4

Testing

Fire is the test of gold; adversity, of strong men.

—Seneca (c. 3 B.C.–A.D. 65)

The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function.

Test-Driven Development

Suppose that we followed three simple rules.

1. Don't write any production code until you have written a failing unit test.
2. Don't write more of a unit test than is sufficient to fail or fail to compile.
3. Don't write any more production code than is sufficient to pass the failing test.

If we worked this way, we'd be working in very short cycles. We'd be writing just enough of a unit test to make it fail and then just enough production code to make it pass. We'd be alternating between these steps every minute or two.

The first and most obvious effect is that every single function of the program has tests that verify its operation. This suite of tests acts as a backstop for further development. It tells us whenever we inadvertently break some existing functionality. We can add functions to the program or change the structure of the program without fear that in the process, we will break something important. The tests tell us that the program is still behaving properly. We are thus much freer to make changes and improvements to our program.

A more important but less obvious effect is that the act of writing the test first forces us into a different point of view. We must view the program we are about to write from the vantage point of a caller of that program. Thus, we are immediately concerned with the interface of the program as well as its function. By writing the test first, we design the software to be *conveniently callable*.

What's more, by writing the test first, we force ourselves to design the program to be *testable*. Designing the program to be callable and testable is remarkably important. In order to be callable and testable, the software has to be decoupled from its surroundings. Thus, the act of writing tests first *forces us to decouple the software*!

Another important effect of writing tests first is that the tests act as an invaluable form of documentation. If you want to know how to call a function or create an object, there is a test that shows you. The tests act as a suite of examples that help other programmers figure out how to work with the code. This documentation is compilable and executable. It will stay current. It cannot lie.

Example of Test-First Design

Just for fun, I recently wrote a version of *Hunt the Wumpus*. This program is a simple adventure game in which the player moves through a cave, trying to kill the Wumpus before being eaten by the Wumpus. The cave is a set of rooms connected by passageways. Each room may have passages to the north, south, east, or west. The player moves about by telling the computer which direction to go.

One of the first tests I wrote for this program was `testMove` (Listing 4-1). This function created a new `WumpusGame`, connected room 4 to room 5 via an east passage, placed the player in room 4, issued the command to move east, and then asserted that the player should be in room 5.

Listing 4-1

```
[Test]

public void TestMove()
{
  WumpusGame g = new WumpusGame();
  g.Connect(4,5,"E");
  g.GetPlayerRoom(4);
  g.East();
  Assert.AreEqual(5, g.GetPlayerRoom());
}
```

All this code was written before any part of `WumpusGame` was written. I took Ward Cunningham's advice and wrote the test the way I wanted it to read. I trusted that I could make the test pass by writing the code that conformed to the structure implied by the test. This is called *intentional programming*. You state your intent in a test before you implement it, making your intent as simple and readable as possible. You trust that this simplicity and clarity points to a good structure for the program.

Programming by intent immediately led me to an interesting design decision. The test makes no use of a `Room` class. The action of *connecting* one room to another communicates my intent. I don't seem to need a `Room` class to facilitate that communication. Instead, I can simply use integers to represent the rooms.

This may seem counterintuitive to you. After all, this program may appear to you to be all about rooms, moving between rooms, finding out what rooms contain, and so on. Is the design implied by my intent flawed because it lacks a `Room` class?

I could argue that the concept of connections is far more central to the `Wumpus` game than the concept of room. I could argue that this initial test pointed out a good way to solve the problem. Indeed, I think that is the case, but it is not the point I'm trying to make. The point is that the test illuminated a central design issue at a very early stage. *The act of writing tests first is an act of discerning between design decisions.*

Note that the test tells you how the program works. Most of us could easily write the four named methods of `WumpusGame` from this simple specification. We could also name and write the three other direction commands without much trouble. If later we wanted to know how to connect two rooms or move in a particular direction, this test will show us how to do it in no uncertain terms. This test acts as a compilable and executable document that describes the program.

Test Isolation

The act of writing tests before production code often exposes areas in the software that ought to be decoupled. For example, Figure 4-1 shows a simple UML diagram of a payroll application. The `Payroll` class uses the `EmployeeDatabase` class to fetch an `Employee` object, asks the `Employee` to calculate its pay, passes that pay to the `CheckWriter` object to produce a check, and, finally, posts the payment to the `Employee` object and writes the object back to the database.

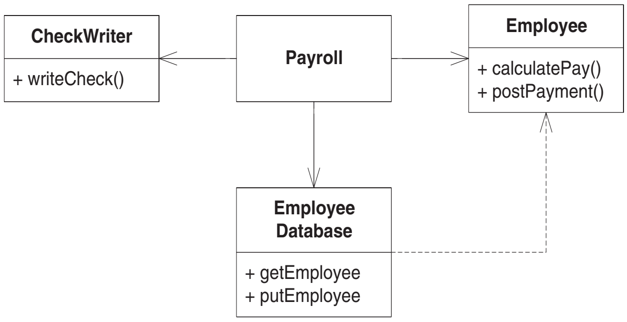


Figure 4-1
Coupled payroll model

Presume that we haven't written any of this code yet. So far, this diagram is simply sitting on a whiteboard after a quick design session.[1] Now we need to write the tests that specify the behavior of the `Payroll` object. A number of problems are associated with writing this test. First, what database do we use? `Payroll` needs to read from some kind of database. Must we write a fully functioning database before we can test the `Payroll` class? What data do we load into it? Second, how do we verify that the appropriate check got printed? We can't write an automated test that looks on the printer for a check and verifies the amount on it!

The solution to these problems is to use the MOCK OBJECT pattern.[2] We can insert interfaces between all the collaborators of `Payroll` and create test stubs that implement these interfaces.

Figure 4-2 shows the structure. The `Payroll` class now uses interfaces to communicate with the `EmployeeDatabase`, `CheckWriter`, and `Employee`. Three MOCK OBJECTs have been created that implement these interfaces. These MOCK OBJECTs are queried by the `PayrollTest` object to see whether the `Payroll` object managed them correctly.

Listing 4-2 shows the intent of the test. It creates the appropriate MOCK OBJECTs, passes them to the `Payroll` object, tells the `Payroll` object to pay all the employees, and then asks the MOCK OBJECTs to verify that all the checks were written correctly and that all the payments were posted correctly.

Of course, this test is simply checking that `Payroll` called all the right functions with all the right data. The test is not checking that checks were written or that a true database

1. [Jeffries2001]
2. [Mackinnon2000]

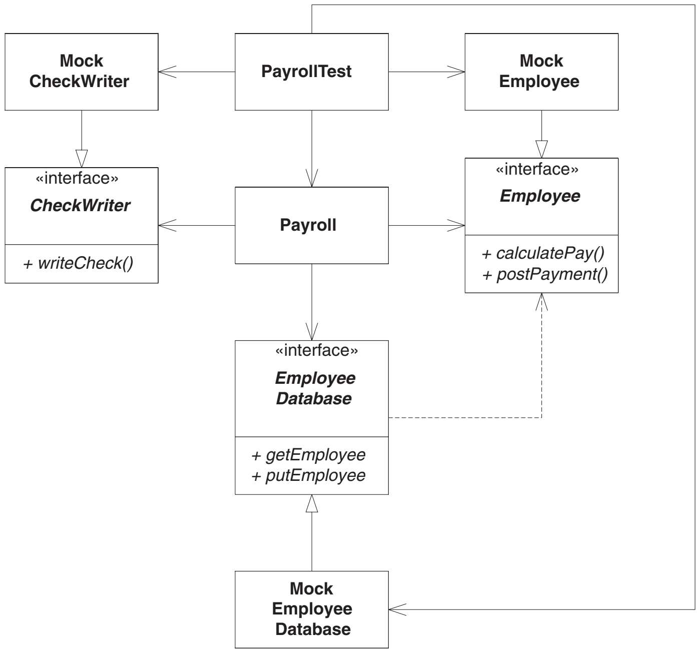


Figure 4-2
Decoupled `Payroll` using MOCK OBJECTS for testing

Listing 4-2
`TestPayroll`

```
[Test]
public void TestPayroll()
{
  MockEmployeeDatabase db = new MockEmployeeDatabase();
  MockCheckWriter w = new MockCheckWriter();
  Payroll p = new Payroll(db, w);
  p.PayEmployees();
  Assert.IsTrue(w.ChecksWereWrittenCorrectly());
  Assert.IsTrue(db.PaymentsWerePostedCorrectly());
}
```

was properly updated. Rather, it's checking that the `Payroll` class is behaving as it should in isolation.

You might wonder what the `MockEmployee` is for. It seems feasible that the real `Employee` class could be used instead of a mock. If that were so, I would have no compunction about using it. In this case, I presumed that the `Employee` class was more complex than needed to check the function of `Payroll`.

Serendipitous Decoupling

The decoupling of `Payroll` is a good thing. It allows us to swap in different databases and checkwriters for both testing and extending of the application. I think it is interesting that this decoupling was driven by the need to test. Apparently, the need to isolate the module under test forces us to decouple in ways that are beneficial to the overall structure of the program. *Writing tests before code improves our designs*.

A large part of this book is about design principles for managing dependencies. Those principles give you some guidelines and techniques for decoupling classes and packages. You will find these principles most beneficial if you practice them as part of your unit testing strategy. It is the unit tests that will provide much of the impetus and direction for decoupling.

Acceptance Tests

Unit tests are necessary but insufficient as verification tools. Unit tests verify that the small elements of the system work as they are expected to, but they do not verify that the system works properly as a whole. Unit tests are white box tests[3] that verify the individual mechanisms of the system. Acceptance tests are black box tests[4] that verify that the customer requirements are being met.

Acceptance tests are written by folks who do not know the internal mechanisms of the system. These tests may be written directly by the customer or by business analysts, testers, or quality assurance specialists. Acceptance tests are automated. They are usually composed in a special specification language that is readable and writable by relatively nontechnical people.

Acceptance tests are the ultimate documentation of a feature. Once the customer has written the acceptance tests that verify that a feature is correct, the programmers can read those acceptance tests to truly understand the feature. So, just as unit tests serve as compilable and executable documentation for the internals of the system, acceptance tests serve as compilable and executable documentation of the features of the system. In short, *the acceptance tests become the true requirements document*.

Furthermore, the act of writing acceptance tests first has a profound effect on the architecture of the system. In order to make the system testable, it has to be decoupled at the high architecture level. For example, the user interface has to be decoupled from the

3. A test that knows and depends on the internal structure of the module being tested.
4. A test that does not know or depend on the internal structure of the module being tested.

business rules in such a way that the acceptance tests can gain access to those business rules without going through the UI.

In the early iterations of a project, the temptation is to do acceptance tests manually. This is inadvisable because it deprives those early iterations of the decoupling pressure exerted by the need to automate the acceptance tests. When you start the very first iteration knowing full well that you must automate the acceptance tests, you make very different architectural trade-offs. Just as unit tests drive you to make superior design decisions in the small, acceptance tests drive you to make superior architecture decisions in the large.

Consider, again, the payroll application. In our first iteration, we must be able to add and delete employees to and from the database. We must also be able to create paychecks for the employees currently in the database. Fortunately, we have to deal only with salaried employees. The other kinds of employees have been held back until a later iteration.

We haven't written any code yet, and we haven't invested in any design yet. This is the best time to start thinking about acceptance tests. Once again, intentional programming is a useful tool for us to use. We should write the acceptance tests the way we think they should appear, and then we can design the payroll system accordingly.

I want the acceptance tests to be convenient to write and easy to change. I want them to be placed in a collaborative tool and available on the internal network so that I can run them any time I please. Therefore, I'll use the open-source FitNesse tool.[5] FitNesse allows each acceptance test to be written as a simple Web page and accessed and executed from a Web browser.

Figure 4-3 shows an example acceptance test written in FitNesse. The first step of the test is to add two employees to the payroll system. The second step is to pay them. The third step is to make sure that the paychecks were written correctly. In this example, we are assuming that tax is a straight 20 percent deduction.

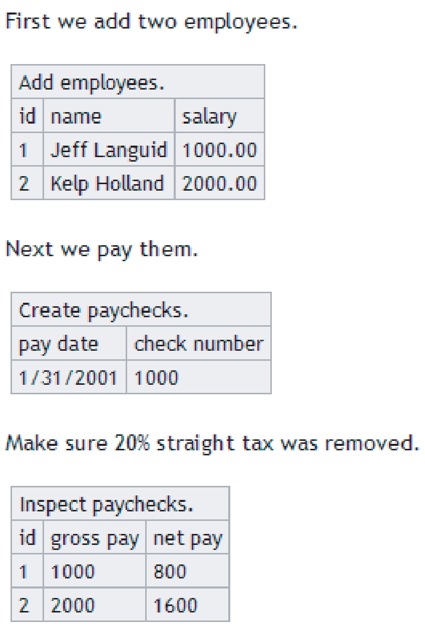
Clearly, this kind of test is very easy for customers to read and write. But think about what it implies about the structure of the system. The first two tables of the test are functions of the payroll application. If you were writing the payroll system as a reusable framework, they'd correspond to application programming interface (API) functions. Indeed, in order for FitNesse to invoke these functions, the APIs must be written.[6]

Serendipitous Architecture

Note the pressure that the acceptance tests placed on the architecture of the payroll system. The very fact that we considered the tests first led us to the notion of an API for the

5. `www.fitnesse.org`

6. The manner in which FitNesse calls these API functions is beyond the scope of this book. For more information, consult the FitNesse documentation. Also see [Mugridge2005].


First we add two employees.

Add employees.		
id	name	salary
1	Jeff Languid	1000.00
2	Kelp Holland	2000.00

Next we pay them.

Create paychecks.	
pay date	check number
1/31/2001	1000

Make sure 20% straight tax was removed.

Inspect paychecks.		
id	gross pay	net pay
1	1000	800
2	2000	1600

Figure 4-3
Sample acceptance test

functions of the payroll system. Clearly, the UI will use this API to achieve its ends. Note also that the printing of the paychecks must be decoupled from the `Create Paychecks` function. These are good architectural decisions.

Conclusion

The simpler it is to run a suite of tests, the more often those tests will be run. The more the tests are run, the sooner any deviation from those tests will be found. If we can run all the tests several time a day, then the system will never be broken for more than a few minutes. This is a reasonable goal. We simply don't allow the system to backslide. Once it works to a certain level, it never backslides to a lower level.

Yet verification is only one of the benefits of writing tests. Both unit tests and acceptance tests are a form of documentation. That documentation is compilable and executable and therefore accurate and reliable. Moreover, these tests are written in unambiguous languages that are readable by their audience. Programmers can read unit tests because they are written in their programming language. Customers can read acceptance tests because they are written in a simple tabular language.

Possibly the most important benefit of all this testing is the impact it has on architecture and design. To make a module or an application testable, it must also be decoupled. The more testable it is, the more decoupled it is. The act of considering comprehensive acceptance and unit tests has a profoundly positive effect on the structure of the software.

Bibliography

[Jeffries2001] Ron Jeffries, *Extreme Programming Installed*, Addison-Wesley, 2001.

[Mackinnon2000] Tim Mackinnon, Steve Freeman, and Philip Craig, "Endo-Testing: Unit Testing with Mock Objects," in Giancarlo Succi and Michele Marchesi, *Extreme Programming Examined*, Addison-Wesley, 2001.

[Mugridge2005] Rick Mugridge and Ward Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, Addison-Wesley, 2005.