 151, 176
White, Set, 280
Wiener, Lauren, 62, 87, 154, 155, 176, 179, 189, 196
Wilkerson, Brian, 62, 87, 154, 155, 169, 172, 176, 179, 189, 196
Wilkinson, Stephanie, 297
William of Occam, 203
Williams, Charles B., 235
Williams, Ian, 256
Wirfs-Brock, Rebecca, 62, 87, 154, 155, 176, 179, 189, 196
Wirth, Niklaus, 4, 12

Y
Yourdon, Edward, 31, 38, 57, 62, 71, 87, 162, 176, 179, 196, 207, 210, 216, 340

Z
Zdonik, S., 279
Zullighoven, H., 72, 87
Subject Index

A
Abort, 244
Abstract class, 23, 161
Abstraction, 5
Abstract use case, 69, 133–134
Accelerators, APA:4
Access classes, APA:2, APA:11–12
Access layer, 237–280
classes, 264–268, APA:2–3,
APA:14–17
DBMS; see Database
management systems
(DBMS)
distributed databases; see
Distributed databases
logical and physical databases,
243–244
concurrency policy, 244
shareability, 243–244
transactions, 244
multidatabase systems (MDBS),
260–263
object-oriented DBMS
(OODBMS), 252–255
traditional DBMS vs.,
254–255
object-relational systems; see
Object-relational systems
object store and persistence,
238–239
software development and, 84
ViaNet case study, 269–275
AccountTransactionUI interface
object, 309, 315–316
Actions, 311–316
Activity diagrams, 109–111,
APA:1, APA:4
Actors, 128, 131, 134–136
ViaNet case study, 141–146
Adjective classes, 155
noun phrase approach, 159–160
Aggregation, 27, 99, 178, 182–184
antisymmetry, 182
assembly, 183
collection-member, 184
container, 183
structures, APA:2, APA:3,
APA:13, APA:16, APA:18
transitivity, 182
ViaNet case study, 187–188
Algorithm-centric methodology,
15
Alpha testing, 334
American Express Stored Value
Group, 52
Analysis, 63
object model, 70, 90
phase, OOBE, 71
prototype, 48
Ancestor, 181
Antipatterns, 76
Antisymmetry, 182
A-part-of; see Aggregation
A Pattern Language, 72
Apple Computer Co., 263
Applicable documents, APA:1,
APA:3
Application programming interface
(API), 262–263
Application windows (main
windows), 292–293,
299–300
Assembly, 183, 187
Association, 177, 178–180, APA:2,
APA:3, APA:13–14, APA:16,
APA:18; see also Class diagrams
class, 97–98
common patterns, 179
communication, 179
defined, 178
derived, 180
directed actions (derived), 180
identifying, 179
implementation, 180
location, 179
meta-models and, 118
navigation, 96
objects, 26–27
relationships, ViaNet case study,
185–186
role, 95–96
class diagrams, 95–96
ternary, 180
unnecessary, eliminating, 180
A Timeless Way of Building, 72
Atomic types of objects, 253
Attribute(s), 14; see also Object relationships, attributes, and methods
classes, 155
get, 225
of objects, 17–18
refining, class design, 221–223
reviewing, noun phrase approach, 160–161
set, 225
types, 222
Audience, APA:1, APA:3
Axioms, 285
object-oriented design (OOD), 202–203

B
BankClient AccessUI interface
object, 309, 313
Bankers Trust, 53, 336
Base, 181
Base classes, 23
Behavior, of objects, 18
Behavior diagrams; see Dynamic modeling (behavior diagrams)
Beta testing, 334
Binary association, 95
Binary association notation, 95
Bitmap, APB:3–4
Black box testing, 328
Booch methodology, 65–68
macro process, 66–68
conceptualization, 66
evolution, 67
implementation, 67
maintenance, 67
model analysis/development, 66–67
system architecture
design/creation, 67
micro process, 67–68
Bottom-up approach, 181, 186
Bottom-up testing, 330
Branch testing coverage, 329
Bull HN Information Systems Inc., 336
Business layer
classes, 266–267, APA:2,
APA:11–12, APA:13–14
object analysis, 127
software development and, 83–84
Business object analysis, 127
Business processing, 248
Business process modeling, OOA, 129
C
Candidate classes, noun phrase approach, 156–158
Capturing patterns, 76–77
Cardinality, 26
Check boxes, APB:5
CheckingAccountUI interface
class diagram, 311
Chicago Manual of Style, 139
Clarity, modeling and, 91
Class/classes, 31–32, 265
class diagrams; see Class diagrams designing, 217–235
attributes, refining, 221–223
for ViaNet Bank objects, 223–225
design patterns, 213
groupings, of objects, 16–17
hierarchy, 21–25
dynamic inheritance, 24
inherence, 23–24
identification approaches, 154–174
common patterns; see Common class patterns approach
CRC, 169–172
noun phrase approach; see Noun phrase approach
use-case driven:
sequence/collaboration modeling, 164–169
interface notation, 95
meta-classes, 34–35
methods and protocols; see Methods and protocols, classes
notation: static structure, 94, 95
object-oriented design (OOD)
philosophy, 217–218
packages and, 230–232
pitfalls, avoiding, 226–227
process of, 219
relationship identification,
ViaNet case study, 184
responsibilities, and collaborators (CRC), 169–172
UML OCL, 218–219
visibility of classes; see Visibility of classes
Class diagrams, 94–101
aggregation and composition (a-part-of), 99
association class, 97–98
association role, 95–96
binary association notation, 95
Class diagrams—Cont.
class notation
interface, 95
static structure, 94, 95
generalization, 99–101
multiplicity, 97
n-ary association, 98–99
object diagram, 94
OR association, 97
qualifier, 96–97
Classification, 151–176
class identification approaches; see Class/classes
defined, 152
naming classes, 172–174
theory of, 152–153
Client-server association, 26
Client-server computing; see Distributed databases
Cohesion, 206
Collaboration, 46
Collaboration diagram, UML, 105–106
Collaboration modeling, 164–169
Collaborators, 169–172
Collection-member, 184, 187
Colors, guidelines for using, 300–302
Combo boxes, APB:6–8
Command buttons, 298–299, APB:5
Command menus, 300
Commercial off-the-shelf (COTS) software, 347
Commit, 244
Common class patterns approach, 162–164
caption class, 162
events class, 162, 163
organization class, 162, 163
people (person) class, 162, 163
places class, 162–163
tangible things and devices class, 163, 164
ViaNet case study, 163–164
Common coupling, 205
Common cover, documentation, 139
Common object environment (COE), 251–252
Common object request broker architecture (CORBA), 251–252
Common object request broker architecture (CORBA), 251–252
Common vocabulary, noun phrase approach, 158–159
Communication, use-case diagrams, 103
Database management systems (DBMS)—Cont.
data schema, 242
interface, 242–243
models, 240–241
hierarchical, 240
network, 241
relational, 241
multidatabase systems (M DBS), 260–263
query capabilities, 242–243
structured query language (SQL), 242, 243
views, 240
Database processing, 248
Database schema, 242
Data-centric development, 15
Data coupling, 205
Data definition language (DDL), 242
Data entry windows; see Forms and
data entry windows
Data flow, 64
Data flow diagram (DFD), 93
Data manipulation language
(DML), 242–243
Data query, 254
Data store, 64
Debugging, 327
Deep structures, patterns template,
75
Deployment diagrams, 112–114
Derived association, 180
Derived classes, 23
Design patterns
frameworks vs., 78
object-oriented design (OOD),
212–214
Design phase, OOBE, 71
Design rules, 203
micro level UI; see Micro level
UI design
Design with inheritance (corollary
6), 204, 208–212
multiple inheritance, 211
Destructor method, 225
Dialog boxes, 292, 296–297, APB:3
Digital k, 50
Directed actions association, 180, 185
Discriminator, 100
Distributed component object
model (DCOM), 252
Distributed databases, 245–249
business processing, 248
Distributed databases—Cont.
components, 248
cooperative processing, 248–249
database processing, 248
distributed object computing
(DOC), 250–252
common object request broker
architecture (CORBA), 251–252
Microsoft's Active X/DCOM,
252
distributed processing, 248–249
two-tier architecture, 247
three-tier architecture, 247
user interface, 248
Distributed object computing
(DOC), 250–252
Distributed processing, 248–249
Do (keyword), 108
Documentation, 47, 138–140
Document template, APA:1–25
scope, APA:1, APA:3
applicable documents, APA:1,
APA:3
audience, APA:1, APA:3
organization, APA:1, APA:3
system requirements
specification
activity diagram of business
process, APA:1, APA:4
applicable documents, APA:1
problem statement, APA:4
UML class diagrams; see
Unified modeling
language (UML)
testing, APA:18–25
test cases, APA:18–25
usability testing scenarios,
APA:24
user satisfaction testing,
APA:25
Domain object model, 70, 90
Domain prototype, 48–49
Domain specific method, 225
Dow Jones, 246
Draw message, 19
Drive method, 26
Drop-down combo box, APB:7
Drop-down list, APB:7
Dynamic behavior, patterns
template, 75
Dynamic binding, 34, 254
Dynamic inheritance, 24
Dynamic modeling (behavior
diagrams), 91, 103–114

Database management systems
(DBMS), 237, 239–243
data definition language (DDL),
242
data manipulation language
(DML), 242–243

Correctness, 43
Correspondence, 43
Coupling, 204–206
Cursor, APB:3
Cut-and-paste reusability, 210

D
Dynamic modeling—Cont.
  UML collaboration diagram, 105–106
  UML interaction diagrams, 104–106
  UML sequence diagrams, 104–105

E
  Edit boxes, APB:4
  Editing patterns, 77
  Edit menu, 299
  80–20 rule, for documentation, 139

Elements of Style, 139
Encapsulated objects, 253
Encapsulation, 20–21
Encapsulation leakage, 220
Entry event, 108
Error-based testing, 327
Error messages, view layer, 296–297
Event-driven design, GUI, APB:3
Events, 31, 111, 313–316
class, 162, 163
Evolution, Booch methodology, 67
Exit event, 108
Extended Intelligence, Inc., 52
Extends
  association, 133–134, 145
  use-case diagrams, 103
Extensibility, UML, 116–117
Extensible systems, 254
Externality entity, 64

F
  Facet Decision Systems, Inc., 256
  Failures, hardware/software, 254
  Familiarity, modeling and, 91
  Federated database systems, 261
  Feedback, micro level UI design, 291
  File menu, 299
  Firing, 109
  Fonts, APB:4
guidelines for using, 302
Forces, patterns template, 74, 75
Foreign key, 241
Formal classes, 23, 161
Forms and data entry windows, 292, 293–296
Forward engineering, 255
4th Wave Inc., 297
Frameworks, 77–78
design patterns vs., 78
Fraser Salmon Model, 256
Fuzzy categories, 155–156

G
  Gartner Group Inc., 52
  General Electric, 62
  Generalization, 99–101
class diagrams, 99–101
hierarchy, 177, 181
Generative patterns, 73–74
Generativity, 74
Graphical operating system, APB:1
Graphical user interface (GUI), 281, 282, 297, APB:1–9
event-driven design, APB:3
g graphical operating system, APB:1
objects, APB:3–4
accelerators, APB:4
bitmap, APB:3–4
check boxes, APB:5
combo boxes, APB:6–8
cursor, APB:3
dialog boxes, APB:3
edit boxes, APB:4
fonts, APB:4
icons, APB:3
list boxes, APB:6
menus, APB:3
push buttons, APB:5
radio buttons, APB:5
scroll bars, APB:8
slider buttons, APB:8
spin boxes, APB:8
strings, APB:4
text, APB:8
windows, how it works, APB:1–3

Guard conditions, 111

H
  Hardware failures, 254
  Help window, 300
  Hierarchical models, DBMS, 240
  High-quality software, building, 42–44
  Homogenization, 261
  Horizontal prototype, 48
  How model, 131, 131n

I
  IBM, 38, 50, 321
  Icons, APB:3
  Identity, objects and, 32–34
  If statements, 145
  Implementation, 63
    association, 180
    Booch methodology, 67
diagrams, 111–114
    component diagrams, 112
deployment diagrams, 112–114
  phase, OOBES, 71
use-case approach; see Use-case approach
Inappropriate behaviors, inheriting, 211–212
Incremental testing, 53
Independence axiom, 202
Information axiom, 202
Information hiding, 20–21
Inheritance, 23–24, 253
and software quality testing, 331
table-inherited classes mapping, 258–259
Input, view layer classes, 284
Instance, 16
  navigation, 259–260
  variables, 20
  Interaction diagrams, UML, 104–106
  Interface(s)
    behavior: events and actions,
      APA:3, APA:18
    behavior, ViaNet case study; see View layer: interface objects
  consistency of, 292
  DBMS, 242–243
  forgiving nature of, 291
  objects; see View layer: interface objects
  simplicity (UI design rule 1),
    micro level UI design, 288–290
  transparency and naturalness (UI design rule 2), 290
  visual aspect of, 291
Internal transition compartment, 107
I/O methods, 225
Irrelevant classes, 155
Iterative development, 80

J
  Jacobson methodologies, 68–71
  object-oriented business engineering (OOBE), 71
Jacobson methodologies—Cont.
object-oriented software engineering (OOSE): objectory, 70–71
use cases, 68–69

K
Key mechanisms, patterns template, 75
Keys for instance navigation, 259–260
KISS method, 289
Known uses, patterns template, 75–76

L
Language errors, 327
Large number of simpler classes [reusability (corollary 3)], 204, 206–207
Late binding, 254
Layered approach, 82–84
access layer, 84
business layer, 83–84
user interface (view) layer, 84
Layered architecture, 6
Lifeline, 104–105
List boxes, APB:6
Location association, 179, 185
Logical and physical databases, 243–244
Logic errors, 327
Lower bound, 97

M
Macro development process, Booch methodology; see Booch methodology
Macro level UI design, 285–287, 305–308
Maintenance
Booch methodology, 67
modeling and, 91–92
MainUI object interface, 309, 313–314
Measurable value, 131
Mediators, 269
Menus, APB:3
Messages, objects responding to, 18–20
Meta-classes, 34–35
Meta-data, 239
Meta-model, 117–119
Metaphor, 290
Method, 16–17
Method classes, 265–266, 268, 274
Methodology, object-oriented systems development, 4
Methods, 14, 191–192, APA:2, APA:3, APA:14, APA:16–17; see also Object relationships, attributes, and methods for ViaNet Bank objects; see ViaNet Bank ATM case study
Methods and protocols, classes, 225–227
simplification rules, 226
UML operation presentation, 227
ViaNet Bank objects, 227–230
account class create method, 229–230
account class deposit method, 228–229
account class withdraw method, 229
ATM machine class operations, 230
BankClient class verify password method, 228
checking account class withdraw method, 230
Micro development process, Booch methodology, 67–68
Micro level UI design, 285, 287–292
design rules, 288–292
feedback, immediate, 291
interface, forgiving nature of, 291
interface, visual aspect of, 291
interface simplicity (UI design rule 1), 288–290
interface transparency and naturalness (UI design rule 2), 290
mode avoidance, 292
user control of software (UI design rule 3), 290–292
ViaNet case study, 308–309
Microsoft, 28, 81, 251, 263, 282–283, 294, 334, 345
Active X/DCOM, 252
Middleware, 245
Modal dialog, 292
Mode, 291, 292
Mode avoidance, 292
Model, 231
analysis/development, Booch methodology, 66–67
constraints, UML, 116–117
defined, 89
dependency, 115–116
management, UML, 114–116
Modeling; see also Unified modeling language (UML)
reasons for, 91–92
Motivation, design patterns, 212–213
Multidatabase systems (MDBS), 260–263
open database connectivity, 262–263
Multiple inheritance, 25, 181, 186, 211
Multiplicity, 97
Myers’s debugging principles, 337

N
Name, patterns template, 74
Name compartment, 107
Naming a use case, 137
Naming classes, 172–174
N-ary association, 98–99, 180
Network models, DBMS, 241
Neutralization, 261
Next to, 185
Nonanonymous reviews, patterns, 77
Nongenerative patterns, 73–74
Note, UML, 117
Noun phrase approach, 154–162
adjective classes, 155, 159–160
attribute classes, 155
attributes, possible, reviewing, 160–161
candidate classes, 156–158
class purpose, reviewing, 161–162
common vocabulary, building, 158–159
initial list of noun phrases, 156–158
irrelevant classes, 155
redundant classes, 155, 158–159
selecting classes from relevant and fuzzy categories, 155–156
tentative classes, identifying, 154–155
ViaNet Bank ATM system, 156
Object-oriented design—Cont. 
corollaries—Cont. 
No. 3, large number of simpler classes (reusability), 204, 206-207
No. 4, strong mapping, 204, 207-208
No. 5, standardization, 204, 208
No. 6, design with inheritance, 204, 208-212
design patterns, 212-214
process, 200-202
Object-oriented methodologies, 61-87
Booch methodology; see Booch methodology frameworks, 77-78
Jacobson methodologies; see Jacobson methodologies patterns; see Pattern(s)
Rumbaugh object modeling technique; see Rumbaugh technique
survey of, 62-63
unification of, 61-62
Object-oriented philosophy, 14-15
Object-oriented programming, 14-15
Object-oriented software engineering (OOSE): 
objectory, 70-71
Object-oriented systems development book, organization of, 6-11
life cycle; see Systems development life cycle methodology, 4
object orientation, reasons for, 5-6
orthogonal views of software, 4
overview, 3-12
unified approach (UA); see Unified approach (UA)
Object-oriented user interface (OUI), 282-283
Objectory, 70-71
Object persistence, 34
Object reference, 33
Object-relational systems—Cont. 
table–multiple classes mapping, 258
Object-relation mapping, 256-257
Object relationships, attributes, and methods, 177-196
aggregation, 178
a-part-of-structure, 178
association; see Association attribute and method
identification, 188-189
attribute definition, 189-191
for the account class, 190-191
for the ATM machine class, 191
for the transaction class, 191
ViaNet Bank, 190-191
class responsibility, 188-190
generalization hierarchy, 177
object responsibility: methods and messages, 191-192
UML diagrams and, 192
use cases and, 192
super-sub class relationships, 181-182
super-sub structure, 177
ViaNet Bank ATM system case study, 184-188
aggregation/a-part-of relationships, 187-188
association relationships, 185-186
class relationship identification, 184
super-sub relationships, 186-187
UML class diagram development, 184-185
Object request brokers (ORBs), 251
Object responsibility: methods and messages, 191-192
Object(s) 
aggregations and object containment, 27
attributes, 17-18
behavior and methods, 18
class groupings of, 16-17
class hierarchy, 21-25
dynamic inheritance, 24
inheritance, 23-24
multiple inheritance; see
Multiple inheritance consumer-producer association, 26-27
Object(s)—Cont.
defined, 15–16

dynamic binding, 34

eapsulation and information

hiding, 20–21

GUI; see Objects, GUI

identity and, 32–34

messages, responding to, 18–20

meta-classes, 34–35

object-oriented philosophy, 14–15

payroll case study; see Payroll

case study

polymorphism, 25–26

relationships and associations,

26–27

state and properties of, 17–18

static binding, 34

Objects, GUI, APB:3–4

accelerators, APB:4

bitmap, APB:3–4

check boxes, APB:5

combo boxes, APB:6–8

cursor, APB:3

dialog boxes, APB:3

edit boxes, APB:4

fonts, APB:4

icons, APB:3

list boxes, APB:6

menus, APB:3

push buttons, APB:5

radio buttons, APB:5

scroll bars, APB:8

slider buttons, APB:8

spin boxes, APB:8

strings, APB:4

text, APB:8

Object storage and interoperability;

see Access layer

Object wrapper, 250, 250n

Occam’s razor, 203

OMT dynamic model, Rumbaugh

technique, 63–64

OMT functional model, Rumbaugh

technique, 64

Open database connectivity

(ODBC), 262–263

OR association, 97

Order, 178

Order object, 31

Organization, APA:1, APA:3

Organization class, 162, 163

Originality, disregard of, patterns,

77

Orthogonal views of software, 4

Output, view layer classes, 284–285

P

Packages, APA:18

and classes, management of,

230–232

dividing use cases into, 137

UML, 114–116

ViaNet case study, 146

Palettes, 300

Part of, 185

Part-whole relationship, 99

Path testing, 329

Pattern-Oriented Software

Architecture, 72

Pattern(s), 71–77

antipatterns, 76

capturing, 76–77

components of; see template
defined, 72–73

editing, 77

generative and nongenerative,

73–74

mining, 76

name, 212

nonanonymous reviews, 77

originality, disregard of, 77

practicability, focus on,

76–77

reviews, nonanonymous, 77

shepherding, 77

template, 74–76

context, 74

deep structures, 75

dynamic behavior, 75

eamples, 75

forces, 74, 75

key mechanisms, 75

known uses, 75–76

name, 74

preconditions, 74

problem, 74

proven solution to recurring

problem, 76

rationale, 75

recurring problem, proven

solution to, 76

related patterns, 75

resolution of forces, 75

resulting context, 75

side effects, 75

solution, 75

static structure, 75

thumbnail sketch, 76

trade-offs, 74

thumbnail, 76

writers’ workshops, 77

Payroll case study, 28–32

object-oriented approach,

30–32

structured approach, 28–30

People (person) class, 162, 163

Per-class protection, 20

Per-object protection, 20

Persistence, 237–239, 254

object, 34

Persistent attributes, 266

Persistent data, 238, 266

Persons, 31

Pessimistic policy, 244

Places, 31

Places class, 162–163

Polymorphism, 25–26

Practicability, focus on, patterns,

76–77

Preconditions, patterns template, 74

Premature binding, 254

Primary key, 241

Private protocol (visibility), 220,

221

Problem, patterns template, 74

Problem statement, APA:4

Procedures, 14

Process, 64

Programming techniques, 5

Programs, defined, 4

Properties, 14, 16

Proposed repository, unified

approach (UA), 81–82

Protected protocol (visibility), 220,

221, 223

Protocols, 20–21, 219, 225–227

Proto-pattern, 73

Prototyping, 47–49, 285

analysis prototype, 48

domain prototype, 48–49

horizontal prototype, 48

of the user interface, 302–304

vertical prototype, 48

Proven solution to recurring

problem, patterns template, 76

Public methods, 329

Public protocol (visibility), 220,

221

Push buttons, APB:5

Q

Qualifier, 96–97

Quality assurance; see Software

quality assurance

Query capabilities, 242–243
### Subject Index

#### R
- Radio buttons, APB:5
- Railroad paradox, 134–136
- Rapid application development (RAD), 51–52

#### Rationale
- design patterns, 212–213
- patterns template, 75

#### Receiver
- 18

#### Recurring problem, patterns template, 76

#### Redundant classes, 155, 265, 268
- review of, noun phrase approach, 158–159

#### Referential integrity, 257

#### Regression testing, 334

#### Related patterns, patterns template, 75

#### Relational models, DBMS, 241

#### Relationships, 265, 268
- and associations, objects, 26–27
- meta-models and, 118

#### Relevant categories, selecting classes from, 155–156

#### Request translation, 264–265
- access layer, 84

#### Requirement determination, 125

#### Resolution of forces, patterns template, 75

#### Resulting context, patterns template, 75

#### Results translation, 265
- access layer, 84

#### Reusability, 5, 53, 181, 186
- of tests, software quality assurance, 331

#### Reverse architecting, 76

#### Reverse engineering, 255

#### Reviews, nonanonymously, patterns, 77

#### Ribbons, 300

#### Roles, 46

#### Rumbaugh technique, 63–65
- object model, 63
- OMT dynamic model, 63–64
- OMT functional model, 64

#### Run-time errors, 327

### S

#### SavingsAccountUI, 311, 314–315

#### Scenario-based testing, 328

#### Schema, 239

#### Scope, APA:1, APA:3
- applicable documents, APA:1, APA:3

#### Software quality assurance—Cont.
- test plan, 333–335
- alpha testing, 334
- beta testing, 334
- configuration control system, 335
- objectives of test, 333
- test analysis, 334
- test case development, 334
- tests for, 326–328
- debugging, 327
- error-based testing, 327
- language errors, 327
- logic errors, 327
- run-time errors, 327
- scenario-based testing, 328
- syntax errors, 327
- usage-based testing, 328
- ViaNet case study, 337–338

#### Solution, patterns template, 75

#### Specializations, 186

#### Spin boxes, APB:8

#### Spring-loaded modes, 292

#### Stamp coupling, 205

#### Standardization (corollary 5), 204, 208

#### State and properties, of objects, 17–18

#### Statechart diagrams, 106–109

#### State diagram, 106

#### Statement testing coverage, 329

#### Static binding, 34

#### Static models, 90–91

#### Static structure, 94
- patterns template, 75

#### Status bars, 300

#### Stepwise refinement, 28

#### Stereotypes, 117

#### Stop message, 18–19, 23, 24

#### Stored procedure, 257

#### Strings, APB:4

#### Strong mapping (corollary 4), 204, 207–208

#### Structured approach, payroll case study, 28–30

#### Structured query language (SQL), 242, 243

#### Subclasses, 21, APA:2, APA:3, APA:13, APA:16, APA:18

#### Subsystem, 231

#### Sun Microsystems, 255–256

#### Superclasses, 21, 181, APA:2, APA:3, APA:13, APA:16, APA:18
Unified approach (UA)—Cont.
layered approach to software development; see Layered approach
modeling based on UML, 80–81
object-oriented analysis (OOA), 79–80
use case identification, 128
object-oriented design, 80
proposed repository, 81–82
Unified Modeling Language
Notation Guide Version 1.1, 93
Unified modeling language (UML),
6, 8–9, 62, 80–81, 89–122
activity diagram, 109–111
analysis object model, 90
attribute presentation, 222–223
clarity and, 91
class diagrams; see Class diagrams
diagrams, 93, APA:2, APA:8–18
access classes, APA:2,
APA:11–12
access layer classes, APA:2–3,
APA:14–17
a-part-of structures and aggregation, APA:2,
APA:16
association, APA:2, APA:16
methods, APA:3, APA:16–17
subclasses, APA:2, APA:16
superclasses, APA:2,
APA:16
a-part-of structures or aggregation, APA:2,
APA:3, APA:13, APA:16,
APA:18
association, APA:2, APA:13–14
business classes, APA:2,
APA:11–12
business layer classes, APA:2,
APA:13–14
class diagrams; see Class diagrams
development, ViaNet case study, 184–185
methods, APA:2, APA:14
object responsibility: methods and messages, 192
packages, APA:18
subclasses, APA:2, APA:13
superclasses, APA:2, APA:13
user interface (view layer)
classes, APA:3, APA:17–18
Unified modeling language—Cont.
diagrams—Cont.
user interface (view layer)
classes—Cont.
a-part-of structures
(aggregation), APA:3,
APA:18
association, APA:3, APA:18
interface behavior: events
and actions, APA:2,
APA:18
screen, APA:3, APA:18
subclasses, APA:3, APA:18
superclasses, APA:3, APA:18
view classes, APA:2, APA:
11–12
domain object model, 90
dynamic modeling; see Dynamic modeling (behavior diagrams)
extensibility, 116–117
familiarity and, 91
implementation diagrams,
111–114
component diagrams, 112
deployment diagrams, 112–114
interaction diagrams, 104–106,
APA:2, APA:8
introduction to, 92–93
maintenance and, 91–92
meta-model, 117–119
model constraints and comments,
116–117
modeling, reasons for, 91–92
model management: packages
and model organization,
114–116
note, 117
OCL, classes, designing, 218–219
operation presentation, class design, 227
packages, 114–116
simplification and, 92
statechart diagrams, 106–109
static models, 90–91
stereotype, 117
test model, 90
use-case diagram, 101–103
use case model, APA:1–2,
APA:4–8
use-case model, 90
Unique identifier (UID), 33
Unnecessary associations, 180
Upper bound, 97
Usability testing, 343–345, APA:24
Usage-based testing, 328
Use-case approach, 46, 164–169
development driven by, 70
diagrams, 101–103
identification of use cases
unified approach (UA) and, 128
ViaNet case study, 141–146
Jacobson methodologies, 68–69
model for; see Use-case model
object responsibility: methods
and messages, 192
OOA; see Object-oriented analysis (OOA)
systems development, 44–53
analysis prototype, 48
domain prototype, 48–49
horizontal prototype, 48
implementation, 49–52
component-based development (CBD), 50
computer-aided software engineering (CASE),
49–50
rapid application development (RAD),
51–52
software components, 50–51
incremental testing, 53
object-oriented analysis
(OOA), 45–47
object-oriented design
(OOD), 47
prototyping, 47–49
vertical prototype, 48
Use-case model, 46, 70, 90,
129–138
actors, 131, 134–136
detail and, 136–137
extends association, 133–134
guidelines for finding use cases,
136
measurable value, 131
naming a use case, 137
packages
dividing use cases into, 137
ViaNet case study, 146
transaction, 132
use case defined, 131
uses association, 133–134
User-centered interface,
287–288
User control of software (UI design rule 3), 290–292
Use relationship, 26
User interface, 248, APA:3, APA:17–18; see also View layer; interface objects
software development and, 84
User satisfaction; see also System usability and user satisfaction testing, APA:25
Uses, use-case diagrams, 103
Uses association, 133–134

V
Validation, 43, 341
Verification, 43, 341
Vertical prototype, 48
ViaNet Bank ATM case study
access layer design, 269–275
attributes
defining, 190–191
refining, 223–225
account class, 223–224
ATM machine class, 224
BankClient class, 223
checking account class, 224
savings account class, 224–225
transaction class, 224
class identification, 156
common class patterns approach, 163–164
CRC, 171–172
document template; see Document template
methods definition, 192–194
account class operations, 192–193
BankClient class operations, 193
checking account class operations, 193–194
objects, class design methods and protocols, 227–230
account class create method, 229–230
account class deposit method, 228–229
account class withdraw method, 229
ATM machine class operations, 230
ViaNet Bank ATM case study—Cont.
objects, class design methods and protocols—Cont.
BankClient class verify password method, 228
checking account class withdraw method, 230
relationship analysis; see Object relationships, attributes, and methods
sequence/collaboration modeling, 165–169
software quality assurance, 337–338
system usability testing, 350–352
use case identification; see Object-oriented analysis (OOA)
user interface design; see View layer; interface objects
View classes, APA:2, APA:11–12
View layer; interface objects, 281–321
application windows (main windows), 292–293
colors, guidelines for using, 300–302
command button layout, 298–299
command menus, 300
dialog boxes, 292, 296–297
edit menu, 299
error messages, 296–297
file menu, 299
fonts, guidelines for using, 302
forms and data entry windows, 292, 293–296
help window, 300
macro level design process, 285–287
micro level UI design; see Micro level UI design
object-oriented user interface (OOUI), 282–283
prototyping the user interface, 302–304
purpose of, 292–302
status bars, 300
toolbars, 300
View layer; interface objects—Cont.
user interface
design of, as creative process, 281–284
prototyping of, 302–304
ViaNet Bank ATM case study, 304–316
AccountTransactionUI interface object, 309, 315–316
BankClient AccessUI, 309, 313
CheckingAccountUI interface object, 311
interface behavior, defining, 311–316
macro process, 305–308
MainUI object interface, 309, 313–314
micro process, 308–309
SavingsAccountUI, 311, 314–315
view layer classes, 284–285
view menu, 300
window menu, 300
View menu, 300
Visibility of classes, 219–221
encapsulation leakage, 220
private protocol, 220, 221
protected protocol, 220, 221, 223
protocol, 219
public protocol, 220, 221
Vocabulary, documentation and, 139

W
Waterfall approach, 41, 41n
White box testing, 329
Windows
how it works, APB:1–3
menu, 300
messages, APB:1
Wizard of Oz, 344
Works for, 185
Writers’ workshops, patterns, 77

Y
Year 2000 (Y2K) problem, 296n