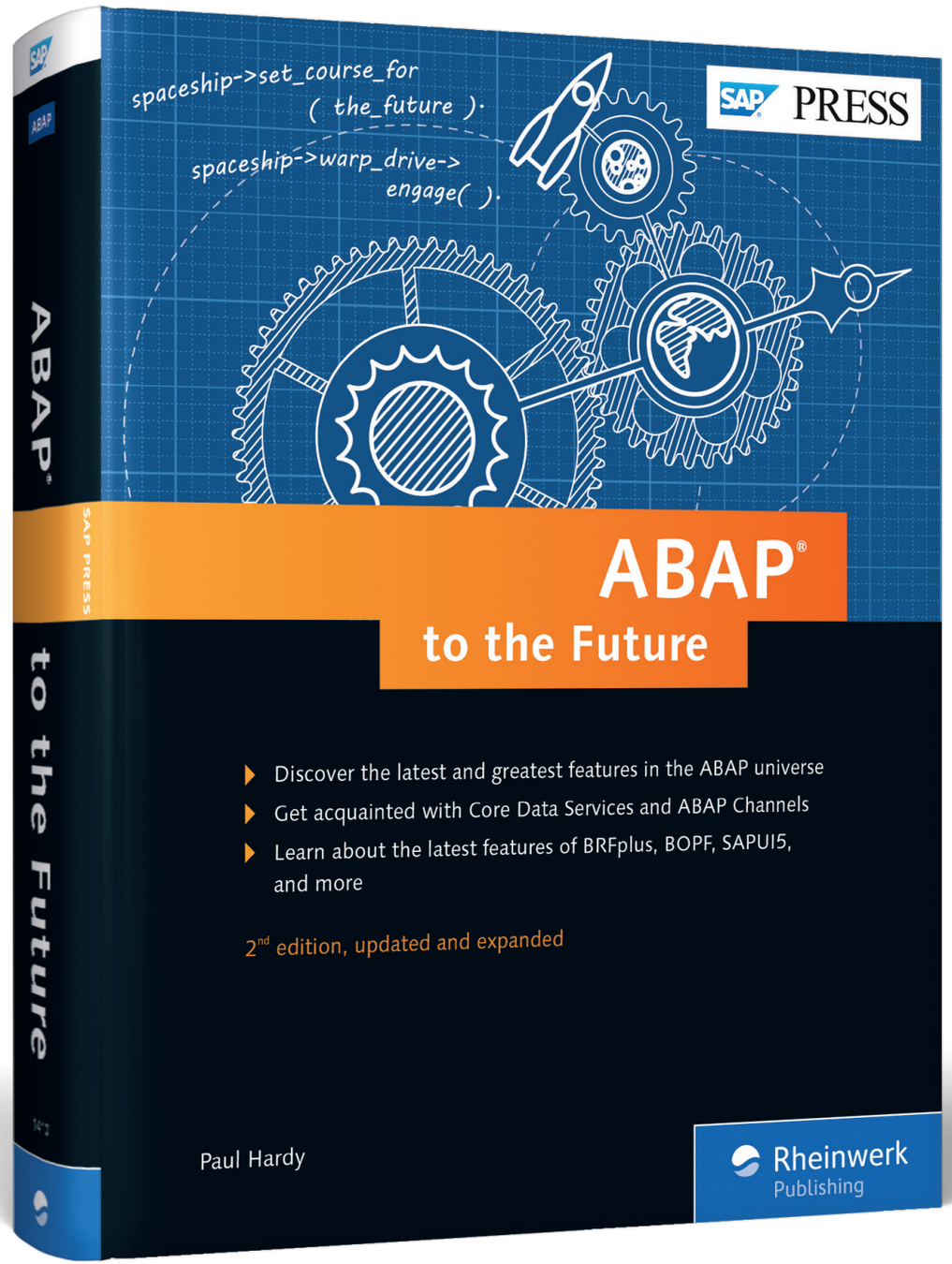
SAP PRESS
First-hand knowledge.
Rheinwerk Publishing



Reading Sample

This sample chapter describes how to use ABAP Unit for test-driven development (TDD) when creating and changing custom programs to make your changes with minimal risk to the business. This chapter explains what TDD is and how to enable it via the ABAP Unit framework.

Paul Hardy
ABAP to the Future
801 Pages, 2016, $79.95
ISBN 978-1-4932-1410-5

www.sap-press.com/4161

Code without tests is bad code. It doesn't matter how well-written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.
—Michael Feathers, Working Effectively with Legacy Code

3 ABAP Unit and Test-Driven Development

Nothing is more important, during the process of creating and changing custom programs, than figuring out how to make such changes with minimal risk to the business. The way to do this is via test-driven development (TDD), and the tool to use is ABAP Unit. This chapter explains what TDD is and how to enable it via the ABAP Unit framework.

In the traditional development process, you write a new program or change an existing one, and after you're finished you perform some basic tests, and then you pass the program on to QA to do some proper testing. Often, there isn't enough time, and this aspect of the software development lifecycle is brushed over—with disastrous results. TDD, on the other hand, is the opposite of the traditional development process: You write your tests before creating or changing the program. That turns the world on its head, which can make old-school developers' heads spin and send them running for the door, screaming at the top of their lungs. However, if they can summon the courage to stay in the room and learn about TDD, then they will be a lot better off.

The whole aim of creating your tests first is to make it so that, once the tests have all been written, as you create—and more importantly change—your application, you can follow the menu path Program • Test • Unit Test at any given instant to see a graphical display of what's currently working in your program and what's either not yet created or broken (Figure 3.1). This way, when the time comes to move the created or changed code into test, you can be confident that it's correct.

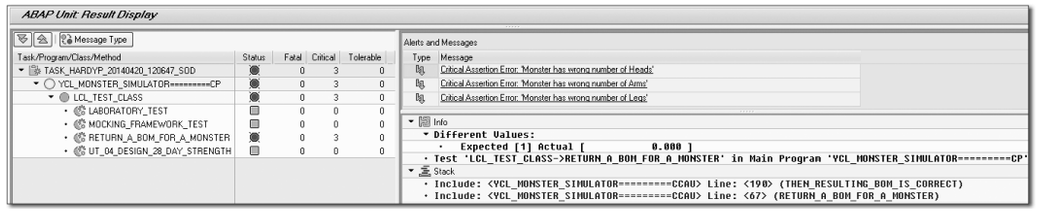


Figure 3.1 ABAP Unit Results Screen

Because that's a highly desirable outcome from everybody's perspective, this chapter explains how to transform your existing code into test-driven code via three main steps:

1. Eliminating dependencies in your existing programs
2. Implementing mock objects in your existing programs
3. Using ABAP Unit to write and implement test classes

Each step will be covered in detail. After that, some additional advice on how to improve TDD via automated frameworks will be presented. Finally, the chapter will conclude with a look at how to run large numbers of unit tests on different data sets without having to hard-code them all.

Unit Testing Procedural Programs

The examples in this chapter deal with testing object-oriented (OO) code (i.e., methods), and most of the examples of ABAP Unit you will find in books or on the web will also focus on OO programming. This doesn't mean that you can't test procedural programs; it's far easier to test an OO program, but it's not the end of the world to add unit tests to a function module or a procedural program.

You most likely have huge monolithic procedural programs in your system that it would be too difficult to rewrite in an OO fashion, because doing so would take a lot of time and not provide any new functionality. In such cases, whenever you are called on to maintain a section of that program—fixing a bug or adding new functionality—you can make relatively minor changes to break up the dependencies as described in this chapter, and then slowly add test methods to bring little islands of the program under test. As time goes by, more and more islands of the program will have tests, making it more and more stable, until one day the whole archipelago of the program will be covered.

3.1 Eliminating Dependencies

In the US game show "Jeopardy!", a contestant's answer must be in the form of a question—like this:

Game show host: Something that stops you dead in your tracks when you want to write a test.

Contestant: What is a dependency?

The contestant is correct. But to go into a little more detail, a dependency is what you have when you want to do a test for your productive code but you can't—because the productive code reads from the database or writes to the database or accesses some external system or sends an email or needs some input from a user or one of a million other things you cannot or do not want to do in the development environment.

As an example, consider a monolithic SAP program that schedules feeding time at the zoo and makes sure all the animals get the right amount and type of food. Everything works fine. The keepers have some sort of mobile device that they can query to see what animals need feeding next, and there's some sort of feedback they have to give after feeding the animals. They want to keep track of the food inventory, and so on; none of the details are important for this example.

All is well until one day when they get two pandas on loan from China, and the programmer has to make some changes to accommodate their specific panda-type needs (e.g., bamboo shoots) without messing up all the other animals. The programmer can do a unit test on the panda-specific changes he has made, but how can he be sure that an unforeseen side effect will not starve the lions, causing them to break out and eat all the birds and all the monkeys before breaking into the insect house and crushing and eating the beehive?

Normally, there's no way to do it; there are just too many dependencies. The program needs to read the system time, read the configuration details on what animal gets fed when, read and write inventory levels, send messages to the zookeepers, receive and process input from those same zookeepers, interface with assorted external systems, and so on.

However, you really don't want to risk the existing system breaking, leading the lions to dine on finch, chimps, and mushy bees, so how can we enable tests?

The first step is to break up the dependencies. In this section, you'll learn how to do exactly that, a process that involves two basic steps:

1. Look at existing programs to identify sections of code that are dependencies (and are thus candidates to be replaced by mock objects during a test).
2. Change your production code to separate out the concerns into smaller classes that deal with each different type of dependency (we'll look at two ways of doing so: one quick and dirty, the other slower and more time-intensive but ultimately more rewarding).

3.1.1 Identifying Dependencies

Listing 3.1 shows an example of a common ABAP application. In this case, our good old friend the baron (remember him from the Introduction?) doesn't want any neighboring mad scientists building monsters and thus encroaching on his market, so as soon as hears about such a competitor he drops a nuclear bomb on him (as any good mad scientist would). Listing 3.1 illustrates this by first getting the customizing settings for the missiles and making sure they're ready to fire, then confirming with the operator that he really wants to fire the missile, then firing the missile, and finally printing out a statement saying that the missile was fired. At almost every point in the code, you'll deal with something or somebody external to the SAP system: the database, the operator (user), the missile firing system, and the printer.

```
FORM fire_nuclear_missile.

* Read Database
  CALL FUNCTION 'READ_CUSTOMIZING'.
* Query Missile Sensor
  CALL FUNCTION 'GET_NUCLEAR_MISSILE_STATUS'.

* Business Logic
  IF something.
    "We fire the missile here
  ELSEIF something_else.
    "We fire the missile there
  ENDIF.

* Ask user if he wants to fire the missile
  CALL FUNCTION 'POPUP_TO_CONFIRM'.

* Business Logic
  CASE user_answer.
```



```
    WHEN 'Y'.
      "Off we go!
    WHEN 'N'.
      RETURN.
    WHEN OTHERS.
      RETURN.
  ENDCASE.

* Fire Missile
  CALL FUNCTION 'TELL_PI_PROXY_TO_FIRE_MISSILE'.

* Print Results
  CALL FUNCTION 'PRINT_NUCLEAR_SMARTFORM'.

ENDFORM."Fire Nuclear Missile
```

Listing 3.1 Common ABAP Application

In this code, you want to test two things:

- That you direct the missile to the correct target
- That if the user aborts halfway through, the missile does not actually fire

However, because of the way the routine is written, you can't do a meaningful test unless the following points are true:

- You have an actual database with proper customizing data.
- You have a link to the sensor on the missile.
- You have a user to say yes or no.
- You actually fire the missile to see what happens (possibly resulting in the world being destroyed).
- You have a printer hooked up.

These are all examples of *dependencies*. As long as they're part of your code, you can't implement TDD.

Unfortunately, most ABAP code traditionally looks like the example in Listing Section 3.1. Thus, when preparing an existing program to be unit tested, the first thing to do is make a list of anything that is not pure business logic (i.e., calls to the database, user input, calls to external systems, etc.), exactly as in the preceding bullet list.

3.1.2 Breaking Up Dependencies Using Test Seams

There are two ways to break up dependencies—quick and dirty versus slow and correctly. Often, the powers that be will force you into the quick and dirty method, so in version 7.5 of ABAP, we now have test seams designed to make such dependencies testable without major surgery on the existing program.

This approach only works on function groups and global classes. Old function groups often have lots of dependencies mixed in with the business logic, but it is horrifying if a global class gets into that state. If the code to be tested is in an executable program or a module pool but can be quickly moved to such a construct (function group/global class), then test seams are worth a try. If there's a lot of effort involved, then you're better off jumping straight to Section 3.1.3 and doing some serious rearranging of the code.

Warning: Houston, We Have a Problem

The `TEST-SEAM` concept should never be used in new programs, because it's a dirty workaround that horrifies all serious programmers (they would say production code should be unaware what parts of it are going to be tested) and is designed purely as an interim measure to use while redesigning old, badly written programs.

In the example in this section, we've moved the whole program into a function module so you can see how to use test seams. Inside the function module, each dependency is surrounded by a test seam with a unique name describing the dependency (Listing 3.2).

```
FORM fire_nuclear_missile.

  TEST-SEAM read_database.
* Here the dependency is needing an actual database with real data
    CALL FUNCTION 'ZREAD_MONSTER_CUSTOMIZING'.
  END-TEST-SEAM.

  TEST-SEAM missile_sensor.
* Here the dependency is needing contact with an actual missile system
    CALL FUNCTION 'ZGET_NUCLEAR_MISSILE_STATUS'.
  END-TEST-SEAM.

* Actual Business Logic (that you want to test)
* You would want to test that the missile gets sent to the right place
* i.e., gets dropped on your enemy, not on you
* IF something.
*   "We fire the missile here
* ELSEIF something_else.
*   "We fire the missile here
* ENDIF.

* Ask the user if they want to fire the missile
  TEST-SEAM user_input.
    DATA: user_answer TYPE char01.
    CALL FUNCTION 'POPUP_TO_CONFIRM'
      EXPORTING
        titlebar      = 'Missile Confirmation'
        text_question  = 'Do you want to launch the missile?'
        text_button_1  = 'Yes'
        text_button_2  = 'No'
        default_button = '2'
      IMPORTING
        answer         = user_answer2
      EXCEPTIONS
        text_not_found = 1
        OTHERS         = 2.

    IF sy-subrc <> 0.
      RETURN.
    ENDIF.

  END-TEST-SEAM.

* Some more business logic
* You would want to test that saying "no" prevented the
* missile from firing
  CASE user_answer.
    WHEN '1'.
      "Off We Go! Bombs Away!
    WHEN '2'.
      RETURN.
    WHEN OTHERS.
      RETURN.
  ENDCASE.

  TEST-SEAM missile_interface.
* Here the dependency is needing an actual missile to fire
    CALL FUNCTION 'ZTELL_PI_PROXY_TO_FIRE_MISSILE'.
  END-TEST-SEAM.

  TEST-SEAM printer.
* Here the dependency is needing an actual printer
    CALL FUNCTION 'ZPRINT_NUCLEAR_SMARTFORM'.
  END-TEST-SEAM.

ENDFORM.                    "fire_nuclear_missile
```

Listing 3.2 Surrounding Dependencies with Test Seams

Highlighting the dependencies in this way not only allows unit testing (as covered in Section 3.3) but also says—in letters of fire a thousand miles high—that the code is badly designed and needs changing, which, as mentioned earlier, you're often not allowed to do because the powers that be wrongly think such changes are "risky" when they are the exact reverse.

3.1.3 Breaking Up Dependencies Properly

The correct way (once they're identified) of breaking up dependencies—which takes a lot of effort—is redesigning your program so that it adopts a *separation of concerns* approach. This approach dictates that you have one class for database access, one for the user interface layer, and one for talking to an external system; that is, each class does one thing only and does it well. This is known as the *single responsibility principle*. Designing an application this way enables you to change the implementation of, say, your user interface layer without affecting anything else. This type of breakup is vital for unit tests.

Warning: Houston, We Have a Problem

As an aside and a warning, I've seen a great example in which someone split out all the database access into its own class, presumably following the separation of concerns model. However, that person also made every single variable and method static—and you can't subclass static methods. As a result, the end program still wasn't eligible for unit testing.

To illustrate the separation of concerns approach, take database access as an example. This means that you would go through your program looking for every `SELECT` statement and extract them out in a method of a separate database access class. Repeat the process for other functions of the programs, such as processing user input, communicating with external systems, and anything else you can identify as a dependency, each having one specific class that serves each such purpose. (You may be wondering how to decide which functions need to be split into their own class. Luckily, this is an iterative process; when you start writing tests and the test fails because it does not really have access to proper database entries, user input, or an external system, it will become clear what is a dependency and thus needs to be isolated into its own class.)

The next step is to change the production code to make calls to methods of these newly created classes. The end result will look like Listing 3.3.

```
FORM fire_nuclear_missile.

* Read Database
  mo_database_access->read_customising( ).
  mo_missile_interface->get_nuclear_missile_status( ).

* Business Logic
  IF something.
    "We fire the missile here
  ELSEIF something_else.
    "We fire the missile there
  ENDIF.

* Ask user if they want to fire the missile
  mo_user_interface->popup_to_confirm( ).

* Business Logic
  CASE user_answer.
    WHEN 'Y'.
      "Off we go!
    WHEN 'N'.
      RETURN.
    WHEN OTHERS.
      RETURN.
  ENDCASE.

* Fire Missile
  mo_missile_interface->tell_pi_proxy_to_fire_missile( ).

* Print Results
  mo_printer->print_nuclear_smartform( )..

ENDFORM."Fire Nuclear Missile
```

Listing 3.3 Calling Methods of Classes

Note that the functionality has not been changed at all; the only difference is that the calls to various external systems (the dependencies) are now handled by classes as opposed to functions or `FORM` routines. All that remains untouched is the business logic.

With the dependencies successfully eliminated, you can now implement mock objects.

3.2 Implementing Mock Objects

After you've isolated each dependency into its own class, you can change your existing programs to take advantage of the ABAP Unit framework. There are two steps to this:

1. Create *mock* objects that appear to do the same thing as real objects dealing with database access and the like, but which are actually harmless duplicates solely for use in unit tests.
2. Make sure that all the classes under tests (often a unit test will use several classes, but there is always one main one that you are testing—the *class under test*) are able to use these mock objects instead of the real objects, but only when a test is underway. This is known as *injection.*

Mock Objects vs. Stub Objects

When talking about mock objects, the terms *stub* and *mock* are often used interchangeably; technically, though, there is a difference. If you're testing how your class affects an external system, then the fake external system is a mock, and if you're testing how the fake external system affects your class, then the fake external system is a stub. (Either way, the point is that you use a fake external system for testing.)

Before jumping into creating mock objects and injection, let's first take a quick look at test injection, introduced with test seams. This is *not* how you should implement mock objects, but you should see it in action at least once before dismissing it.

3.2.1 Test Injection for Test Seams

Test injection for test seams is the poor man's version of implementing mock objects. Instead of replacing entire routines with a duplicate inside a mock class, you flag sections (one or more lines of code) inside of such routines as shown in Section 3.1.2 so they can be replaced with different code during a test. In other words, you have the option to surround sections of production code with test seams. When the program runs in production, the actual code within the test seam is executed. When running a test, you define some bogus code that runs instead, the format of which is as shown in Listing 3.4.


```
METHOD fire_nuclear_missile."Test Method
    TEST-INJECTION read_database.
* Set assorted variables, as if you had read them from the
* actual database
    END-TEST-INJECTION.

    TEST-INJECTION user_input.
      user_answer = '1'.
    END-TEST-INJECTION.

    PERFORM fire_nuclear_missile.

  ENDMETHOD.
```

Listing 3.4 Test Injections for Test Seams

This works fine; a test injection can be empty and so no code is executed during the test, so no database data is read, no missile is fired, and all is well.

This is all well and good but *don't do it*; it's more trouble than it's worth, and if proper programmers catch you, they'll make you stand in the corner with a dunce cap on your head. Instead, proceed according to the next section.

3.2.2 Creating Mock Objects

For testing purposes, what you actually want is to define mock classes and mock objects. *Mock classes* are classes that run in the development environment. They don't really try to read and write to the database, send emails, fire nuclear missiles, and so on, but they test the business logic nonetheless. Mock objects follow the same principles as regular objects; that is, in the same way that a monster object is an instance of the real monster class, a mock monster object is an instance of a mock monster class.

This is where the basic features of OO programming come into play: subclasses and interfaces. To continue the previous example (about firing nuclear missiles), you'll next create a subclass of the database access class that doesn't actually read the database but instead redefines the database access methods to return hard-coded values based upon the values passed in. In Listing 3.5, you'll see some possible redefined implementations of methods in mock subclasses that could replace the real classes in the example.

```
METHOD read_customising. "mock database implementation
*---------------------------------------------------------------------*
* IMPORTING input_value
```

```
* EXPORTING export_vlaue
*--------------------------------------------------------------------*
  CASE input_value.
    WHEN one_value.
      export_value = something.
    WHEN another_value.
      export_value = something_else.
    WHEN OTHERS.
      export_value = something_else_again.
  ENDCASE.
ENDMETHOD. "read customising mock database implementation

METHOD popup_to_confirm. "mock user interface implementation
*--------------------------------------------------------------------*
* RETURNING rd_answer TYPE char01
*--------------------------------------------------------------------*
  rd_answer = '1'."Yes

ENDMETHOD. "mock user interface implementation

METHOD fire_missile. "Mock External Interface Implementation

* Don't do ANYTHING - it's just a test

ENDMETHOD. "Fire Missile - Mock Ext Interface - Implementation
```

Listing 3.5 Mock Method Redefinitions of Assorted Real Methods

In this example, you create subclasses of your database access class, your user interface class, and your external system interface class. Then, you redefine the methods in the subclasses such that they either do nothing at all or return some hard-coded values.

Object-Oriented Recommendation

In order to follow one of the core OO recommendations—to favor composition over inheritance—you should create an interface that's used by your real database access class and also have the mock class be a subclass that implements that interface. In the latter case, before ABAP 7.4, you'd have to create blank implementations for the methods you are not going to use, and that could be viewed as extra effort. Nevertheless, interfaces are a really Good Thing and actually save you effort in the long run. Once you read books like *Head First Design Patterns* (see the Recommended Reading box at the end of the chapter), you'll wonder how you ever lived without building up class definitions using interfaces.

3.2.3 Proper Injection

Usually, classes in your program make use of smaller classes that perform specialized functions. The normal way to set this up is to have those helper classes as private instance variables of the main class, as shown in Listing 3.6.

```
CLASS lcl_monster_simulator DEFINITION.

  PRIVATE SECTION.
    DATA:
    "Helper class for database access
    mo_pers_layer TYPE REF TO zcl_monster_pers_layer,
    "Helper class for logging
    mo_logger     TYPE REF TO zcl_logger.

ENDCLASS.
```

Listing 3.6 Helper Classes as Private Instance Variables of Main Class

These variables are then set up during construction of the object instance, as shown in Listing 3.7.

```
METHOD constructor.

    CREATE OBJECT mo_logger.

    CREATE OBJECT mo_pers_layer
      EXPORTING
        io_logger   = mo_logger     " Logging Class
        id_valid_on = sy-datum.     " Validaty Date

  ENDMETHOD.                        "constructor
```

Listing 3.7 Variables Set Up during Construction of Object Instance

However, as you can see, this design does not include any mock objects, which means that you have no chance to run unit tests against the class. To solve this problem, you need a way to get the mock objects you created earlier inside the class under test. The best time to do this is when an instance of the class under test is being created.

When creating an instance of the class under test, you use a technique called *constructor injection* to make the code use the mock objects so that it behaves differently than it would when running productively. With this technique, you still have private instance variables of the database access classes (for example), but

now you make these into optional import parameters of the constructor. The constructor definition and implementation now looks like the code in Listing 3.8.

```
PUBLIC SECTION.
    METHODS: constructor   IMPORTING
    io_pers_layer TYPE REF TO zcl_monster_pers_layer OPTIONAL
    io_logger     TYPE REF TO zcl_logger             OPTIONAL.

METHOD constructor."Implementation

    IF io_logger IS SUPPLIED.
      mo_logger = io_logger.
    ELSE.
      CREATE OBJECT mo_logger.
    ENDIF.

    IF io_pers_layer IS SUPPLIED.
      mo_pers_layer = io_pers_layer.
    ELSE.
      CREATE OBJECT mo_pers_layer
        EXPORTING
          io_logger   = mo_logger    " Logging Class
          id_valid_on = sy-datum.    " Validity Date
    ENDIF.

ENDMETHOD."constructor implementation
```

Listing 3.8 Constructor Definition and Implementation

The whole idea here is that the constructor has optional parameters for the various classes. The main class needs these parameters in order to read the database, write to a log, or communicate with any other external party that's needed. When running a unit test, you pass in (*inject*) mock objects into the constructor that simply pass back hard-coded values of some sort or don't do anything at all. (In the real production code, you don't pass anything into the optional parameters of the constructor, so the real objects that do real work are created.)

Arguments Against Constructor Injection

Some people have complained that the whole idea of constructor injection is horrible, because a program can pass in other database access subclasses when executing the code for real outside of the testing framework. However, I disagree with that argument, because constructor injection can give you benefits outside of unit testing.

As an example, consider a case in which a program usually reads the entire monster-making configuration from the database—unless you're performing unit testing, when you pass in a fake object that gives hard-coded values. Now, say a requirement comes in that the users want to change some of the configuration values on screen and run a what-if analysis before saving the changes. One way to do that is to have a subclass that uses the internal tables in memory as opposed to the ones in the database, and you pass that class into the constructor when running your simulator program in what-if mode.

At this point, you now have mock objects and a way to pass them into your program. This means that you're ready to write the unit tests.

3.3 Writing and Implementing Unit Tests

At last, the time has come to talk about actually writing the test classes and how to use the ABAP Unit framework. In this section, you'll walk through this process, which involves two main steps:

1. Set up the definition of a test class. The definition section of a class, as always, is concerned with the what, as in, "What should a test class do?" This is covered in Section 3.3.1.
2. Implement a test class. Once you know what a test class is supposed to do, you can go into the detail of how it's achieved technically. This is covered in Section 3.3.2.

Executable Specifications

Some people in the IT world like to call unit tests *executable specifications*, because they involve the process of taking the specification document, breaking it down into tests, and then, when the program is finished, executing these tests. If they pass, then you are proving beyond doubt that the finished program agrees with the specification. If you can't break the specification into tests, then it means that the specification is not clear enough to turn into a program. (Actually, most specifications I get fall into that category. But to be fair to the business analysts, there is only so much that you can write on the back of a post-it note.)

3.3.1 Defining Test Classes

There are several things you need to define in a test class, and the following subsections will go through them one by one. Broadly, the main steps in defining your test class are as follows:

1. Enable testing of private methods.
2. Establish the general settings for a test class definition.
3. Declare data definitions.
4. Set up unit tests.
5. Define the actual test methods.
6. Implement helper methods.

Start the ball rolling by creating a test class. Start with a global Z class that you've defined, and open it in change mode via SE24 or SE80. In this global class, follow the menu option Goto • Local Definitions • Implementations • Local Test Class.

Enabling Testing of Private Methods

To begin, enable testing of private methods. The initial screen shows just a blank page with a single comment, starting with `use this source file`. In Listing 3.9, you add not only the normal definition line but also a line about `FRIENDS`; this is the line that enables you to test the private methods of your main class. In the following example, the global class you'll be testing is the monster simulator, and the only way you can test its private methods is if you make it friends with the test class.

```
*"* use this source file for your ABAP unit test classes
CLASS lcl_test_class DEFINITION DEFERRED.

"Need to make the class under test "friends" with the test class
"in order to enable testing for private methods
CLASS zcl_monster_simulator DEFINITION LOCAL FRIENDS lcl_test_class.
```

Listing 3.9 Defining Test Class

A lot of people say that testing private classes is evil and that you should only test the methods the outside world can see. However, over the course of time so many bugs have been found in any method at all, be it public or private, that you should have the option to test anything you feel like.

General Settings for a Test Class Definition

Once you've created the class, it's time to establish the general settings for the class, as shown in Listing 3.10.


```
CLASS lcl_test_class DEFINITION FOR TESTING
   RISK LEVEL HARMLESS
   DURATION SHORT
   FINAL.
```

Listing 3.10 Test Class General Settings

Let's take this one line at a time. In the first line, you tell the system this is a test class and thus should be invoked whenever you're in your program and select the Unit Test option from whichever tool you're in (the menu path is subtly different depending on which transaction you are in).

Now, you come to `RISK LEVEL`. For each system, you can assign the maximum permitted risk level. Although unit tests never run in production, it's possible for them to run in QA or development. By defining a risk level, you could, for example, make it so that tests with a `DANGEROUS` risk level can't run in QA but can run in development. (Leaving aside that I feel unit tests should *only* be run in development, how could a unit test be dangerous? I can only presume it's dangerous if it really does alter the state of the database or fire a nuclear missile. Try as I might, I can't think why I would want a test that was dangerous. Tests are supposed to stop my real program being dangerous, not make things worse. Therefore, I always set this value to `HARMLESS`, which is what the tests I write are.)

Next is `DURATION`—how long you think the test will take to run. This is intended to mirror the `TIME OUT` dump you get in the real system when a program goes into an endless loop or does a full table scan on the biggest table in the database. You can set up the expected time periods in a configuration transaction.

How long should those time periods be? Well, I'll tell you how long *I* think a unit test should take to run: It should be so fast that a human cannot even think of a time period so small. The whole point of unit tests is that you can have a massive amount of them and not be afraid to run the whole lot after you've changed even one line of code. It's like the syntax check; most developers run that all the time, but they wouldn't if it took ten minutes. Hopefully, the only reason a unit test would take a minute or more to run is if it actually did read the database or process a gigantic internal table in an inefficient way. If you *are* worried about the method under test going into an endless loop or about having to process a really huge internal table—sequencing a human genome or something—then you could set the `DURATION` to `LONG`, and it would fail due to a time out. Thus far, though, I have never found a reason to set it to anything other than `SHORT`.

Declaring Data Definitions

Continuing with the definition of your test class, you've now come to the data declarations. The first and most important variable you declare will be a variable to hold an instance of the class under test (a main class from the application you're testing various parts of).

This class, in accordance with good OO design principles, will be composed of smaller classes that perform specific jobs, such as talking to the database. In this example, when the application runs in real life, you want to read the database and output a log of the calculations for the user to see. In a unit test, you want to do neither. Luckily, your application will be designed to use the injection process described in Section 3.2.3 to take in a database layer and a logger as constructor parameters, so you can pass in mock objects that will pretend to handle interactions with the database and logging mechanism. As you're going to be passing in such mock objects to a constructor, you need to declare instance variables based on those mock classes (Listing 3.11).

```
PUBLIC SECTION.

PRIVATE SECTION.
  DATA: mo_class_under_test TYPE REF TO zcl_monster_simulator,
        mo_mock_pers_layer  TYPE REF TO zcl_mock_pers_layer,
        mo_logger           TYPE REF TO zcl_mock_logger.
```

Listing 3.11 Defining Mock Classes for Injecting into Test Class

In Listing 3.8, you saw how in the constructor in production, the class would create the real classes, but during a unit test mock classes are passed into the constructor of the class under test by the `SETUP` method, which runs at the start of each test method.

The full list of the data definitions in the test class are shown in Listing 3.12. In addition to the mock classes, there are some global (to a class instance) variables for things such as the input data and the result. It's good to set things up this way because passing parameters in and out of test methods can distract someone looking at the code (e.g., a business expert) from making sure that the names of the test methods reflect what's supposed to be tested.

```
PRIVATE SECTION.
DATA: mo_class_under_test TYPE REF TO zcl_monster_simulator,
      mo_mock_pers_layer  TYPE REF TO zcl_mock_monster_pers_layer,
      mo_mock_logger      TYPE REF TO zcl_mock_logger,
```



```
      ms_input_data       TYPE zvcs_monster_input_data,
      mt_bom_data         TYPE ztt_monster_items,
      md_creator          TYPE zde_monster_creator_name.
```

Listing 3.12 Variables for Test Class Definition

After defining the data, you now need to say what methods are going to be in the test class.

Defining the SETUP Method

The first method to define is always the `SETUP` method, which resets the system state so that every test method behaves as if it were the first test method to be run. Therefore, any of those evil global variables knocking about must either be cleared or set to a certain value, and the class under test must be created anew. This is to avoid the so-called temporal coupling, in which the result of one test could be influenced by the result of another test. That situation would cause tests to pass and fail seemingly at random, and you wouldn't know if you were coming or going.

This method can't have any parameters—you'll get an error if you try to give it any—because it must perform the exact same task each time it runs and importing parameters might change its behavior. The code for defining the `SETUP` method is very simple:

```
METHODS: setup,
```

Defining the Test Methods

After defining the `SETUP` method, you'll move onto defining the actual test methods. At this stage, you haven't actually written any production code (i.e., the code that will run in the real application), and all you have is the specification. Therefore, next you're going to create some method definitions with names that will be recognizable to the person who wrote the specification (Listing 3.13). The `FOR TESTING` addition after the method definition says that this method will be run every time you want to run automated tests against your application.

```
return_a_bom_for_a_monster FOR TESTING
make_the_monster_sing FOR TESTING
make_the_monster_dance FOR TESTING
make_the_monster_go_to_france FOR TESTING
```

Listing 3.13 Unit Test Methods

These are the aims of the program to be written (sometimes these are called *use cases*); you want to be sure the application can perform every one of these functions and perform them without errors. This is why you need unit tests.

Implementing Helper Methods

The last step in defining the test class is to implement *helper methods* (i.e., methods in your test class definition without the `FOR TESTING` addition). These are normal private methods that are called by the test methods.

The purpose of helper methods is to perform low-level tasks for one or more of the actual test methods; in normal classes, public methods usually contain several small private methods for the same reason. Helper methods usually fall into two categories:

1. Helper methods that contain boilerplate code that you want to hide away (because although you need this code to make the test work, it could be a distraction to someone trying to understand the test)
2. Helper methods that call one or more ABAP statements for the sole purpose of making the core test method read like plain English

Inside each unit test method (the methods that end with `FOR TESTING`), you will have several helper methods with names that have come straight out of the specification. As an example, the specification document says that the main purpose of the program is to return a bill of materials (BOM) for a monster, and you do that by having the user enter various desired monster criteria, which are then used to calculate the BOM according to certain rules.

Helper methods have names that adhere to a concept known as *behavior-driven development*, the idea that tests should be managed by both business experts *and* developers. When using behavior-driven development, the general recommendation is to start all the test methods with `IT SHOULD`, with the `IT` referring to the application being tested (the class under test). Thus, you would have names such as `IT SHOULD FIRE A NUCLEAR MISSILE`, such names coming straight out of the specification that describes what the program is supposed to achieve.

In ABAP, you are limited to thirty characters for names of methods, variables, database tables, and so on—so you have to use abbreviations, which potentially makes ABAP less readable than languages like Java. In Java, you would have

really long test method names, like `It Should Return a BOM for a Purple Monster`, but in ABAP you can't afford to add the extra characters of `IT SHOULD` to the method name. Instead, you can put the `IT SHOULD` in a comment line with dots after it, padding the comment line out to thirty characters to make it really obvious what the maximum permitted length of the names of the test methods are. You will then declare the test methods underneath the dotted line, being aware of when you're running out of space for the name. An example is shown in Listing 3.14.

```
*--------------------------------------------------------------------*
* Specifications
*--------------------------------------------------------------------*
    "IT SHOULD.....................
    "User Acceptance Tests
    return_a_bom_for_a_monster FOR TESTING,
    make_the_monster_sing FOR TESTING
    make_the_monster_dance FOR TESTING
    make_the_monster_go_to_france FOR TESTING
```

Listing 3.14 Test Methods That Describe What an Application Should Do

Other Names for Behavior-Driven Development

Sometimes, these sorts of behavior-driven development unit tests are described as *user acceptance tests*. Although there is no actual user involved, the reason for the terminology is that this sort of test simulates the program exhibiting a behavior that the user would expect when he performs a certain action within the program. Outside of SAP, the testing framework called FitNesse describes itself as such an automated user acceptance testing framework.

You may also see behavior-driven development referred to as the *assemble/act/assert* way of testing (which doesn't read as much like natural language, but it makes some people very happy, because every word starts with the same letter).

Whatever you want to call these types of automated tests, they usually involve several methods—and often several classes as well—all working together, as shown in Listing 3.15.

```
*--------------------------------------------------------------------*
* Low-Level Test Implementation Methods
*--------------------------------------------------------------------*
    "GIVEN.........................
    given_monster_details_entered,
```

```
    "WHEN.........................
    when_bom_is_calculated,
    "THEN.........................
    then_resulting_bom_is_correct,
```

Listing 3.15 The GIVEN/WHEN/THEN Pattern for Unit Tests

As you can see in the preceding code, unit test methods that follow the behavior-driven development approach have three parts:

1. `GIVEN` describes the state of the system just before the test to be run.
2. `WHEN` calls the actual production code you want to test.
3. `THEN` uses `ASSERT` methods to test the state of the system after the class under test has been run.

Use Natural Language

A test method is supposed to be able to be viewed by business experts to see if the test matches their specifications, so it has to read like natural language. Often, if business experts see even one line of ABAP, their eyes glaze over and you've lost them for good.

3.3.2 Implementing Test Classes

Now that you've defined the test class, you can go ahead with the process of implementing it. At the end of this step, you'll be able to show the business expert who wrote the initial specification that you've made the specification into a program that does what it says on the side of the box.

Given this, each implementation of a test method should look like it jumped straight out of the pages of the specification and landed inside the ABAP Editor (Listing 3.16).

```
METHOD return_a_bom_for_a_monster.

  given_monster_details_entered( ).

  when_bom_is_calculated( ).

  then_resulting_bom_is_correct( ).

ENDMETHOD. "Return a BOM for a Monster (Test Class)
```

Listing 3.16 Implementation of Test Class

The steps for implementing the test classes are as follows:

1. Setting up the test
2. Preparing the test data
3. Calling the production code to be tested
4. Evaluating the test result

Step 1: Setting Up the Test

Our class under test has lots of small objects that need to be passed into it. For the first example, you'll manually create all those small objects and pass them into the constructor method of the class under test to get the general idea of constructor injection. Later on, you'll find out how to reduce the amount of code needed to do this.

As there's no guarantee about the order in which the test methods will run, you want every test method to run as if it were the first test method run, to avoid tests affecting each other. Therefore, when setting up the test, you create each object instance anew and clear all the horrible global variables, as shown in Listing 3.17.

```
  METHOD: setup.
*--------------------------------------------------------------*
* Called before every test
*--------------------------------------------------------------*
    CREATE OBJECT mo_mock_logger.
    CREATE OBJECT mo_mock_monster_pers_layer
      EXPORTING
        io_logger   = mo_logger
        id_valid_on = sy-datum.

    CREATE OBJECT mo_class_under_test
      EXPORTING
        id_creator    = md_creator
        io_pers_layer = mo_mock_pers_layer
        io_logger     = mo_mock_logger.

    CLEAR: ms_input_data,
           md_creator.

  ENDMETHOD. "setup
```

Listing 3.17 Create Class under Test and Clear Global Variables

At this point, you can be sure that during the test you won't actually read the database or output any sort of log, so you can proceed with the guts of the actual tests, which can be divided into the three remaining steps: preparing the test data, calling the production code to be tested, and evaluating the test result.

Step 2: Preparing the Test Data

You now want to create some test data to be used by the method being tested. In real life, these values could come from user input or an external system. Here, you'll just hard-code them (Listing 3.18). Such input data is often taken from real problems that actually occurred in production; for example, a user might have said, "When I created a monster using these values, everything broke down." There could be a large number of values, which is why you hide away the details of the data preparation in a separate method, to avoid distracting anybody reading the main test method.

```
METHOD given_monster_details_entered.

  ms_input_data-monster_strength      = 'HIGH'.
  ms_input_data-monster_brain_size    = 'SMALL'.
  ms_input_data-monster_sanity        = 0.
  ms_input_data-monster_height_in_feet = 9.

  md_creator = 'BARON FRANKENSTEIN'.

ENDMETHOD."Monster Details Entered - Implementation
```

Listing 3.18 Preparing Test Data by Simulating External Input

Step 3: Calling the Production Code to be Tested

The time has come to actually invoke the code to be tested; you're calling precisely one method (or other type of routine), into which you pass in your hard-coded dummy values and (usually) get some sort of result back.

The important thing is that the routine being called doesn't know it's being called from a test method; the business logic behaves exactly as it would in production, with the exception that when it interacts with the database or another external system it's really dealing with mock classes.

In this example, when you pass in the hard-coded input data, the real business logic will be executed, and a list of monster components is passed back (Listing 3.19).


```
"WHEN.....................
  METHOD when_bom_is_calculated.

 mo_class_under_test->simulate_monster_bom(
    EXPORTING is_bom_input_data = ms_input_data
    IMPORTING et_bom_data       = mt_bom_data ).

  ENDMETHOD."when_bom_is_calculated
```

Listing 3.19 Calling Production Code to be Tested

The method that calls the code to be tested should be very short—for example, a call to a single function module or a method—for two reasons:

1. **Clarity**
   Anyone reading the test code should be able to tell exactly what the input data is, what routine processes this data, and what form the result data comes back in. Calling several methods in a row distracts someone reading the code and makes them have to spend extra time working out what's going on.
2. **Ease of maintenance**
   You want to hide the implementation details of what's being tested from the test method; this way, even if those implementation details change, the test method doesn't need to change.

For example, in a procedural program, you might call two or three `PERFORM` statements in a row when it would be better to call a single `FORM` routine—so that if you were to add another `FORM` routine in the real program, you wouldn't have to go and add it to the test method as well. With procedural programs, it would be good to have a signature with the input and output values, but a lot of procedural programs work by having all the data in global variables. Such a program can still benefit from unit testing; it just requires more effort (possibly a lot more effort) in setting up the test to make sure the global variables are in the correct state before the test is run.

Step 4: Evaluating the Test Result

Once you have some results back, you'll want to see if they are correct or not. There are generally two types of tests:

1. **Absolutely basic tests**
   In Chapter 6, Section 6.3, you'll read about *design by contract*, which says what a method absolutely needs before it can work and what it absolutely must do.

In their book *The Pragmatic Programmer*, Andrew Hunt and David Thomas state that unit tests should test for method failures in terms of "contract violations"; for example, do you pass a negative number into a method to calculate a square root of that number (which is silly) or pass a monster with no head into a method to evaluate hat size (a more realistic example)?

2. **Data validation tests**
   This is what most people would call the normal type of unit test in which you have an expected result given a known set of inputs; for example, a method to calculate the square root of sixteen should return four or (back in the real world), when calling a method to supply monster hats, the `HATS_RECIEVED` returning parameter should be seven when the `MONSTER_HEADS` importing parameter is seven, as demonstrated in the famous movie Seven Heads for Seven Monsters.

Next, you'll learn how to test for really basic failures, the standard way of evaluating test results, and how you can enhance the standard framework when the standard mechanism doesn't do everything you want. Finally, you'll see how to achieve 100% test coverage.

Testing for Really Basic Failures

In Chapter 6, you'll see that each routine in a program has a "contract" with the code that calls it, and there are only two ways of violating that contract: Either the calling program is at fault because the input data is wrong (violated precondition) or the routine itself is at fault because the data returned is wrong (violated postcondition).

The idea is that this contract is used to define unit tests for the routine, and those tests are used to verify that the routine is behaving correctly; two sides of the same coin.

A failure of such a test indicates a really serious, fatal, end of the universe as we know it type of bug, which needs to be addressed before dealing with minor (in comparison) matters, like your program adding up one and one and getting three. Listing 3.20 shows how to code such tests.

```
METHOD return_a_bom_for_a_monster.

TRY.

    given_monster_details_entered( ).
```



```
    when_bom_is_calculated( ).

    then_resulting_bom_is_correct( ).

CATCH zcx_violated_precondition.
 cl_abap_unit_assert=>fail( 'Violated Contract Precondition' ).
CATCH zcx_violated_postcondition.
 cl_abap_unit_assert=>fail( 'Violated Contract Postcondition' ).
ENDTRY.

ENDMETHOD."Return a BOM for a Monster (Test Class)
```

Listing 3.20 Unit Test to Check for Basic Errors

The structure in Listing 3.20 is totally generic. In TDD, you create the structure before writing the actual code, so the nature of the pre- and postconditions will only be determined later. In this example, the precondition could be that the input data should be asking for sensible values, like wanting a murderous evil monster as opposed to a cute fluffy one, and the postcondition should be that the resulting monster is scary and not colored pink.

Evaluating Test Results in the Normal Way

Moving on to looking at return values, when you ran the test method that called the production code, either you got a result back—table `MT_BOM_DATA` in the example in Listing 3.21—or some sort of publicly accessible member variable of the class under test was updated—a status variable, perhaps. Next, you'll want to run one or more queries to see if the new state of the application is what you expect it to be in the scenario you're testing. This is done by looking at one or more variable values and performing an evaluation (called an assertion) to compare the actual value with the expected value. If the two values don't match, then the test fails, and you specify the error message to be shown to the person running the test (Listing 3.21).

```
 "THEN.......................
 METHOD then_resulting_bom_is_correct.

 DATA(bom_item_details) = mt_bom_data[ 1 ].

cl_abap_unit_assert=>assert_equals(
act = bom_item_details-part_quantity
exp = 1
msg = 'Monster has wrong number of Heads'
quit = if_aunit_constants=>no ).
```

```
bom_item_details = mt_bom_data[ 2 ].

cl_abap_unit_assert=>assert_equals(
act = bom_item_details-part_quantity
exp = 2
msg = 'Monster has wrong number of Arms'
quit = if_aunit_constants=>no ).

bom_item_details = mt_bom_data[ 3 ].

cl_abap_unit_assert=>assert_equals( act = bom_item_details-part_
quantity
exp = 1
msg = 'Monster has wrong number of Legs'
quit = if_aunit_constants=>no ).

ENDMETHOD."Then Resulting BOM is correct - Implementation
```

Listing 3.21 Using Assertions to Check If Test Passed

When evaluating the result, you use the standard `CL_ABAP_UNIT_ASSERT` class, which can execute a broad range of tests, not just test for equality; for example, you can test if a number is between five and ten. There's no need to go into all the options here; it's easier if you just look at the class definition in SE24. (For a bit more about `ASSERT`, see the box ahead.)

Multiple Evaluations (Assertions) in One Test Method

Many authors writing about TDD have stated that you should have only one `ASSERT` statement per test. As an example, you shouldn't have a test method that tests that the correct day of the week is calculated for a given date and at the same time tests whether an error is raised if you don't supply a date. By using only one `ASSERT` statement per test, it's easier for you to quickly drill down into what's going wrong.

I would change that rule slightly, so that you're only testing one outcome per test. Although your test might need several `ASSERT` statements to make sure it's correct, the fact that you are only testing one outcome will still make it easy to figure out what's wrong. The default behavior in a unit test in ABAP, however, is to stop the test method at the first failed assertion and not even execute subsequent assertions within the same method. Every test method will be called, even if some fail—but only the first assertion in each method will be checked.

You can avoid this problem by setting an input parameter. In Listing 3.21, there are three assertions. By adding the following code, you can make the method continue with subsequent assertions even if one fails:

```
quit = if_aunit_constants=>no
```


Defining Custom Evaluations of Test Results

The methods supplied inside `CL_ABAP_UNIT_ASSERT` are fine in 99% of cases, but note that there is an `ASSERT_THAT` option that lets you define your own type of test on the result. Let's look at an example of the specialized assertion `ASSERT_THAT` in action. First, create a local class that implements the interface needed when using the `ASSERT_THAT` method (Listing 3.22).

```
CLASS lcl_my_constraint DEFINITION.

  PUBLIC SECTION.
    INTERFACES if_constraint.

ENDCLASS.                    "lcl_my_constraint DEFINITION
```

Listing 3.22 ASSERT_THAT

You have to implement both methods in the interface (naturally); one performs whatever tests you feel like on the data being passed in, and the other wants a detailed message back saying what went wrong in the event of failure. In real life, you would have a member variable to pass information from the check method to the result method, but the example shown in Listing 3.23 just demonstrates the basic principle.

```
CLASS lcl_my_constraint IMPLEMENTATION.

  METHOD if_constraint~is_valid.
*--------------------------------------------------------------*
* IMPORTING data_object TYPE data
* RETURNING result      TYPE abap_bool
*--------------------------------------------------------------*
* Local Variables
    DATA: monster_description TYPE string.

    monster_description =  data_object.

    result = abap_false.

    CHECK monster_description           CS 'SCARY'.
    CHECK strlen( monster_description ) GT 5.
    CHECK monster_description           NS 'FLUFFY'.

    result = abap_true.

  ENDMETHOD.                    "IF_CONSTRAINT~is_valid
```

```
  METHOD if_constraint~get_description.
*-------------------------------------------------------*
* RETURNING result TYPE string_table
*-------------------------------------------------------*

    DATA(error_message) = 'Monster is not really that scary'.

    APPEND error_message TO result.

  ENDMETHOD.                    "IF_CONSTRAINT~get_description

ENDCLASS."My Constraint - Implementation
```

Listing 3.23 Implementation of a Custom Constraint Class

All that remains is to call the assertion in the `THEN` part of a test method, as shown in Listing 3.24.

```
DATA(custom_constraint) = NEW lcl_my_constraint( ).

cl_abap_unit_assert=>assert_that( exp = custom_constraint
                                  act = scariness_description ).
```

Listing 3.24 Call Assertion

As you can see, the only limit on what sort of evaluations you can run on the test results is your own imagination.

Achieving 100 Percent Test Coverage

When evaluating your results, you want to make sure that you've achieved 100% test coverage. In the same way that the US army wants no man left behind, you want no line of code to remain untested. That is, if you have an `IF` statement with lots of `ELSE` clauses or a `CASE` statement with 10 different branches, then you want your tests to ensure that every possible path is followed at some point during the execution of the test classes to be sure that nothing ends in tears by causing an error of some sort.

That's not as easy as it sounds. Aside from the fact that you need a lot of tests, how can you be sure you haven't forgotten some branches? Luckily, there's tool support for this. As of release 7.31 of ABAP, you can follow the menu path Local Test Classes • Execute • Execute Tests With • Code Coverage. As mentioned earlier in the book, the same feature is available through ABAP in Eclipse.

3.4 Automating the Test Process

In the example presented in this chapter, the code was deliberately simple in order to highlight the basic principles without getting bogged down with unnecessary detail. However, in the real world programs are never simple. Even if they start off simple, they keep growing and mutating, the ground of the original purpose becoming buried under the ever-falling snow of new functionality.

If you think about what you've read above, you'll see that this can lead to quite a large effort in writing the test code, in two areas:

1. **Setting up the class under test via the `SETUP` method**
   Well-designed OO programs use lots of small reusable classes, and these often need to be inserted into the main class during construction. The smaller classes often need still smaller classes inserted into them during their construction, and so on. In practical terms, this can mean a lot of lines of code to build up your test class, passing in assorted mock doubles and elementary data parameters, such as date, and organizational elements, such as plant or sales organization.
2. **Creating mock objects**
   Although you need mock objects to take the place of real objects during a test in order to avoid actual database access and the like, it can take a good deal of effort to create these and to write the logic inside them to return hard-coded values. Even worse, if you're passing in constructor objects as interfaces as opposed to actual classes (which all the OO experts recommend), then you need to expend even more effort, because you have to create an implementation for every method in your mock object that uses the interface, including the methods you're not interested in.

Luckily, there are solutions for both problems and ahead, you'll read about two frameworks to solve those problems:

- A framework to automate dependency injection (which I created myself)
- A framework to automate creating mock objects (SAP standard)

Then, you'll see how to combine both techniques. The section will end with a look at what to do when you have a really large number of situations to test for, which is often the case.

Mock Objects Framework

Prior to ABAP 7.4 SPS 9, the only framework available to automate mock object creation was the open-source framework MockA (see Recommended Reading at the end of the chapter).

Since then, SAP has introduced its own standard framework to fill that gap, which is described in this chapter.

3.4.1 Automating Dependency Injection

Let's say, for the sake of example, that the main class under test needs two object instances to be passed into it during construction, and one of those objects needs some parameters itself. In other words, the main class has *dependencies*: The main class *depends* on the fact that those object instances need to be created before it itself can be created.

In real life, this could lead to dozens of lines of code in your `SETUP` method, creating objects all over the place. This is bad, because the more `CREATE OBJECT` statements you have in your program, the less flexible it becomes. The logic goes as follows: Creating an object via `CREATE OBJECT` forces the resulting instance to be of a specific class. Having objects of a specific class makes your program less resistant to change. If your main class contains lots of little helper classes—as it should—then you may need to have a great deal of `CREATE OBJECT` statements, meaning that your program is full of rigid components.

This leads to two problems: First, you have to clutter up your code with a large number of `CREATE OBJECT` statements. Second, those `CREATE OBJECT` statements usually create instances of a hard-coded class type rather than a dynamically determined subclass.

Now, you're always going to have to state the specific subclass you want somewhere. However, you'll see that it's possible to decouple this from the exact instant you call the `CREATE OBJECT` statement and, as a beneficial by-product, have fewer `CREATE OBJECT` statements in the first place.

The process of passing the objects a class needs to create itself is known as *dependency injection*; you saw this at work in Section 3.2.3 when you manually passed mock objects into a constructor method. Here, you seek to automate the process by dynamically working out what objects (dependencies) an instance of the main

class needs to create itself and by creating and passing those objects in all at once. This will drastically reduce the amount of `CREATE OBJECT` statements—and because you're going to be using a program to dynamically determine *what* objects need to be created, that same program might as well also dynamically determine what *type* (subclass) those created objects should be.

Listing 3.25 is an example of this approach; this code attempts to achieve the same thing as the `SETUP` method in Listing 3.17, but with fewer `CREATE OBJECT` statements. If you look at the definition of the `SETUP` method earlier in the chapter, you'll see three `CREATE OBJECT` statements, and in each case the object you're creating has to be defined by a `DATA` statement. (Some objects also need data parameters that are elementary, my dear Watson.)

Listing 3.25 rewrites the same `SETUP` method, using a Z class created to use dependency injection. Look at this like a newspaper: Look at the four high-level method calls first (which represent the headlines), and then dive into the implementation of each component method (representing the actual newspaper article) one at a time.

First, set some values for elementary data parameters. Then, specify any subclasses you want to substitute for real classes. Finally, create an instance of the class under test.

```
"Set values for any elementary parameters that the objects
    "being created need
    zcl_bc_injector=>during_construction( :
    for_parameter = 'ID_CREATOR'  use_value = md_creator ),
    for_parameter = 'ID_VALID_ON' use_value = sy-datum ).

    "We want to use a test double for the database object
    zcl_bc_injector=>instead_of(
    using_main_class = 'ZCL_MONSTER_SIM_PERS_LAYER'
    use_sub_class    = 'ZCL_MOCK_MONSTER_PERS_LAYER' ).

    "Same deal for the logger
    zcl_bc_injector=>instead_of(
    using_main_class = 'ZCL_MONSTER_LOGGER'
    use_sub_class    = 'ZCL_MOCK_LOGGER' ).

    "Off we go!
    zcl_bc_injector=>create_via_injection(
    CHANGING co_object = mo_class_under_test ).
```

Listing 3.25 SETUP Method Rewritten Using Z Statement

As you can see, when using this approach you don't need DATA statements to declare the database access object or the logging object. In addition, you only have one CREATE statement. This may not seem like much in this simple example, but the advantage increases proportionally with the complexity of the class being tested.

Note

You can also use this same methodology if the importing parameter of the object constructor is an interface. You just pass the interface name in to the INSTEAD_OF method rather than the main class name.

The first method called in Listing 3.25 is the DURING_CONSTRUCTION method. This is shown in greater detail in Listing 3.26; it analyzes elementary parameters and then does nothing fancier than adding entries to an internal table.

```
METHOD during_construction.
* Local Variables
  DATA: dummy_string               TYPE string ##needed,
        data_element_name          TYPE string,
        parameter_value_information LIKE LINE OF mt_parameter_values.

  parameter_value_information-identifier = for_parameter.
  parameter_value_information-do_value = REF #( use_value ).

  CHECK sy-subrc = 0.

  CALL METHOD cl_abap_structdescr=>describe_by_data_ref
    EXPORTING
      p_data_ref           = parameter_value_information-do_value
    RECEIVING
      p_descr_ref          = DATA(type_description)
    EXCEPTIONS
      reference_is_initial = 1
      OTHERS               = 2.

  IF sy-subrc <> 0.
    RETURN.
  ENDIF.

  SPLIT type_description->absolute_name AT '=' INTO dummy_string data_
element_name.

  parameter_value_information-rollname = data_element_name.
```



```
  INSERT parameter_value_information INTO TABLE mt_parameter_values.

ENDMETHOD.
```

Listing 3.26 DURING CONSTRUCTION Method

The next method called as part of the rewritten SETUP method is the INSTEAD_OF method (Listing 3.27). This method takes in as parameters the subclasses you want to create instead of a superclass, and that relation is stored in a hashed table.

```
METHOD instead_of.
* Local Variables
  DATA: sub_class_to_use_info LIKE LINE OF mt_sub_classes_to_use,
        created_objects_info  LIKE LINE OF mt_created_objects.

  sub_class_to_use_info-main_class = using_main_class.
  sub_class_to_use_info-sub_class  = use_sub_class.

  "Add entry at the start, so it takes priority over previous
  "similar entries
  INSERT sub_class_to_use_info INTO mt_sub_classes_to_use INDEX 1.

  "A specific object instance can be passed in, sometimes
  "a generated instance created via a framework
  CHECK with_specific_instance IS SUPPLIED.
  CHECK with_specific_instance IS BOUND.

  created_objects_info-clsname = use_sub_class.
  created_objects_info-object  = with_specific_instance.
  INSERT created_objects_info INTO TABLE mt_created_objects.

ENDMETHOD.
```

Listing 3.27 INSTEAD_OF Method

The last part of the rewritten SETUP method is the CREATE_BY_INJECTION method (Listing 3.28). This is written as close to plain English as possible so that the code is more or less self-explanatory. In essence, you're passing the input values you just stored into the constructor method when creating your class under test and any smaller classes it requires.

```
METHOD create_via_injection.
* Local Variables
  DATA: class_in_reference_details TYPE REF TO cl_abap_refdescr.

* Determine the class type of the reference object passed in
  class_in_reference_details ?=
  cl_abap_refdescr=>describe_by_data( co_object ).
```

```
  DATA(class_in_type_details) =
  class_in_reference_details->get_referenced_type( ).
  DATA(class_passed_in)       =
  class_in_type_details->get_relative_name( ).

  "See if we need to create the real class, or a subclass
determine_class_to_create(
EXPORTING
  id_class_passed_in             = CONV #( class_passed_in )
  io_class_in_type_details       = class_in_type_details
IMPORTING
  ed_class_type_to_create        = DATA(class_type_to_create)
  eo_class_to_create_type_detail = DATA(class_to_create_type_detail) ).

"Buffering causes unforseen results, so optional default "off"
  IF mf_use_buffering = abap_true.
    READ TABLE mt_created_objects INTO DATA(created_objects_info)
    WITH TABLE KEY clsname = class_type_to_create.

    IF sy-subrc = 0.
      "We already have an instance of this class we can use
      co_object ?= created_objects_info-object.
      RETURN.
    ENDIF.
  ENDIF."Do we buffer created objects?

"See if the object we want to create has parameters, and if so, fill th
em up
fill_constructor_parameters(
EXPORTING io_class_to_create_type_detail =
class_to_create_type_detail    " Class to Create Type Details
IMPORTING et_signature_values =
DATA(signature_value_table) ). " Constructor Parameters

create_parameter_object(
EXPORTING id_class_type_to_create = class_type_to_create
it_signature_values = signature_value_table " Parameter Values
CHANGING  co_object  = co_object )." Created Object

ENDMETHOD."Create by Injection
```

Listing 3.28 CREATE_BY_INJECTION Method

If you want to drill into this even more, you can download this code from *www.sap-press.com/4161* and run it in debug mode to see what's happening.

In summary, dependency injection provides a way to set up complicated classes while using a lot less code, which will enable you to create test classes with less effort.

Error Handling

There's virtually no error handling in the code just discussed (except for throwing fatal exceptions when unexpected things occur). This could be a lot more elegant—but it's the basic principle of automating dependency injection, not elegance, that's our current focus.

3.4.2 Automating Mock Object Creation: Test Double Framework

Unit testing frameworks have been around for quite some time in other languages, such as Java and C++. ABAP has joined the club rather late in the day. One advantage of this is that ABAP developers can look at problems other languages encountered—and solved—some years ago, and if they find the same problem, then they can implement the same sort of solution without having to reinvent the wheel. Mock objects are a great example of this: Many different mock object frameworks for Java were born to take a lot of the pain out of the process.

It wasn't until ABAP 7.4 that SAP created the ABAP Test Double Framework (hereafter ATDF because that's not such a mouthful and because I don't think I could get away with calling it the "mine's a double" framework), which is the equivalent of the mock object framework in all those other languages.

Good OO design recommends that virtually every class has its public signature defined via an interface. This is known in academic circles as the *Joe Dolce principle,* and the reasons that this is a Good Thing are too many and too complicated to go into here, but suffice it to say that this helps you follow the OO principle of favoring composition over inheritance. The ATDF works by using classes for which the public signature is defined via an interface.

For any given method, there are several generic behavior types that you would expect and that you'll want to test and thus also want to mock. Earlier in the chapter you saw some specific examples of these generic behaviors: either the correct result for a given set of input data or a violation of the methods contract with the calling program. The next two sections cover each case.

Verifying Correct Results

You can use the ATDF to verify correct results, as demonstrated in Listing 3.29. In our example, the class to be mocked is `ZCL_MONSTER_SIMULATOR`, which implements interface `ZIF_MONSTER_SIMULATOR`. Listing 3.29 demonstrates a number of

concepts; let's examine them one at a time before looking at the listing as a whole.

First, you create the mock object instance, which is an instance of a (nonexistent) class that implements the chosen interface. This dummy class has empty implementations for every method defined in the interface, as follows:

```
mock_monster_simulator ?= cl_abap_testdouble=>create( interface_name ).
```

In our example, the method to be mocked is `CALCULATE_SCARINESS`. This may seem odd, but the method name is not mentioned at the start of the process of setting this up; you just state the result that you're expecting back from this yet-unnamed method, as follows:

```
cl_abap_testdouble=>configure_call( mock_monster_simulator )-
>returning( 'REALLY SCARY' ).
```

Now is the time to overcomplicate things and say that, in this unit test, you expect the method to be called once and once only. The previous line of code is modified as follows:

```
cl_abap_testdouble=>configure_call( mock_monster_simulator )-
>returning( 'REALLY SCARY' )->and_expect( )->is_called_times( 1 ).
```

The names of the standard methods make the code read almost like English, which is a Good Thing.

Next, you set up the input data. As mentioned earlier in Listing 3.15, we have a special helper method for this called `GIVEN_MONSTER_DETAILS_ENTERED`, which fills in the values for the input structure, because there could be quite a few such values. Now, you can finally say (1) which method it is you want to mock, and (2) what input values should give the result you just specified (i.e., '`REALLY SCARY`'), as follows:

```
mock_monster_simulator->calculate_scariness( is_bom_input_data = ms_
input_data ).
```

From now on, calls to methods of our mock instance will be indistinguishable from calls to an instance of an actual class. To prove this, perform a real call to the same method (that may seem pointless now, but just you wait and see) to fill a variable with the scariness description, as follows:

```
scariness_description = mock_monster_simulator->calculate_
scariness( ms_input_data ).
```


It's fairly obvious what the result is going to be, but the test method ends with two assertions: one to see if the correct result has been returned and one to see if the method was called once and only once as expected.

Listing 3.29 combines these various lines of code. When you put them all together, what have you got? A lovely unit test!

```
METHOD mocking_framework_test.
* Local Variables
DATA: interface_name TYPE seoclsname
       VALUE 'ZIF_MONSTER_SIMULATOR',
       mock_monster_simulator TYPE REF TO zif_monster_simulator,
       scariness_description  TYPE string.

"Create the Test Double Instance
mock_monster_simulator ?= cl_abap_testdouble=>create( interface_name ).

"What result do we expect back from the called method?
cl_abap_testdouble=>configure_call( mock_monster_simulator )-
>returning( 'REALLY SCARY' )->and_expect( )->is_called_times( 1 ).

"Prepare the simulated input details e.g. monster strength
given_monster_details_entered( ).

"Say what method we are mocking and the input values
mock_monster_simulator->calculate_scariness( is_bom_input_data = ms_
input_data ).

"Invoke the production code to be tested
scariness_description = mock_monster_simulator->calculate_
scariness( ms_input_data ).

"Was the correct value returned?
cl_abap_unit_assert=>assert_equals(
exp = 'REALLY SCARY'
act = scariness_description
msg = 'Monster is not scary enough' ).

"Listen very carefully - was the method only called once?
cl_abap_testdouble=>verify_expectations( mock_monster_simulator ).

ENDMETHOD."Mocking Framework Test
```

Listing 3.29 Coding Unit Test without Needing Definitions and Implementations

As you can see, the ATDF does away with the need to create definitions and implementations of the class you want to mock. You only need to focus on what output values should be returned for what input values for what class. This

methodology uses ABAP's ability to generate temporary programs that live only in memory and only exist as long as the mother program is running. In effect, the framework writes the method definitions and implementations for you at runtime.

Note also that during the test a check is performed to see if the `CALCULATE_SCARINESS` method was in fact called in the preceding code. Even if a mock method doesn't do anything at all in a test situation, you still want to be sure that it's been called.

Method Chaining

Listing 3.29 also uses method chaining, a feature we discussed in Chapter 2. Four methods in a row are called on `CL_ABAP_TESTDOUBLE`: `CONFIGURE_CALL`, `RETURNING`, `AND_EXPECT`, and `IS_CALLED_TIMES`. Before release 7.02 of ABAP, you would have had to use four lines here, create a helper variable on the first line, and use that variable on each subsequent line.

Verifying Contract Violations

Sometimes you want to simulate the exception that's raised when a program encounters nonsense data, for example: if the input data being passed in by the calling program breaks the contract with the method being called.

In the running example used in this chapter, the `CALCULATE_SCARINESS` method has a contract with the calling program such that if the input data structure is totally blank, then there is no way the scariness can be calculated. This means that the calling program is at fault and an exception should be raised. You want to perform a test to make sure that in such a situation an exception actually is raised.

Do so by substituting the `RETURNING` method in `CONFIGURE_CALL` with `RAISE_EXCEPTION`, as shown in Listing 3.30.

```
METHOD mocking_exception_test.
* Local Variables
DATA: interface_name TYPE seoclsname
       VALUE 'ZIF_MONSTER_SIMULATOR',
       mock_monster_simulator TYPE REF TO zif_monster_simulator,
       scariness_description  TYPE string.

"Create the Test Double Instance
mock_monster_simulator ?= cl_abap_testdouble=>create( interface_name ).
```



```
"What result do we expect back from the called method?
DATA(lo_violation) = NEW zcx_violated_precondition_stat( ).
cl_abap_testdouble=>configure_call( mock_monster_simulator )->raise_
exception( lo_violation ).

"Prepare the simulated input details e.g. monster strength
CLEAR ms_input_data.

"Say what method we are mocking and the input values
TRY.
mock_monster_simulator->calculate_scariness( is_bom_input_data = ms_
input_data ).

"Invoke the production code to be tested
scariness_description = mock_monster_simulator->calculate_
scariness( ms_input_data ).

CATCH zcx_violated_precondition_stat.
  "All is well, we wanted the exception to be raised
  RETURN.
ENDTRY.

"Was the correct value returned?
cl_abap_unit_assert=>fail(
msg = 'Expected Exception was not Raised' ).

ENDMETHOD."Mocking Exception Test
```

Listing 3.30 Mocking Exception Using ATDF

Note that in order to mock an exception being raised, the exception being tested for has to be declared in the signature of the method being mocked. Exception classes inheriting from `CX_NO_CHECK` can't be mentioned in a method signature and thus can't be simulated.

See the Recommended Reading box at the end of the chapter for a link to the official blog detailing all the features of ATDF. There are more than I can go into here; what's more, new features are going to be added with each new release, which is wonderful news.

Alternatives to ATDF

Without this framework, the way to proceed is to create mock classes that are subclasses of the real class (e.g., `ZCL_MOCK_DATABASE_LAYER`), redefine some methods, and put some hard-coded logic inside the redefined method to return certain values based upon input values. You could also create a mock class that implements an interface—

but sometimes this is even more work, because in earlier versions of ABAP you need an implementation for every method in the interface.

3.4.3 Combining Dependency Injection and the ABAP Test Double Framework

It's quite possible that you didn't think the examples in the last section were the greatest thing since sliced bread, because *of course* the tests were going to pass; it was like adding up one and one and expecting two. The real value of ATDF comes to light when you pass your generated mock object into a larger class being tested. To demonstrate this, create a `ZCL_MONSTER_LABORATORY` class that takes the `SIMULATOR` object as input and then uses a complex set of business logic to say whether the monster is any good. That business logic is what we want to test.

To be more precise, the `EVALUATE_MONSTER` method of the `LABORATORY` object will call the `CALCULATE_SCARINESS` method of the monster simulator instance, which was passed into it at some point during processing. At that point, you want the mocked up result to be returned to the `CALCULATE_SCARINESS` method, as opposed to what the normal method would return in production.

The code in which the `ZCL_MONSTER_LABORATORY` class imports the `SIMULATOR` object during one of its methods is shown in Listing 3.31. The first part of the code is the same as in Listing 3.29 but with the mocked up simulator instance "injected" into the laboratory object.

```
METHOD laboratory_test.
* Local Variables
DATA: interface_name TYPE seoclsname
       VALUE 'ZIF_MONSTER_SIMULATOR',
       mock_monster_simulator TYPE REF TO zif_monster_simulator.

"Create the Test Double Instance
mock_monster_simulator ?= cl_abap_testdouble=>create( interface_name ).

"What result do we expect back from the called method?
cl_abap_testdouble=>configure_call( mock_monster_simulator )-
>returning( 'REALLY SCARY' )->and_expect( )->is_called_times( 1 ).

"Prepare the simulated input details e.g. monster strength
given_monster_details_entered( ).

"Say what method we are mocking and the input values
```



```
mock_monster_simulator->calculate_scariness( is_bom_input_data = ms_
input_data ).

* Now pass this mocked up simulator object into a class that
* expects a real object as an input.
DATA(laboratory) = NEW zcl_monster_laboratory( ).

DATA(is_the_monster_ok) = laboratory->evaluate_monster(
is_bom_input_data = ms_input_data
io_simulator      = mock_monster_simulator ).

cl_abap_unit_assert=>assert_equals(
exp = abap_true
act = is_the_monster_ok
msg = 'Monster is just not good enough' ).

ENDMETHOD."Laboratory Test
```

Listing 3.31 Passing In Mocked-Up Interface to Real Class

In Listing 3.31, large chunks of code that you would normally need have been removed; you didn't need to code either a definition or an implementation for the mock monster simulator class. Nonetheless, the end result is just as good as if you'd gone down the longer route. The laboratory object neither knows nor cares that what's been passed into it is not a real instance of a class, but instead a generated mock object.

Take this one step further: Pretend the monster simulator needs a whole raft of objects that are a pain to set up (e.g., a complicated laboratory object needs a monster creator object in its constructor, and a monster creator object needs a specialty in its constructor), so you combine ATDF with the dependency injection framework. In the injection framework class that's part of Listing 3.31, the method `FOR_THE` has a parameter through which you can pass in the generated object. You need this so that you can enable your ATDF-generated instance to be used when creating the class under test using injection. This is shown in Listing 3.32.

```
Same set up code as before …. Then ….
* Create the complicated receiving object via injection
DATA: laboratory TYPE REF TO zcl_complicated_laboratory,
       speciality TYPE zde_monster_type VALUE 'SCARY'.

"In unit tests, everything has to start in pristine condition
zcl_bc_injector=>reset( ).
```

```
"A Monster Creator (e.g., the baron ) needs a speciality
zcl_bc_injector=>during_construction( :
for_parameter = 'ID_SPECIALITY'  use_value = speciality ).

"A complicated laboratory needs a Monster Creator object  but this will
 be created automatically by the injector

"Pass in the instance of the simulator we have mocked up
zcl_bc_injector=>for_the(
interface_or_class    = monster_simulator_interface
use_specific_instance = mock_monster_simulator ).

"During injection the mocked up object gets passed in with
"all the other data we have set up
zcl_bc_injector=>create_via_injection(
CHANGING co_object = laboratory ).

DATA(is_the_monster_ok) = laboratory->evaluate_monster( ms_input_
data ).

cl_abap_unit_assert=>assert_equals(
exp = abap_true
act = is_the_monster_ok
msg = 'Monster is just not good enough' ).
```

Listing 3.32 Combining ATDF with Dependency Injection

In Listing 3.32, you're passing in a generated object that will be used when the injection class creates the class under test.

What the examples that culminate in Listing 3.32 show is that it's possible to simplify the `SETUP` method dramatically when creating the class under test. You can use ATDF to set up test doubles with a lot less effort and use injection to avoid having to code long strings of `CREATE OBJECT` statements that pass the results into each other before handing the end result into the class under test when it's finally created. The two frameworks weren't created with the intention of working together, but by a happy accident they fit together like the pieces of a jigsaw puzzle.

3.4.4 Unit Tests with Massive Amounts of Data

In the United Kingdom, children can buy *I-Spy* books, in which they have to spot various things. Once they've spotted them all, they send the completed list to Big Chief I-Spy, who sends them a feather in return. If you were on the lookout for a feather, you have may have spied that in all the preceding examples, regardless of

method, hard-coded data was used. The rest of the book goes on and on about how hard-coded values are the work of the devil, so there's some disparity here—a circle that needs to be squared.

In most cases, you want to test your method with a wide variety of possible inputs to make sure the correct result is returned in every case. One way to do this is to code one unit test with one set of inputs to ensure it comes back with the correct result, and then move the transport into test (or production) and wait for people to tell you everything falls apart when you input a different set of results. (Hopefully, you can see that might not be the ideal way to go about things.) It would be so much better if you started off with a wide range of scenarios, ran tests for all of them, made sure they all worked, and then moved the program to test. Everyone would be a lot happier—especially you.

Getting a list of scenarios was easy: I went to the business users (Igor and his hunchback mates) and asked for a list of a hundred sets of monster requirements and their monster BOMs. Before I could say "Jack Robinson," I had a spreadsheet in my hot little hands. Wonderful! Now, should I manually code one hundred different test methods, each with the same method call with a different set of inputs followed by assertions with a different set of results? Doesn't sound like much fun.

You could create a database table (but you might have to create different ones for different programs) or store the test data in the standard ECATT automated tests script system. My favorite solution to this problem, however, is an open-source project created by a programmer called Alexander Tsybulsky and his colleagues, who came up with a framework called the *Mockup Loader*, which lives on the GitHub site and can be downloaded to your system via the link found in the Recommended Reading box at the end of this chapter.

ABAP Open-Source Projects

Several times throughout this book, we'll refer to open-source ABAP projects, which started life in the SAP Code Exchange section of the SAP Community Network website but nowadays live on sites like GitHub. The obvious benefit is that these are free. Some development departments have rules against installing such things, but I feel they're just cutting off their nose to spite their face.

The important point to note is that these are not finished products, so installing them is not like installing the sort of SAP add-on you pay for. It's highly likely you'll encounter bugs and that the tool won't do 100% of what you want it to do. In both cases, I

strongly encourage you to fix the bug or add the new feature, and then update the open-source project so that the whole SAP community benefits.

The very first open-source project you'll need to install is SAPlink so that you can download any others you need easily. You can get SAPlink from *www.saplink.org*.

Before you begin, you should have the test data loaded inside the SAP development system (where the tests will run, of course). That's much better than having the test data on some sort of shared directory or, worse, on a developer's laptop.

The GitHub page for the Mockup Loader gives detailed instructions for storing a spreadsheet inside the MIME repository, which allows you to store various files (like spreadsheets) inside SAP. You can upload as many spreadsheets as you want: one for input data, one for output data, or both in one sheet, as in the following example. Even better—if you have different types of data, you can store each one as a sheet inside the one big worksheet, keeping everything in the same place.

Once the test data in the spreadsheet is uploaded into SAP, the fun begins. Listing 3.33 demonstrates a test method that evaluates lots of test cases at once, without any of the fancy things mentioned elsewhere in the chapter so as not to distract from what's being demonstrated.

In this example, the idea is to loop through different sets of customer requirements to make sure the correct "component split" is returned. For now, you can ignore SSATN and SSPDT and the percentage split; those elements will be detailed in Chapter 8.

Start with a spreadsheet with five columns; the first three are customer requirements (e.g., what the customer desires in a monster) and the last two are result columns (percentages of SSATN and SSPDT, respectively). At the start of the test method, declare a structure that exactly matches the columns in the spreadsheet; the spreadsheet has to have a header row that exactly matches the names of the fields in this structure.

When you upload your spreadsheet to the MIME repository using Transaction SMW0, you give the file a name. In the test method, you must also specify the fact that this is a MIME object and the name of that object. You could specify `FILE` and a directory path, but that would be uncool; you'd never be invited to parties again.

Then, create an instance of your mockup loader. If you spelled the name of the MIME object incorrectly, this is where you'll find out in a hurry, due to a fatal error message. Next, load the MIME object into an internal table based on the structure you declared earlier. At this point, if the columns in the structure don't match the columns in the spreadsheet, an exception is raised and the test fails. This is good: Making sure the test data format is correct is just another step in getting the unit tests to pass.

The rest is plain sailing: Loop through the test cases, call the method being tested each time with the specific test case input data, and see if the result matches the specific test case result data. As mentioned earlier, you can use the `QUIT` parameter to determine if you want to see all the results at once or stop at the first failure; the code in Listing 3.33 stops at the first failure.

```
METHOD mockup_loader.
* Local Variables
    TYPES: BEGIN OF l_typ_monster_test_data,
             strength   TYPE  zde_monster_strength,
             brain_size TYPE  zde_monster_brain_size,
             sanity     TYPE  zde_monster_sanity,
             ssatn      TYPE  zde_component_type_percentage,
             sspdt      TYPE  zde_component_type_percentage,
           END OF   l_typ_monster_test_data.

* Need to specify the type of the table, to make sure
* correct tests are done on the data loaded from MIME
    DATA test_cases_table TYPE TABLE OF l_typ_monster_test_data.

    "Name of Entry in SMW0
    zcl_mockup_loader=>class_set_source(
      i_type = 'MIME'
      i_path = 'ZMONSTER_TEST_DATA' ).

    TRY.
        DATA(mockup_loader) = zcl_mockup_loader=>get_instance( ).
      CATCH zcx_mockup_loader_error INTO DATA(loader_exception).
        cl_abap_unit_assert=>fail( loader_exception->get_text( ) ).
    ENDTRY.

    TRY.
"Load test cases. The format is SPREADSHEET NAME/Sheet Name
     mockup_loader->load_data(
      EXPORTING i_obj       = 'MONSTER_TEST_DATA/monster_tests'
      IMPORTING e_container = test_cases_table ).
```

```
      CATCH zcx_mockup_loader_error INTO loader_exception.
        cl_abap_unit_assert=>fail( loader_exception->get_text( ) ).
    ENDTRY.

    LOOP AT test_cases_table INTO DATA(test_case).
      mo_class_under_test->get_component_split(
        EXPORTING
          id_strength   = test_case-strength
          id_brain_size = test_case-brain_size
          id_sanity     = test_case-sanity
        IMPORTING
          id_ssatn      = DATA(actual_percentage_ssatn)
          id_sspdt      = DATA(actual_percentage_sspdt) ).

      cl_abap_unit_assert=>assert_equals(
      exp = test_case-ssatn
      act = actual_percentage_ssatn
      msg = |{ test_case-strength } + { test_case-brain_
size } + { test_case-sanity } gets incorrect SSATN %age| ).

      cl_abap_unit_assert=>assert_equals(
      exp = test_case-sspdt
      act = actual_percentage_sspdt
      msg = |{ test_case-strength } + { test_case-brain_
size } + { test_case-sanity } gets incorrect SSPDT %age| ).

    ENDLOOP."Test Cases

  ENDMETHOD."Mockup Loader
```

Listing 3.33 Test Method to Load Multiple Test Cases

Every so often, a new problem will arise in production; you'll just add a new line to your spreadsheet, upload the changed version, and then fix the newly added (broken) test.

This approach can be combined with everything else mentioned in this section. For a nice (complicated) example, you could set up a bunch of mock objects with fake expected behavior, pass them into the class under test using dependency injection, and then run a bucket load of test cases using the mockup loaded. As Snoopy would say, "You see how it all comes together?"

3.5 Summary

Mountain climbers will tell you that their pastime is not easy, but it's all worth it once you've achieved the incredibly difficult task of climbing the mountain and are standing on the summit, on top of the world, able to see for miles. It may not seem similar on the surface, but unit testing is like that. It's not easy at all—quite the reverse—but once you've enabled your existing programs with full test coverage and you create all new programs using this methodology, then you too suddenly have a much-improved view.

Quite simply, you can make any changes you want to—radical changes—introduce new technology, totally refactor (redesign) the innards of the program, anything at all, and after you change even one line of code you can follow the menu path Test • Unit Test and know within seconds if you've broken any existing functions. This is not to be sneezed at. It is in fact the Holy Grail of programming.

One question that hasn't come up yet is this: What if someone else comes along and changes your program by adding a new feature, but accidently breaks something else and doesn't bother to run the unit tests, and thus doesn't realize that he's broken something? Clearly, you somehow need to embed the automated unit tests into the whole change control procedure. Conveniently, this leads nicely to the subject of the next chapter: ABAP Test Cockpit.

Recommended Reading

- ***Head First Design Patterns***
  Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra, O'Reilly Media, 2004
- **Behavior-Driven Development**
  http://dannorth.net/introducing-bdd (Dan North)
- **The Art of Unit Testing**
  http://artofunittesting.com (Roy Osherove)
- **The Pragmatic Programmer**
  https://en.wikipedia.org/wiki/The_Pragmatic_Programmer (Andrew Hunt and David Thomas, The Pragmatic Bookshelf, 1999)
- **Dependency Injection**
  http://scn.sap.com/community/abap/blog/2013/08/28/dependency-injection-for-abap (Jack Stewart)
- **mockA**
  http://uwekunath.wordpress.com/2013/10/16/mocka-released-a-new-abap-mocking-framework (Uwe Kunath)

- **ABAP Test Double Framework**
  http://scn.sap.com/docs/DOC-61154 (Parjul Meyana)
- **Mockup Loader**
  http://scn.sap.com/community/abap/blog/2015/11/12/unit-testing-mockup-loader-for-abap (Alexander Tsybulsky)

Contents

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

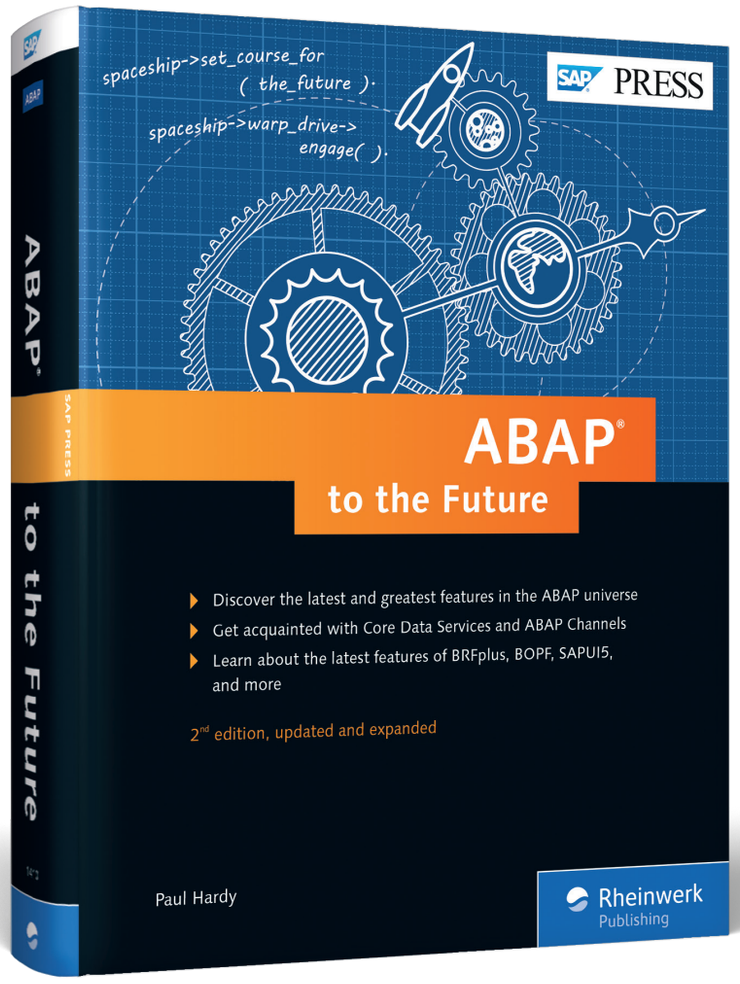
U

V

W

X

Z



Paul Hardy

ABAP to the Future

801 Pages, 2016, $79.95
ISBN 978-1-4932-1410-5

www.sap-press.com/4161

Paul Hardy joined Heidelberg Cement in the UK in 1990. For the first seven years, he worked as an accountant. In 1997, a global SAP rollout came along; he jumped on board and has never looked back since. He has worked on country-specific SAP implementations in the United Kingdom, Germany, Israel, and Australia.

After starting off as a business analyst configuring the good old IMG, Paul swiftly moved on to the wonderful world of ABAP programming. After the initial run of data conversion programs, ALV reports, interactive DYNPRO screens, and (urrggh) SAPscript forms, he yearned for something more and since then has been eagerly investigating each new technology as it comes out. Particular areas of interest in SAP are business workflow, B2B procurement (both point to point and SAP Ariba-based), logistics execution, and variant configuration, along with virtually anything new that comes along.

Paul can regularly be found blogging away on the SCN site and presenting at SAP conferences in Australia (Mastering SAP Technology and the SAP Australian User Group annual conference). If you happen to ever be at one of these conferences, Paul invites you to come and have a drink with him at the networking event in the evening and to ask him the most difficult questions you can think of (preferably SAP-related).