CHAPTER

Making Programs Fail

3

Before a program can be debugged, we must set it up such that it can be *tested*—that is, executed with the intent to make it fail. In this chapter, we review basic testing techniques, with a special focus on automation and isolation.

3.1 TESTING FOR DEBUGGING

User reports are not the only way of learning about problems. Typically, most problems (and in the best of all worlds, *all* problems) are found by *testing* at the developer's site before they ever could be experienced by a user. Testing is the process of executing a program with the intent of producing some problem. Once such a problem has been found by testing, the process of tracing down the defect and fixing it is the same as if the problem had been reported by a user (except that problems found by testing, or any other means of quality assurance, are less embarrassing, and generally much cheaper to fix). First comes the problem (from a test or a user), then comes the debugging.

This classical view of testing is called *testing for validation*. It is geared toward uncovering *yet unknown* problems. A great deal of research revolves around the question of how to test programs such that the tests uncover as many problems as possible. We summarize the basics in Section 3.8. In the context of debugging, though, testing also has its place. However, it is geared toward uncovering a *known* problem. Such *testing for debugging* is required at many stages of the debugging process, and thus throughout this book:

- One must *create a test* to *reproduce the problem* (see Chapter 4).
- One must rerun the test multiple times to *simplify the problem* (see Chapter 5).
- One must rerun the test to *observe the run* (see Chapter 8).
- One must rerun the test to *verify whether the fix has been successful* (see Section 15.4 in Chapter 15).
- One must rerun the test before each new release such that the problem (or a similar one) *will not occur in the future*. This is also called *regression testing* (see Chapter 16).

As testing occurs so frequently while debugging, it is a good thing to *automate* it as much as possible. In general, by using automation, more thorough tests can be achieved with less effort. Automation:

- Allows the reuse of existing tests (for instance, to test a new version of a program).
- Allows one to perform tests that are difficult or impossible to carry out manually (such as massive random tests).
- Makes tests repeatable.
- Increases confidence in the software.

All of these benefits apply to validation as well as to debugging, such as the previously listed.

Automation not only streamlines the "classic" testing and debugging tasks, but enables additional *automated debugging techniques*, such as those discussed in this book:

- Automated tests enable *automated simplification of test cases* (see Chapter 5).
- One can use automated tests to *isolate failure causes automatically*, including:
  - Failure-inducing input (see Section 13.5 in Chapter 13)
  - Failure-inducing code changes (see Section 13.7 in Chapter 13)
  - Failure-inducing thread schedules (see Section 13.6 in Chapter 13)
  - Failure-inducing program states (see Section 14.4 in Chapter 14)

In this chapter, we will thus focus on how to set up automated tests that support our (automated and nonautomated) debugging tasks. We examine the question:

HOW CAN WE LEVERAGE TESTS TO SUPPORT DEBUGGING?

3.2 CONTROLLING THE PROGRAM

Consider a real-world example, related to the MOZILLA Web browser—or more specifically, its HTML layout engine named *Gecko*. In July 1999, two years before the final completion of MOZILLA 1.0, BUGZILLA (the MOZILLA problem database) listed more than 370 open problem reports—problem reports that were not even reproduced. At the same time, test automation was in bad shape. To test MOZILLA , developers essentially had to visit a number of critical Web pages, such as *www.cnn.com/*, and (visually) check whether the layout and functionality was okay.

EXAMPLE 3.1: MOZILLA problem report 24735

```
Ok the following operations cause mozilla to crash consistently
on my machine

-> Start mozilla
-> Go to bugzilla.mozilla.org
-> Select search for bug
-> Print to file setting the bottom and right margins to .50
   (I use the file /var/tmp/netscape.ps)
-> Once it's done printing do the exact same thing again on
   the same file (/var/tmp/netscape.ps)
-> This causes the browser to crash with a segfault
```

Example 3.1 shows the problem report we want to turn into an automated test case. Under certain conditions, MOZILLA crashes when printing a page. How do we automate this sequence of actions? In general, an automated test must simulate the *environment* of the program—that is, the test must provide the program's input and assess the program's output. Simulating an environment can be very tricky, though. If the environment involves users who interact with the program, the automated test must simulate actual users (including all of their capabilities).

Figure 3.1 shows the steps our user simulation must conduct, which are:

1. Launch MOZILLA.
2. Open the Open Web Location dialog.
3. Load *bugzilla.mozilla.org*.
4. Open the Print dialog.
5. Enter appropriate print settings.
6. Invoke the actual printing.

However, our user simulation must also *synchronize* its actions with the application. For instance, the simulation can "click" in the dialog only after it has popped up. The same applies for the second printing, which can only start after the first printing is done. Thus, the user simulation must not only provide input but interpret the output.

Such efforts can be avoided by identifying *alternate interfaces*, where control and assessment are easier to automate. Figure 3.2 shows a typical decomposition of a program into three layers:

- The *presentation layer* handles the interaction with the user (or whatever constitutes the environment of the program).
- The *functionality layer* encapsulates the actual functionality of the program, independent of a specific presentation.
- The *unit layer* splits the functionality across multiple units, cooperating to produce a greater whole.

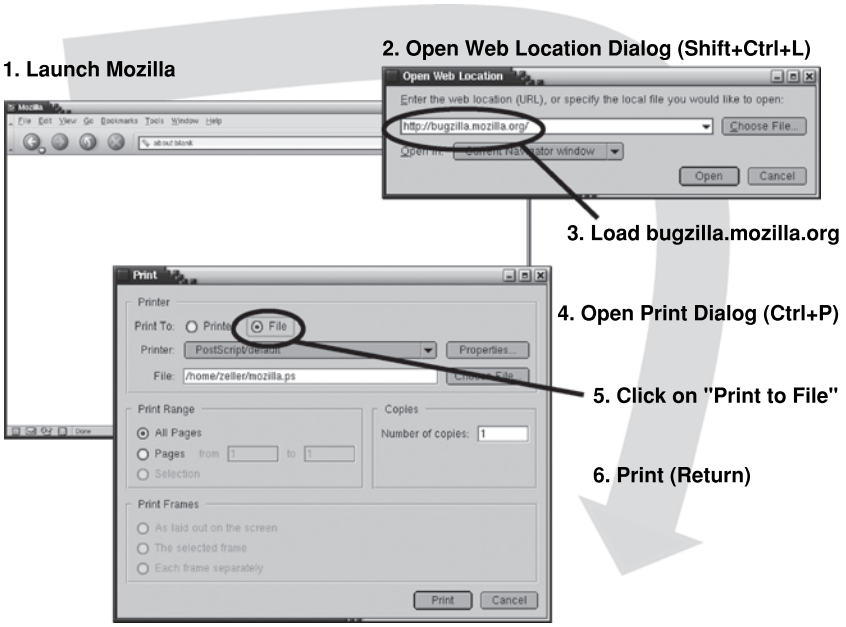

FIGURE 3.1

Making MOZILLA print (and crash). This takes just six easy steps.

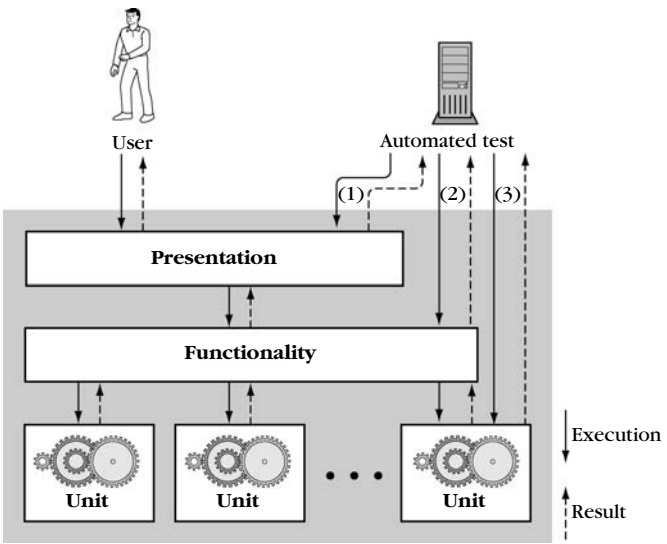

FIGURE 3.2

Testing layers. A program can be tested (1) at the presentation layer, (2) at the functionality layer, or (3) at the unit layer.

Whereas the user (and the environment) interact only with the presentation layer, an automated test can use all three layers for automating execution and for retrieving and assessing results. Each layer requires individual techniques, though, and brings its own benefits and drawbacks for testing and debugging. In the next three sections, we discuss testing at the individual layers and check for the following features:

- *Ease of execution:* How easy is it to get control over program execution?
- *Ease of interaction:* How easy is it to interact with the program?
- *Ease of result assessment:* How can we check results against expectations?
- *Lifetime of test case:* How robust is my test when it comes to program changes?

3.3 TESTING AT THE PRESENTATION LAYER

Let's start with the presentation layer, where the program interacts with its environment. How does one test at the presentation layer? Basically, one simulates the *input* and monitors the *output*. Depending on the nature of the input and output, this can be done at multiple *abstraction levels*. For a networking device, for instance, we can capture input and output at the *data link layer* (monitoring and sending individual bits between two ends of a physical connection), or at the *transport layer* (monitoring and sending data packets between two machines). The higher the abstraction level, the more details are abstracted away, which makes it easier to simulate interaction. On the other hand, one risks abstracting away the very details that cause a failure.

As a more detailed (and frequent) example of abstraction levels, let's take a look at *user interaction*. User interaction can be simulated at two abstraction levels: at a *low level* (expressing interaction as a sequence of mouse and keyboard events) or at a *higher level*, denoting interaction using graphical user controls of the application.

3.3.1 Low-Level Interaction

At the lowest abstraction level, user input becomes a stream of mouse and keyboard events. Such event streams can be *captured* (i.e., recorded from the input devices) and *replayed*, substituting the stream from actual input devices by the previously recorded stream.

As an example, Example 3.2 shows a script recorded by the open-source tool ANDROID (not to be confused with Google's *Android* operating system) to reproduce the MOZILLA interaction shown in Figure 3.1. To make it more user readable, the script has been simplified to the relevant events.

Each of these `send_xevents` command simulates a user action. The command

```
send_xevents @550,170 click 1
```

EXAMPLE 3.2: ANDROID script to make MOZILLA print. This script simulates user interaction at a low level by means of keyboard and mouse interaction.

```
# 1. Launch mozilla and wait for 2 seconds
exec mozilla &
send_xevents wait 2000

# 2. Open URL dialog (Shift+Control+L)
send_xevents keydn Control_L
send_xevents keydn Shift_L
send_xevents key L
send_xevents keyup Shift_L
send_xevents keyup Control_L
send_xevents wait 500

# 3. Load bugzilla.mozilla.org
#    and wait for 5 seconds
send_xevents @400,100
send_xevents type {http://bugzilla.mozilla.org}
send_xevents key Return
send_xevents wait 5000

# 4. Open Print Dialog (Ctrl+P)
send_xevents @400,100
send_xevents keydn Control_L
send_xevents key P
send_xevents keyup Control_L
send_xevents wait 500

# 5. Click on "Print to File"
send_xevents @550,170 click 1

# 6. Print (Return)
send_xevents key Return
send_xevents wait 5000
```

tells ANDROID to move the mouse pointer to position (550,170), and then to simulate a click of mouse button 1 (the left mouse button). Likewise, the command `key` simulates the press of a key, and `type` is shorthand for typing several keys in a row. The commands `keydn` and `keyup` are handy for simulating modifiers such as Shift, Alt, or Ctrl that need to be held down while other keys are pressed.

As nobody wants to read or maintain tests that deal with absolute screen coordinates, such event scripts are largely *write-only*. Furthermore, any recorded information is *fragile*: The slightest change in the user's display or the program's interface makes the recorded scripts obsolete.

To illustrate the fragility, just try to invoke the script *twice* in a row: The second time the script executes, the file to be printed to already exists, and thus MOZILLA wants special confirmation before overwriting the file. This extra dialog, though, is

not handled in our script and thus will fail miserably. Other changes that quickly make the script obsolete include a different placement of the MOZILLA main window or its dialogs (all coordinates in the script are absolute) and changes in font size, screen size, layout, user language, or even interaction speed.

If we record and replay nonuser interaction at a low level, such as data flow on a network, any changes to the program or the protocol will also make recorded scripts quickly obsolete. Nonetheless, such recorded information can be very useful for automating user interaction again and again—as long as it is used for one single debugging session in one specific environment.

3.3.2 System-Level Interaction

One way of overcoming the problem of fragility (Section 3.3.1) is to control not only the single application but the entire machine. For this purpose, one typically uses a *virtual machine* system that simulates an entire machine as software. The virtual machine FAUMachine, for instance, allows us to simulate many types of input and can even inject faults such as simulated hardware defects. Example 3.3 shows a simple script.

Use of virtual machines for testing and debugging typically requires that a number of well-defined virtual machines be available. Therefore, virtual machines are nice to have if one desires or requires complete control at the system level. Although a large set of virtual machines requires careful administration, it is still easier to administer and configure virtual rather than real machines.

3.3.3 Higher-Level Interaction

A more comfortable way of making user interaction scripts more robust against changes and thus more persistent is to choose a *higher abstraction level*—that is, controlling the application not by means of coordinates but by actual graphical user controls. As an example, consider Example 3.4. It shows a script in the APPLESCRIPT language that makes MOZILLA on Mac OS load and print the page *bugzilla.mozilla.org*. APPLESCRIPT is designed to be readable by end users. The ¬ character lets you split one line of script onto two.

EXAMPLE 3.3: A script for automating execution of a virtual FAUmachine. This script interacts at the system level, simulating the hardware of a real machine

```
# Power on the machine and wait for 5s
power <= true; wait for 5000;

# Click mouse button 1
m_b1 <= true; wait for 300; m_b1 <= false;

# Click the CDROM change button
cdctrl'shortcut_out_add("/cdrom%change/…");
```

EXAMPLE 3.4: APPLESCRIPT makes MOZILLA print. This script excerpt interacts with MOZILLA at a higher level. It refers to named GUI elements to simulate actions.

```
-- 1. Activate mozilla
tell application "mozilla" to activate

-- 2. Open URL dialog via menu
tell application "System Events"
  tell process "mozilla"
    tell menu bar 1
      tell menu bar item "File"
        click menu item "Open Web Location"
      end tell
    end tell
  end tell
end tell

-- 3. Load bugzilla.mozilla.org
--    and wait for 5 seconds
tell window "Open Web Location"
  tell sheet 1
    set value of text field 1 to ¬
      "http://bugzilla.mozilla.org/"
  end tell
  click button 1
end tell
delay 5
```

⋮

The main difference with the ANDROID script shown in Example 3.2 is that APPLESCRIPT no longer references user controls by position but by *names* such as `Open Web Location` and *relative numbers* such as `menu bar 1`. This makes the script much more robust against size or position changes (only the labels and the relative ordering of the user interface controls must remain constant).

Again, such scripts can also be recorded from user interactions. Several capture/replay tools are available that work at the level of named user controls. However, even if we raise the abstraction level to user controls scripts remain fragile: Any single renaming or rearrangement of controls causes all scripts to become obsolete.

3.3.4 Assessing Test Results

Whether we are controlling the application using event streams or user controls, one major problem remains: Our simulation must still *examine the program's output*.

- Examining the output is necessary for *synchronization*, as the simulated user may have to wait until a specific action completes. In our MOZILLA script, we circumvented this problem by introducing appropriate delays.

- Examining the program's output is necessary for *result assessment*. Eventually, our test must determine whether the result matches the expectations or not. In our MOZILLA example, this was particularly easy. The crash of a program is relatively easy to detect, but if we had to verify MOZILLA's output on the screen we would have a difficult time processing and assessing this output.

To sum up, the advantage of testing at the presentation layer is that it is always feasible. We can always simulate and automate a user's behavior. However, this is already the only advantage. In general, one should use the presentation layer for testing only:

- If the problem occurs in the presentation
- If the presentation layer is easily used by computers
- If there is no other choice (for instance, because there is no clear separation between presentation and functionality, or because the lower layers are inaccessible for testing)

The rule of thumb is: The friendlier an interface is to humans, the less friendly it is to computers. Therefore, we should have a look at alternative interfaces that are better suited to automation.

3.4 TESTING AT THE FUNCTIONALITY LAYER

Rather than simulate user interaction, it is much preferable to have the program provide an interface that is *designed for automation*—or, more generally, designed for interaction with technical systems. Such an interface may be designed for interaction with programming languages (for instance, the programming language the application itself is written in). However, some programs provide interfacing with *scripting language*, allowing even end users and nonprogrammers to automate execution in a simple way.

Example 3.5 shows an APPLESCRIPT program that uses the scripting capabilities of the Safari Web browser to load a given Web page and to print it, mimicking our MOZILLA example. This script uses commands such as

```
set the URL of the front document
```

that work regardless of what the user interface looks like, and thus make the script unaffected by any changes of the user interface. Note, though, that not every Safari feature is scriptable. To print a page (step 2 in Example 3.5), we still have to fall back to the presentation layer.

Support for automation at the functionality layer greatly differs by operating environment. In Mac OS, APPLESCRIPT is available for several applications. In Windows, this role is filled by *Visual Basic*. Example 3.6 shows a VBSCRIPT program that loads a file into Internet Explorer (IE; note how this program waits until the page is actually loaded). Under Linux and UNIX, there is no single standard for scripting—no scripting support for MOZILLA, for instance.

EXAMPLE 3.5: Loading a site in Safari using APPLESCRIPT. This script uses Safari's built-in functionality layer to open Web pages—except for printing, where one has to resort to simulating user interaction

```
# 1. Load document
tell application "Safari"
  activate
  if not (exists document 1)
    make new document at the beginning of documents
  end if
  set the URL of the front document ¬
    to "http://bugzilla.mozilla.org/"
  delay 5
end tell

# 2. Print it
# No script support for printing, so we go via the GUI
tell application "System Events"
  tell process "safari"
    keystroke "p" using command down
  end tell
end tell
```

EXAMPLE 3.6: Loading a site in Internet Explorer using VBSCRIPT. The script uses IE's functionality layer to open pages

```
' Load document
Set IE = CreateObject("InternetExplorer.Application")
IE.navigate "http://bugzilla.mozilla.org/"
IE.visible=1

' Wait until the page is loaded
While IE.Busy
    WScript.Sleep 100
Wend
```

Nonetheless, the advent of Web components has encouraged further separation of functionality and presentation, thus making automation far easier for future applications. Every major scripting language (such as VBSCRIPT, PERL, PYTHON, and, APPLESCRIPT) can use Web component interfaces such as SOAP to interact with local and distributed components and services. Essentially, arbitrary Web components can be accessed using arbitrary scripting languages.

You may be tempted to define your own *home-grown* scripting language that is built into the application. In general, however, this is not worth the investment. Sooner or later you will require variables, control structures, and modularization—and it is difficult to add these features one at a time. It is far easier to incorporate

an existing interpreter for a powerful scripting language such as PYTHON, PERL, or TCL and extend it with application-specific commands. Even more easily, you can turn your application into a .NET component, a JAVA bean, or a CORBA component. All of this makes the functionality available for arbitrary automation purposes and is thus great for automated testing. (Be aware, though, that automation interfaces can be exploited by malicious users. For instance, automation features in Microsoft Office have frequently been used to send document and email viruses automatically.)

Overall, the big advantage of testing at the functionality layers is that the *results can be easily accessed and evaluated*—something that is difficult to do at a presentation layer for humans. For Web components, results typically come in XML format, which is easy to parse and process for all scripting languages. Thus, unless one wants to test individual parts of the program, testing (and debugging) at the functionality level is the way to go.

Unfortunately, all of this requires a clear separation between presentation and functionality. Especially older programs may come as monolithic entities without presentation or functionality layers. In this case, you have three choices:

- You can go through the presentation layer, as discussed in Section 3.3, and suffer all of the problems associated with assessing test results.
- You can do a major redesign to separate presentation and functionality—or at least to reduce dependences between them. We will come back to this idea when discussing designing for debugging (Section 3.7).
- You can decompose the program and access the individual units directly. (This is discussed in the next section.)

3.5 TESTING AT THE UNIT LAYER

Any nontrivial program can be decomposed into a number of individual *units*—that is, subprograms, functions, libraries, modules, abstract data types, objects, classes, packages, components, beans, or whatever decomposition the design and the language provide. These units communicate via *interfaces*—just like the program communicates with its environment.

The idea now is not to automate the execution of the entire program but only *the execution of a specific unit.* This has the advantage that automating the unit in isolation is frequently easier than automating the entire program. The disadvantage, of course, is that you can only automate the behavior of the given unit and thus must count on the unit producing the problem in isolation.

Units are typically not accessible to end users, and thus not necessarily accessible for scripting, as discussed in Section 3.4. However, they are accessible to programmers, using the same means as the enclosing program to access their services—typically, simple invocation of functions and methods in the language of the program.

Whereas units are among the eldest concepts of programming, the concept of automated testing at the unit level has seen a burst of interest only in the last few years. This is due to the big interest in extreme programming, which mandates automated tests as early and often as possible (and notably the creation of a unit test case *before* implementation), and to the fact that massive automated testing has become much more affordable than, say, 20 years ago.

All of these tools provide a *testing framework* that collects a number of individual *unit tests*—a test that covers a single unit. Unit tests are supposed to run automatically—that is, without any user interaction. On request, the testing framework runs some or all unit tests and displays a summary of the executed unit tests and their respective outcomes. When a single unit test executes, a testing framework does three things:

1. *It sets up an environment for embedding the unit.* Frequently, a unit will require services of other units or the operating environment. This part sets up the stage.
2. *It tests the unit.* Each possible behavior of the unit is covered by a test case, which first performs the operation(s) and then verifies whether the outcome is as expected.
3. *It tears down the environment again.* This means it brings everything back in the state encountered initially.

Consider an example of how to use unit tests. Assume that as part of a Web browser you manage a JAVA class for uniform resource locators (URLS) such as the following.

```
http://www.askigor.org/status.php?id=sample#top
```

A URL class has a constructor that takes a URL as a string. Among others, it provides methods for retrieving the protocol (e.g., `http`), the host (e.g., `www.askigor.org`), the path (e.g., `/status.php`), and the query (e.g., `id=sample`).

Suppose you want to test these methods. Because you are working with a JAVA class, one of the first choices for unit testing is the JUNIT testing framework. JUNIT provides all we want from a testing framework. It allows us to organize and conduct automated tests in a simple yet effective fashion. (In fact, JUNIT has been so successful that its approach has been adopted for more than 100 languages, including CPPUNIT for C++, VBUNIT for VBSCRIPT, PYUNIT for PYTHON, and so on.)

To test the URL class with JUNIT, you create a test case `URLTest` that is a subclass of `TestCase`. The source code `URLTest.java` is shown in Example 3.7. In this template, the `setUp()` method is responsible for setting up the environment for embedding the unit. The `tearDown()` method destroys the environment again. Our environment consists of a rational member variable `askigor_url` containing the URL. This variable can be used in all further tests.

EXAMPLE 3.7: URLTest.java—a unit test for URLS

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class URLTest extends TestCase {

    private URL askigor_url;

    // Create new test
    public URLTest(String name) {
        super(name);
    }

    // Assign a name to this test case
    public String toString() {
        return getName();
    }

    // Setup environment
    // will be called before any testXXX() method
    protected void setUp() {
        askigor_url = new URL("http://www.askigor.org/" +
                              "status.php?id=sample");
    }

    // Release environment
    protected void tearDown() {
        askigor_url = null;
    }
```

We can add the individual tests to this class. In JUNIT, each test comes in a separate method. We shall add four methods that test for equality and nonequality, respectively, as shown in Example 3.8. The `assertEquals()` method makes the test fail if the two arguments do not equal each other.

We next need a suite that runs all tests, as shown in Example 3.9. By default, any method of the test class with a name that begins with the word `test` will be run as a test. For the last step, we have to give the class a main method that invokes a GUI for testing. This piece of code is shown in Example 3.10. This concludes the `URLTest` class.

The main method we have added to the test case allows us to execute it as a stand-alone application. If we do so, we obtain the graphical user interface shown in Figure 3.3. Clicking on Run runs all tests at once. The bar below shows the status. If the bar is green (as in the left window), all tests have been run successfully. If the bar is red (as in the right window), some tests have failed.

The important thing about unit tests is that they run *automatically*—that is, we can assess the unit state with a click of a single button. Recent studies by Saff and

EXAMPLE 3.8: Actual tests in `URLTest.java`

```
// Test for protocol ("http", "ftp", etc.)
public void testProtocol() {
    assertEquals(askigor_url.getProtocol(), "http");
}

// Test for host
public void testHost() {
    int noPort = -1;
    assertEquals(askigor_url.getHost(),
                 "www.askigor.org");
    assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
    assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
    assertEquals(askigor_url.getQuery(), "id=sample");
}
```

EXAMPLE 3.9: Setting up a test suite in `URLTest.java`

```
// Set up a suite of tests
public static TestSuite suite() {
    TestSuite suite =
        new TestSuite(URLTest.class);
    return suite;
}
```

EXAMPLE 3.10: A main method for `URLTest.java`

```
    // Main method: Invokes GUI
    public static void main(String args[]) {
        String[] testCaseName =
            { URLTest.class.getName() };

        // Run using a textual user interface
        // junit.textui.TestRunner.main(testCaseName);

        // Run using a graphical user interface
        junit.swingui.TestRunner.main(testCaseName);
    }
}
```

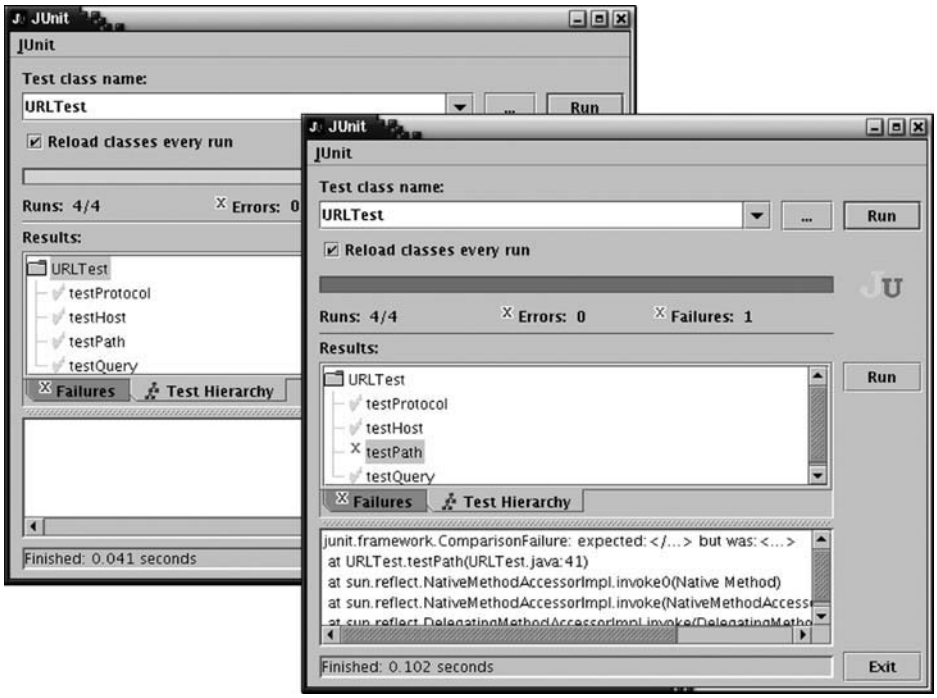

FIGURE 3.3

The JUNIT graphical user interface. The left dialog shows a passing test, the right dialog a failing test—with a failure description in the bottom test field.

Ernst (2004b) show that users write better code faster if the test runs automatically *each time they save the program* (i.e., not even a button click is needed). This idea of *continuous testing* suggests that you simply cannot test early and often enough.

3.6 ISOLATING UNITS

Automated unit testing of low-level classes such as `URL` is particularly easy, because such classes do not *depend* on anything. That is, we do not have to import and set up an entire environment just to make the URL class run. In principle, we could also unit test entire applications such as Mozilla—in a manner similar to testing at the functionality layer (Section 3.4) but using an API rather than a scripting language.

However, all of this automation again requires that the unit in question clearly separate functionality and presentation, and make its results available for automatic assessment. This is true for many programs, which thus make it possible for functionality to be examined (and tested and debugged) in isolation.

However, there are programs in which the functionality depends on the presentation, such that it is impossible to separate them. Example 3.11 shows an example. The function `print_to_file()` prints the current Web page to a file. To avoid overwriting an existing file, it asks the user for confirmation if the file already exists. From the user's perspective, such protection against data loss is a strict necessity. From the tester's perspective, though, this confirmation makes the functionality depend on the presentation. This introduces a *circular dependence*, as shown in Figure 3.4:

- The presentation invokes `print_to_file()`, thus depending on the functionality.
- The functionality invokes `confirm_loss()`, thus depending on the presentation.

EXAMPLE 3.11: Functionality depending on presentation

```
// Print current Web page to FILENAME.
void print_to_file(string filename)
{
    if (path_exists(filename))
    {
        // FILENAME exists;
        // ask user to confirm overwrite
        bool confirmed = confirm_loss(filename);
        if (!confirmed)
            return;
    }

    // Proceed printing to FILENAME
    ...
}
```

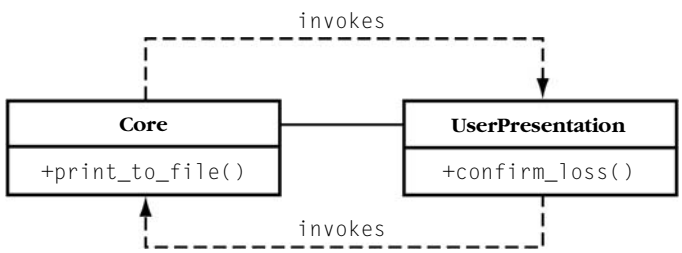

FIGURE 3.4

A circular dependence. The `Core` and `UserPresentation` classes depend on each other and thus cannot be tested (or debugged) separately.

EXAMPLE 3.12: Functionality with parameterized presentation

```
// Print current Web page to FILENAME.
void print_to_file(string filename,
                   Presentation *presentation)
{
    if (path_exists(filename))
    {
        // FILENAME exists; confirm overwrite
        bool confirmed =
            presentation->confirm_loss(filename);
        if (!confirmed)
            return;
    }

    // Proceed printing to FILENAME
    ...
}
```

As a result, presentation and functionality can no longer be separated from each other. This has a bad impact on testing (and debugging), as we can no longer interact at the functionality layer alone. If a testing program invokes `print_to_file()`, setting up `confirm_loss()` to reply automatically will result in a major hassle. The question is thus: How do we break dependences that keep us from isolating units?

In the case of `confirm_loss()`, we could easily hack it such that the function runs in two modes: the "automated" mode disables confirmation, always returning `true`; the "interactive" mode enables confirmation, querying the user. A much more general solution, though, would be to *parameterize* the `print_to_file()` function such that it could work with arbitrary presentations.

This variant of the `print_to_file()` function is shown in Example 3.12. The idea here is to have a `Presentation` class that, among others, again includes the `confirm_loss()` method. However, `Presentation` need not necessarily be a presentation for the user. Instead, as shown in Figure 3.5, `Presentation` is an interface—an abstract superclass that is instantiated only in subclasses. One of these subclasses (e.g., `UserPresentation`) may be geared toward the user and implement all user interaction. Another subclass (e.g., `AutomatedPresentation`) may be geared toward automation, though, and always return `true` whenever `confirm_loss()` is invoked.

What do we get by adopting the inheritance scheme shown in Figure 3.5? We have effectively *broken* the dependence between functionality and presentation—that is, the presentation that is geared toward the user. For testing purposes, we must still provide some functionality we depend on, but this can be encapsulated in a small class such as `AutomatedPresentation`.

Overall, the general principle of breaking a dependence is known as the *dependence inversion principle*, which can be summarized as *depending on*

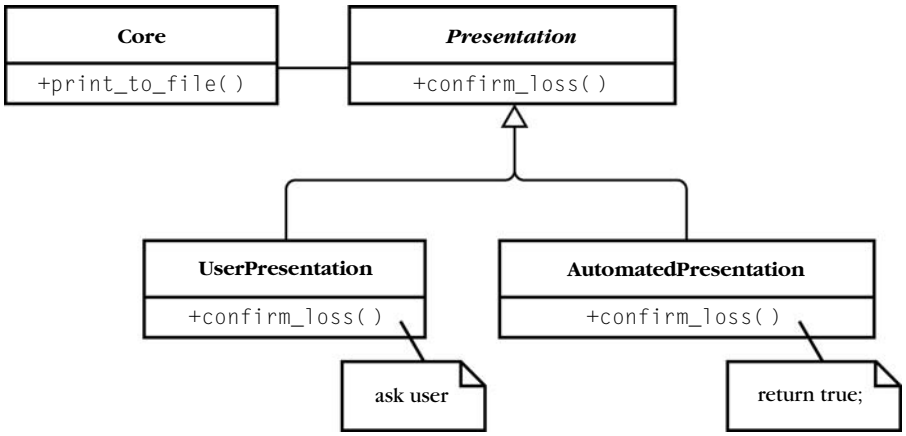

FIGURE 3.5

Depending on abstractions rather than details. `Presentation` is now an abstract superclass, which can be instantiated either as `UserPresentation` (with confirmation) or as `AutomatedPresentation` (without confirmation). The circular dependency between core and presentation is broken.

abstractions rather than details. Whenever you have some component A depending on some component B, and you want to break this dependence, you perform the following:

1. Introduce an abstract superclass B', and make B a subclass of B'.
2. Set up A such that it depends on the abstract B' rather than on the concrete B.
3. Introduce alternate subclasses of B' that can be used with A such that B is no longer required.

By having A depend on the abstract B' rather than on the concrete B, we can set up arbitrary new subclasses of B' without ever having to change A—and we have effectively broken the dependence between A and B.

3.7 DESIGNING FOR DEBUGGING

The principle of reducing dependences by depending on abstractions rather than on details goes a long way. In fact, entire application frameworks can be built this way. Among the most popular examples is the *model-view-controller* architectural pattern, which decouples functionality and presentation at the application level.

To illustrate the model-view-controller pattern, let's imagine we want to build an information system for election day. As illustrated in Figure 3.6, we want to display the election data in a number of graphical formats, including pie and bar charts. We

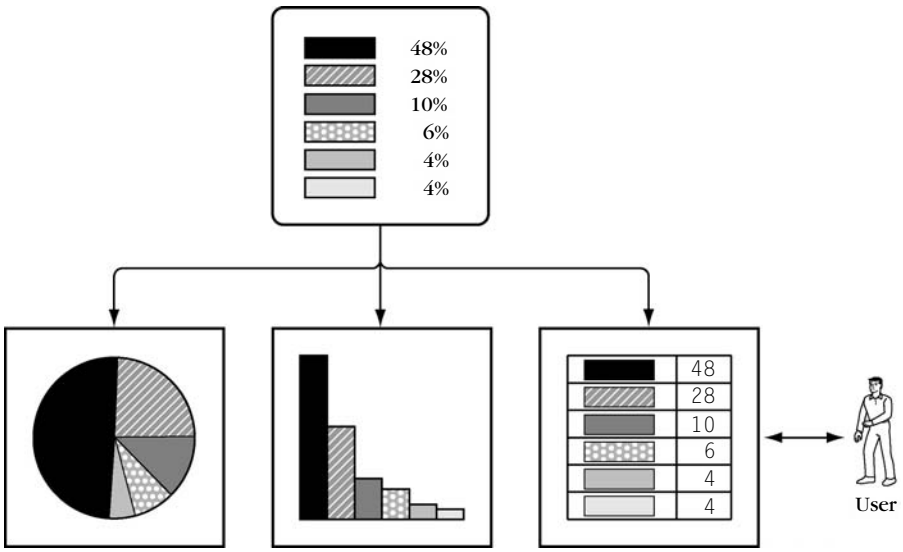

FIGURE 3.6

An information system for election day. The actual data (on top) is displayed in a number of graphical formats, and also manipulated as text.

also want to display the data in the form of a spreadsheet, whereby an operator can manipulate and enter the data.

How would one build such a system? The key idea here is again to separate functionality and presentations. In no way do we want the core functionality being dependent on a specific view. The model-view-controller pattern, as illustrated in Figure 3.7, gives us a general solution to this problem. It splits responsibilities into two parts:

1. A *model* that holds the core data and provides services that operate on this core data.
2. A number of *observers* that register or *attach* to the model and get notified whenever the core data changes.

Observers, again, are divided into two subclasses.

1. A *view* is responsible for displaying the core data in a specific way (such as pie chart views or bar chart views).
2. A *controller* handles input events (typically from the user) and invokes the services of the model.

When a user thus interacts with a controller, he or she will eventually invoke a service that changes the core data. When this happens, all views attached to the model are automatically notified—that is, they can get the data from the model in

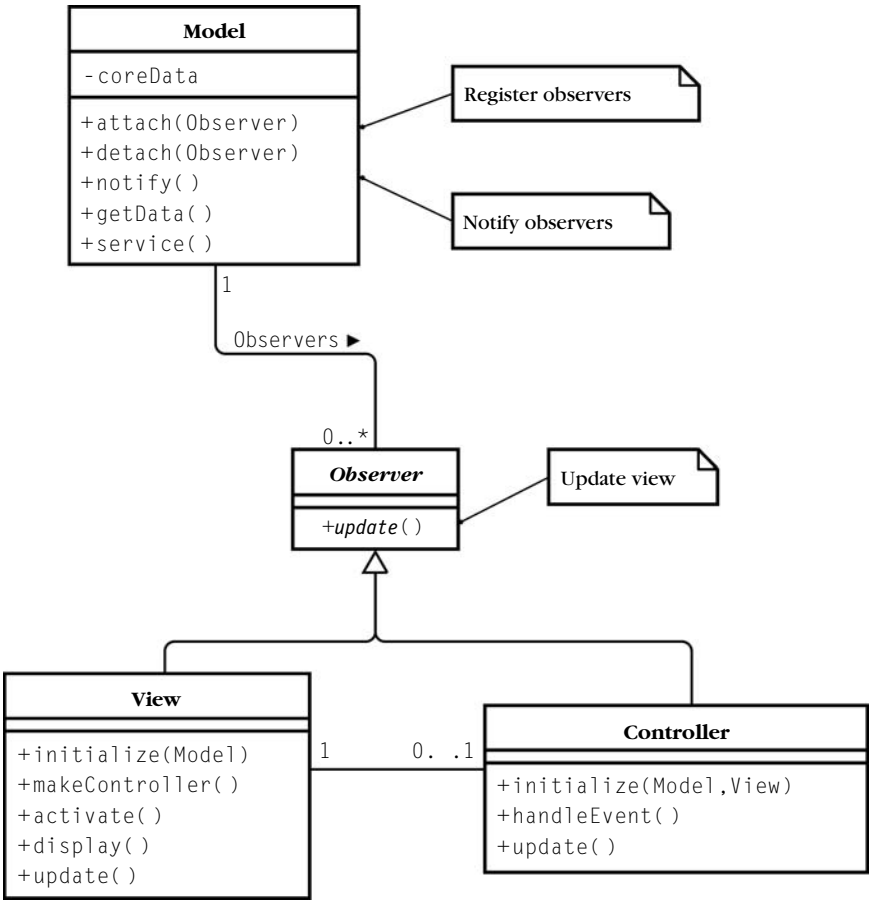

FIGURE 3.7

The model-view-controller pattern. A model has a number of observers, which can be either views or controllers.

order to update their displays. This also includes the view of the user, who thus gets visual feedback.

When it comes to testing and debugging, a model-view-controller architecture has several benefits. For testing, one can build and add new controllers that invoke services of the model—for instance, controllers that automate execution of these services. For debugging, one can register special views that automatically log all changes to the model. Finally, every observer and model can be examined and tested in isolation, thus reducing complexity.

As the model-view-controller pattern shows, it is generally advisable to avoid dependences between presentation and functionality. However, any dependence may eventually cause problems in testing and debugging. Just as we want to examine

systems that are isolated in our controlled environment (rather than embedded in the user's environment), we want to examine units that are isolated in a controlled environment rather than entangled with the entire system. Isolated units are not only easier to test and debug but easier to understand, reuse, and maintain. Reducing dependences is thus a key issue in software design. Fortunately, all software design methods attempt to minimize dependences between units, using the same two principles:

High cohesion. This is the basic principle that tells what to group into a unit. Those parts of a system that operate on common data (and thus depend on these data) should be grouped together—typically, into some unit as supported by the design or programming language. For instance, object-oriented design groups the state (i.e., data) and the functions that work on these data into classes and objects.

Low coupling. This is the principle that reduces dependences. If two units do *not* operate on common data, they should exchange as little information as possible. This principle is also known as *information hiding*, and is the key for understandable, reusable, and extensible systems. The principle of low coupling also prohibits *circular dependences*, as they couple all involved units.

Applying the principles of strong cohesion and low coupling consistently will reduce the number of dependences in any design. Thus, the `confirm_loss()` invocation (Example 3.11) would be counterintuitive as it violates the principle of low coupling by coupling presentation and functionality more than necessary.

Given the time potentially saved on coding, testing, and debugging, any extra hour spent on improving the design is a good investment. A good design will not only make your system more flexible, more robust, and more reusable, but will make it easier to test and to debug. If you want to know more about design, the "Further Reading" section gives a number of useful references.

3.8 PREVENTING UNKNOWN PROBLEMS

So far, this section has been about setting up tests for debugging—that is, how to isolate a unit in a controlled environment. All of this assumes that a problem has already occurred.

Any problem that escapes into the wild (and is experienced by a user) indicates that the product has not been tested (or reviewed or proven) well enough. Consequently, the quality-assurance process must be refined such that the problem in question (and hopefully similar problems) will not occur in the future.

As this is a book about debugging (i.e., the cure of *known* problems), we cannot spend too much space on preventing yet *unknown* problems. This is not to negate that prevention is better than cure. In fact, one might say that, by far, most of computer science is concerned with preventing problems. But when prevention fails,

LIST 3.1: Essential Rules for Testing

Specify. A program cannot be correct on its own—it can only be correct with respect to some *specification* that describes its purpose. Attempt precise, or even formal, specifications that cover the entire behavior, including exceptions. A full specification will be a big help in understanding how the system is supposed to work, and thus help you in writing a correct system.

Test early. This principle states that you must not wait with testing until the entire system is assembled. Instead, run test cases as soon as a unit is implemented, and assemble your system out of carefully tested units.

Test first. Write test cases *before* implementing the unit. This is useful because test cases can serve as *specifications.* Although test cases specify only examples, a sufficient number of test cases can make it difficult to implement something else than the most elegant (and correct) solution.

Test often. At the minimum, run your tests with each release of the system. Better yet, run your tests with every change. The sooner you know that there is a defect, the smaller the set of accumulated changes that might have caused the defect. Automation helps a lot here.

Test enough. Measure the coverage of your tests. How many statements and branches are actually taken? Instrument your code to gather coverage and design your test cases to achieve sufficient coverage. Use random inputs to cover exceptional and extreme situations.

Have others test. Testing for unknown problems is a *destructive* process. By all means, one must try to uncover a weakness in the program. As people in general prefer being constructive to ripping things apart, this is a difficult psychological situation for most. In particular, it makes an author unsuited to test his or her own code. Therefore, always have someone independent test your program, and be open to criticism.

there is need for a cure, and that is what this book is about. Nonetheless, for your reference, Lists 3.1 and 3.2 capture basic rules of testing and quality assurance.

Quality assurance can never reach perfection. Even if all techniques are applied to the extreme, we will still have programs with surprising behavior. However, as a professional developer, you should know about all of these techniques, and be prepared to suggest them whenever it comes to reducing risk. Making mistakes is hard to avoid but not caring to prevent mistakes is unacceptable.

3.9 CONCEPTS

How To

To test for debugging, one must:

- Create a test to reproduce the problem
- Run the test several times during debugging
- Run the test before new releases to prevent regression

LIST 3.2: More Tools and Techniques for Quality Assurance

Have others review. Testing is not the most effective way to catch defects. Reviewing is. No other technique catches so many defects for the same amount of effort. Have someone else review your code and check for possible defects. Think about *pair programming* as a means of increasing the amount of reviews.

Check the code. More and more, computers can detect errors and anomalies in your system. Chapters 7 and 11 give an overview. Running such tools on your code comes at a small cost, but brings greater and greater benefits as computers get faster and faster.

Verify. Several important properties of software systems can today be shown automatically or semiautomatically. If the behavior of your system can be modeled as a finite-state machine, *software model checking* comes in handy to prove correctness. That is how Microsoft validates its device drivers.

Assert. If you cannot fully prove correctness, go the simpler way: Let the computer do the work and have it check its state at runtime (see Chapter 10). Your program may still fail due to a failed assertion, but if all assertions are met the result will be correct with respect to all assertions.

Due to the number of tests needed in debugging, it is thus useful to *automate* as much as possible.

To automate program execution, one can access *three layers*.

- Presentation layer
- Functionality layer
- Unit layer

The layers differ in ease of execution, ease of interaction, ease of result assessment, and robustness against changes.

To test at the presentation layer, the testing environment must stimulate human activities—either *input devices* (low level) or *user controls* (higher level).

To test at the functionality layer, use an interface designed for automation—typically using specific *scripting languages.*

To test at the unit layer, use the *API of a program unit* to control it and to assess its results.

To isolate a unit, break dependences using the *dependence inversion principle*, making the unit depend on abstractions rather than on details.

To design for debugging, reduce the amount of dependences using the principles of *high cohesion* and *low coupling.*

Design patterns such as *model-view-controller* are useful for reducing dependences.

To prevent unknown problems, one can use a variety of techniques, including the following:

- Testing early, testing often, and testing enough
- Reviewing by others and pair programming

- Having the computer check the code for anomalies and common errors
- Proving correctness formally (using computer support)

3.10 TOOLS

JUNIT. JUNIT as well as unit test tools for other languages can be obtained via its Web page at *http://www.junit.org/*.

ANDROID. All scripting languages described in the chapter are also documented online. The ANDROID package, unfortunately, is no longer available for download; an Internet search may help you out.

APPLESCRIPT. APPLESCRIPT documentation and examples are found at *http://www.apple.com/applescript/*. Neuburg (2003) is strongly recommended as a guide to APPLESCRIPT.

VBSCRIPT. VBSCRIPT and other Microsoft tools for scripting can be found via *http://en.wikipedia.org/wiki/VBScript*.

Other Scripting Languages. Other scripting languages suitable for test automation include PYTHON, PERL, TCL, and JAVASCRIPT, all of which are documented by a great deal of information available on the Web.

FAUMachine. The virtual machines discussed in this chapter are also publicly available. The FAUMachine is a virtual machine specifically built for testing purposes. Among others, the FAUMachine allows you to control the entire virtual machine via scripts. FAUMachine can be researched at *http://www.faumachine.org/*.

VMWare. At the time of this writing, VMWare was one of the most popular providers of virtual machines. It can be found at *http://www.vmware.com/*.

Virtual PC. Microsoft also offers *Microsoft Virtual PC* for various operating systems, found at *http://www.microsoft.com/virtualpc/*.

3.11 FURTHER READING

Testing

The book by Myers (1979) has been the classic text on testing for 30 years. It is still up-to-date, and I recommend it as a first read to anyone who is interested in testing. It also includes a chapter on testing for debugging. If you prefer a hands-on approach, try Kaner et al. (1999).

The book by Pezzè and Young (2005) is an in-depth treatment of all things testing and analysis. Psychological issues, in particular the law that developers are unsuited to testing their own code, are addressed in Weinberg (1971).

Automation

Fewster and Graham (1998) and Dustin et al. (2001) focus on *automated* testing, focusing especially on the management view—such as when and how we should automate which parts of our test suite. A more technical view on automated testing can be found on the Web sites devoted to extreme programming and unit testing, in particular *http://www.junit.org/* for JUNIT and *http://www.xprogramming.com/* for extreme programming.

Design

If you do not have it already, *Design Patterns* by Gamma et al. (1994) contains much wisdom regarding how to structure systems. On the architectural level, the *Pattern-Oriented Software Architecture Series* by Buschmann et al. (1996) and Schmidt et al. (2000) contains several useful patterns. The model-view-controller example is taken from this series.

The classic all-in-one textbook on object-oriented software design is the book by Meyer (1997). Other classic design books include those by Booch (1994) and Larman (2002). The *dependence inversion principle* was coined by Martin (1996). The article is available online at *http://www.objectmentor.com/*.

EXERCISES

3.1 In a few words, describe testing for debugging and for validation. Discuss the differences between these purposes.

3.2 Discuss the differences between testing at presentation, functionality, and unit layer. Focus on ease of execution, ease of interaction, ease of result assessment, and robustness against changes.

3.3 Is testing at the presentation layer of a command-line tool the same as functionality testing? Discuss similarities and differences.

3.4 Use your favorite Web browser and try to automate the loading of a Web page, interacting at the presentation or the functionality layer.

3.5 Run the `URLtest.java` JUNIT test (Example 3.7). You need a `URL` class for testing. You can use the `URL` class that is part of the JAVA 1.4 `java.net.URL` package, documented at *http://java.sun.com/*. Simply include `import java.net.URL` in `URLTest.java` and you can start running JUNIT.

3.6 Extend `URLTest.java` to include tests for other methods of the `URL` class. Is the documentation precise enough to let you write test cases?

3.7 In the model-view-controller pattern (Figure 3.7), every observer still depends on a given model. How can you use the dependence inversion principle to break this dependence?

3.8 When it comes to breaking dependences, there are alternatives to introducing abstract classes. Sketch and discuss

(a) The usage of *macros* (C, C++)
(b) The usage of *aspects* (see Section 8.2.3 in Chapter 8)

to break the dependence illustrated in Example 3.11.

3.9 JUNIT works fine to discover defects at the unit level, but fails if a failure is caused by multiple units. Discuss.

> Software features that can't be demonstrated by automated tests simply don't exist.
>
> – KENT BECK
> *Extreme Programming Explained* (2000)