10 retpahC

sdrawkcaB gnikroW

Our next task was Task 5:*Forgot Password Task*. In this task we needed to respond to a user who had forgotten his password. The idea was to put up a simple page that prompted the user for his E-mail address. This page would invoke a servlet that looked up the user's password and emailed it to him.

We still had not solved the problem of testing code in servlets. But we had learned from our past errors that servlets don't stay simple for long and need unit tests. Therefore we decided to put the bulk of the implementation in a testable, non-servlet, class that would be called by the servlet.

TestNoUser

In thinking about the test cases we realized the most interesting ones were those in which we were trying to recover from either a database or mailer problem. For example, the user's E-mail address doesn't exist; or his E-mail address confuses the mailer.

In previous tasks we had loaded the database with special test entries. This forced us to go to extra effort to maintain these entries. Tests could give false results if the database entries were not properly managed. (See "Hit the button twice.," on page 70).

It would be better if we could completely control the test data and conditions from within the test case. We wondered how we could do this?

Ward Cunningham recommends writing the code you want to see without worrying about the infrastructure that supports it. What did we want the tests to look like? Consider the test case where the user enters an E-mail address that is not present in the database. We wrote the following:

```
assert(remind("nobody@home")==NOEMAILFOUND);
```

Based upon this assertion, it was clear to us that the `remind` function's responsibilities were to look up the E-mail address in the database and then send the user the reminder message.

However, this particular test could be thwarted if somebody accidentally put `nobody@home` into the database. We needed a way to isolate the test case from the database.

Spoofing. Rather than allowing `PasswordReminder` direct access to the database, we created an interface that `PasswordReminderDatabase` could use to access the database. Then, inside the test program, we created an anonymous inner class that implemented that interface and always returned `null`.(See Listings 34 and 35.)

Listing 34

PasswordReminderDatabase

```
public interface PasswordReminderDatabase
{
  public String findPasswordFromEmail(String email);
}
```

Listing 35

ForgotPasswordTest.testNoUser refactoring 1.

```
public void testNoUser()
  {
    String noUser = "nobody@home";
    PasswordReminderDatabase db =
      new PasswordReminderDatabase()
    {
      public String findPasswordFromEmail(String email)
      {
        return null;
      }
```

Listing 35 (Continued)

ForgotPasswordTest.testNoUser refactoring 1.

```
        };
        PasswordReminder pr = new PasswordReminder(db);
        assert(pr.remind(noUser)==
            PasswordReminder.NOEMAILFOUND);
    } // testNoUser
```

This technique is called *spoofing*. We fooled the `PasswordReminder` into thinking it had a real database when in reality we completely replaced the database with a *mock-object* that provided the behavior that we wanted to test.

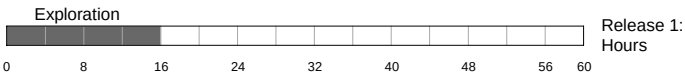
This wasn't compiling yet, because we hadn't written `PasswordReminder`. But we were getting a good picture of what `PasswordReminder` should look like. So we wrote it. (See Listing 36.)

Listing 36

PasswordReminder refactoring 1.

```
public class PasswordReminder
{
  public static final int OK = 0;
  public static final int NOEMAILFOUND = 1;

  private PasswordReminderDatabase itsDb;
  public PasswordReminder(PasswordReminderDatabase db)
  {
    itsDb = db;
  }

  public int remind(String email)
  {
    int status;
    String password = itsDb.findPasswordFromEmail(email);
    if (password != null)
      status = OK;
    else
      status = NOEMAILFOUND;

    return status;
  }

}
```

Clearly this function is incomplete. It doesn't send the reminder E-mail message if the database access succeeds. But one step at a time. Right now we have a green bar!



Working Backwards. Notice the order in which we did this. We started with the initial assertion. We built the test case around that assertion which helped us design the `PasswordReminder`. Then, finally, we wrote the incomplete version of `PasswordReminder`. We backed into the solution starting from an initial assertion and a set of initial constraints. This kept us from guessing about anything. Instead of anticipating and inventing the infrastructure we needed, or the methods we needed, or the objects we needed, we waited until the need was obvious.

But we weren't done. We may not have been done with `PasswordReminder`, but we were done with the current test case. We weren't going to write any more code into `PasswordReminder` until we had a failing test case.

TestGoodEmail

Next we wrote the test case that demonstrated that we could send the reminder message. We wanted that test case to look something like this:

```
assertEquals(PasswordReminder.OK,
             pr.remind("Bob"));
assertEquals("Bob", theToAddr);
assertEquals("Your Object Mentor Password",
             theSubj);
assertEquals(
  "Your Object Mentor Password is saba",
  theBody);
```

For the purposes of this test, we could assume that an OK return indicated that the mail was successfully sent. The rest of the asserts simply inspected the message to make sure it was properly formed.

We didn't want `PasswordReminder` to pass the E-mail message back to its caller. First, nobody other than the test was interested in that message. Second, in order to deal with the message

there would have had to be more code in the servlet. And we didn't want to put code in places that were hard to test.

Also, this unit test was not intended to test that mail actually got sent. Rather its intent was to ensure that `RemindPassword` had the correct logic. It would have been a pain to have E-mail get sent every time we ran this test.

Both problems can be solved by spoofing the mailer. We created a `Mailer` interface, passed it into the `PasswordReminder`, and implemented the test case as shown in Listings 37 and 38.

Listing 37

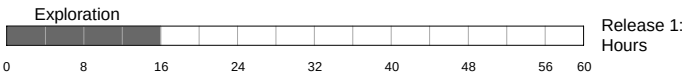
Mailer, refactoring 1.

```
public interface Mailer
{
  public void send(String to, String subj, String body);
}
```

Listing 38

ForgotPasswordTest refactoring 2.

```
public class ForgotPasswordTest extends TestCase
{

  private String theToAddr = "";
  private String theSubj = "";
  private String theBody = "";

  public void setup()
  {
    theToAddr = "";
    theSubj = "";
    theBody = "";
  }

  public void testGoodEmail()
  {
    PasswordReminderDatabase db =
      new PasswordReminderDatabase()
    {
      public String findPasswordFromEmail(String email)
      {return "saba";}
    };

    Mailer m = new Mailer()
    {
      public boolean send(String to, String subj,
                          String body)
      {
        theToAddr = to;
```



Listing 38 (Continued)

ForgotPasswordTest refactoring 2.

```
            theSubj = subj;
            theBody = body;
          }
        };

        PasswordReminder pr = new PasswordReminder(db, m);
        assertEquals(PasswordReminder.OK, pr.remind("Bob"));
        assertEquals("Bob", theToAddr);
        assertEquals("Your Object Mentor Password", theSubj);
        assertEquals("Your Object Mentor Password is saba",
                     theBody);
      }
    }
```

Notice that the mock-mailer loads the instance variables of the test case with the data from the message. This allows the test case to later assert that their values are correct. All that this really tests is that `PasswordReminder.remind` actually calls `Mailer.send` with the right arguments.

Next we added the Mailer argument to `PasswordReminder.remind` that would make this test case *compile*. Upon testing, the bar went red, as expected.

Next we added the code to `PasswordReminder.remind` that made the test pass. See Listing 39.

Listing 39

PasswordReminder refactoring 2.

```
public class PasswordReminder
{
  public static final int OK = 0;
  public static final int NOEMAILFOUND = 1;

  private PasswordReminderDatabase itsDb;
  private Mailer itsMailer;
  public PasswordReminder(PasswordReminderDatabase db,
                          Mailer m)
  {
    itsDb = db;
    itsMailer = m;
  }

  public int remind(String email)
  {
    int status = OK;
    String password = itsDb.findPasswordFromEmail(email);
```

Listing 39 (Continued)

PasswordReminder refactoring 2.

```
      if (password != null)
      {
        itsMailer.send(
          email,
          "Your Object Mentor Password",
          "Your Object Mentor Password is " + password);
      }
      else
        status = NOEMAILFOUND;

      return status;
    }
  }
```

TestBadEmail

Our last test case involved a failure of the mailer to deal with an E-mail address. Once again we started with an assertion.

```
assertEquals(PasswordReminder.EMAILERROR,
             pr.remind("Bob"));
```

Next we changed Mailer.send to return a `boolean`, and spoofed it to return false in this test case. Finally we added `EMAILERROR` to `PasswordReminder`.

The tests failed.

Next we changed `PasswordReminder.remind` to deal with the `boolean` return value from `Mailer`.

The tests passed. We couldn't think of any more test cases. We were done with the non-servlet code. See Listings 40-42

Listing 40

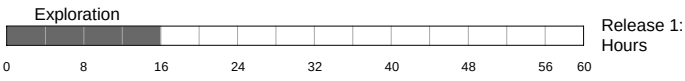
Mailer, refactoring 2.

```
public interface Mailer
{
  public boolean send(String to, String subj, String body);
}
```

Listing 41

ForgotPasswordTest, final refactoring.

```
public class ForgotPasswordTest extends TestCase
{
  public ForgotPasswordTest(String name)
  {
```



Listing 41 (Continued)

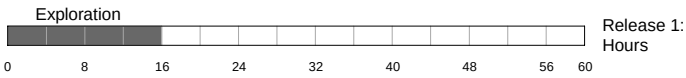
ForgotPasswordTest, final refactoring.

```
    super(name);
  }

  public static Test suite()
  {
    return new TestSuite(ForgotPasswordTest.class);
  }

  private boolean mailSent = false;
  private String theToAddr = "";
  private String theSubj = "";
  private String theBody = "";

  public void setup()
  {
    mailSent = false;
    theToAddr = "";
    theSubj = "";
    theBody = "";
  }

  public void testNoUser()
  {
    String noUser = "IDon'tExist";
    PasswordReminderDatabase db =
      new PasswordReminderDatabase()
    {
      public String findPasswordFromEmail(String email)
      {
        return null;
      }
    };
    Mailer m = new Mailer()
    {
      public boolean send(String to, String subject,
                          String body)
      {
        mailSent = true;
        return true;
      }
    };
    PasswordReminder pr = new PasswordReminder(db, m);
    assert(pr.remind(noUser) ==
           PasswordReminder.NOEMAILFOUND);
    assert(mailSent == false);
  } // testNoUser

  public void testBadEmail()
  {
    PasswordReminderDatabase db =
      new PasswordReminderDatabase()
    {
```

Listing 41 (Continued)

ForgotPasswordTest, final refactoring.

```
        public String findPasswordFromEmail(String email)
        {return "saba";}
      };

      Mailer m = new Mailer()
      {
        public boolean send(String to, String subj,
                            String body)
        {
          theToAddr = to;
          theSubj = subj;
          theBody = body;
          return false;
        }
      };

      PasswordReminder pr = new PasswordReminder(db, m);
      assertEquals(PasswordReminder.EMAILERROR,
                   pr.remind("Bob"));
      assertEquals("Bob", theToAddr);
      assertEquals("Your Object Mentor Password", theSubj);
      assertEquals("Your Object Mentor Password is saba",
                   theBody);
    } // testBadEmail

    public void testGoodEmail()
    {
      PasswordReminderDatabase db =
        new PasswordReminderDatabase()
      {
        public String findPasswordFromEmail(String email)
        {return "saba";}
      };

      Mailer m = new Mailer()
      {
        public boolean send(String to, String subj,
                            String body)
        {
          theToAddr = to;
          theSubj = subj;
          theBody = body;
          return true;
        }
      };

      PasswordReminder pr = new PasswordReminder(db, m);
      assertEquals(PasswordReminder.OK, pr.remind("Bob"));
      assertEquals("Bob", theToAddr);
```



Listing 41 (Continued)

ForgotPasswordTest, final refactoring.

```
    assertEquals("Your Object Mentor Password", theSubj);
    assertEquals("Your Object Mentor Password is saba",
                 theBody);
  }// testGoodEmail
} // ForgotPasswordTest
```

Listing 42

PasswordReminder, final refactoring

```
public class PasswordReminder
{
  public static final int OK = 0;
  public static final int NOEMAILFOUND = 1;
  public static final int EMAILERROR = 2;


  private PasswordReminderDatabase itsDb;
  private Mailer itsMailer;
  public PasswordReminder(PasswordReminderDatabase db,
                          Mailer m)
  {
    itsDb = db;
    itsMailer = m;
  }

  public int remind(String email)
  {
    int status = OK;
    String password = itsDb.findPasswordFromEmail(email);
    if (password != null)
    {
      boolean wasSent =
        itsMailer.send(
          email,
          "Your Object Mentor Password",
          "Your Object Mentor Password is " + password);
      if (!wasSent)
        status = EMAILERROR;
    }
    else
      status = NOEMAILFOUND;

    return status;
  }
}
```

Implementing the Mock-Objects

The Mailer. Next we implemented the two *mock-object* interfaces. The `Mailer` was quite simple. First we wrote the test cases (List-

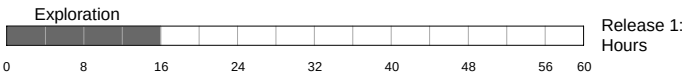
ing 43) and then we wrote `SMTPMailer` (Listing 44) to satisfy those test cases.

Listing 43
EmailTest

```
public class EmailTest extends TestCase
{
  public EmailTest(String name)
  {
    super(name);
  }

  public static Test suite()
  {
    return new TestSuite(EmailTest.class);
  }

  public void testBadEmailAddr()
  {
    Mailer m = new SMTPMailer("mail.wwa.com");
    assert(m.send("noAt", "subj", "body")== false);
  }

  public void testBadMailHost()
  {
    Mailer m = new SMTPMailer("wwa.com");
    assert(m.send("newkirk@objectmentor.com", "subj",
                  "body")== false);
  }

  public void testSuccess()
  {
    Mailer m = new SMTPMailer("mail.wwa.com");
    assert(m.send("newkirk@objectmentor.com", "subj",
                  "body")== true);
  }

}
```

Listing 44
SMTPMailer

```
public class SMTPMailer implements Mailer
{
  private String mailHost = null;

  public SMTPMailer(String mailHost)
  {
    this.mailHost = mailHost;
  }

  public boolean send(String to, String subj, String body)
  {
```



Listing 44 (Continued)
SMTPMailer

```
      boolean sent = true;

      try
      {
        MailMessage msg = new MailMessage(mailHost);
        msg.from("info@objectmentor.com");
        msg.to(to);
        msg.setSubject(subj);

        PrintStream out = msg.getPrintStream();
        out.println(body);
        msg.sendAndClose();
      }
      catch(IOException e)
      {
        e.printStackTrace();
        sent = false;
      }

      return sent;
    }
  }
```

PasswordReminderDatabase. Implementing the `PasswordReminderDatabase` was a bit more interesting. The `findPasswordFromEmail` function already exists in the `Database` class. The obvious strategy to implementing `PasswordReminderDatabase` is to make `Database` implement that interface. However, there are two reasons not to do this.

First, we liked the spoofing approach. We envisioned using it quite a bit more in the future. However, we didn't want every spoof to force us to change the `Database` class.

Second, and more to the point, `findUserByEmail` is static in `Database`. You can't put static functions in interfaces.

So we created an ADAPTER[1] called `PasswordDatabaseAdapter`. This class delegated the `findUserByEmail` function to the static function of `Database`. (See Listing 45)

1. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Reading, Mass.: Addison-Wesley, 1995.

Listing 45

PasswordDatabaseAdapter

```
public class PasswordDatabaseAdapter implements
PasswordReminderDatabase
{
  public String findPasswordFromEmail(String email)
  {
    return Database.findPasswordFromEmail(email);
  }
}
```

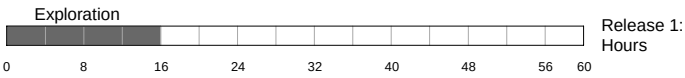
ForgotPassword Servlet

Finally, still backing into the solution, we wrote the servlet. It's pretty simple. It creates the necessary objects, invokes remind, and deals with the status. (See Listing 46.)

Listing 46

ForgotPassword Servlet

```
public class ForgotPassword extends HttpServlet
{
  private HttpServletRequest request;
  private HttpServletResponse response;
  private String email;
  private String url;

  public void doGet(HttpServletRequest req,
                    HttpServletResponse resp)
    throws ServletException, IOException
  {
    request = req;
    response = resp;
    email = (String)request.getParameter("email");
    url = (String)request.getParameter("url");
    PasswordReminderDatabase db =
      new PasswordDatabaseAdapter();
    Mailer m =
      new SMTPMailer(getInitParameter("mail-server"));
    PasswordReminder pr = new PasswordReminder(db, m);
    int status = pr.remind(email);
    if (status == PasswordReminder.OK)
      redirect("sent");
    else if (status == PasswordReminder.EMAILERROR)
      redirect("bademail");
    else if (status == PasswordReminder.NOEMAILFOUND)
      redirect("noemail");
  }

  private void redirect(String base)
    throws ServletException, IOException
  {
```



Listing 46 (Continued)

ForgotPassword Servlet

```
      String baseURI = getInitParameter(base);
      String uri =
        baseURI + "?email=" + email + "&url=" + url;
      response.sendRedirect(uri);
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletException, IOException
    {
      doGet(request, response);
    }
  }
```

There appears to be some duplication between this servlet and RedirectingServlet. We leave that refactoring as an exercise for the reader ;^).

Conclusion

After we got the servlet in place, we ran our manual acceptance tests. Everything worked as planned.

The total time spent on this task was about four hours instead of the six that we had estimated.

The most important lesson learned in this task was the practice of backing into solutions from initial premises and constraints. The technique of using mock-objects to keep all the test related information inside the test cases was very useful, and also helped us maintain our backwards thinking.