





Browse the Book

This chapter covers eliminating dependencies in your existing programs, implementing test doubles, and using ABAP Unit to write and implement test classes. It also includes advice on how to improve test-driven development via automated frameworks.

-  **ABAP Unit and Test-Driven Development**
-  **Table of Contents**
-  **Index**
-  **The Author**

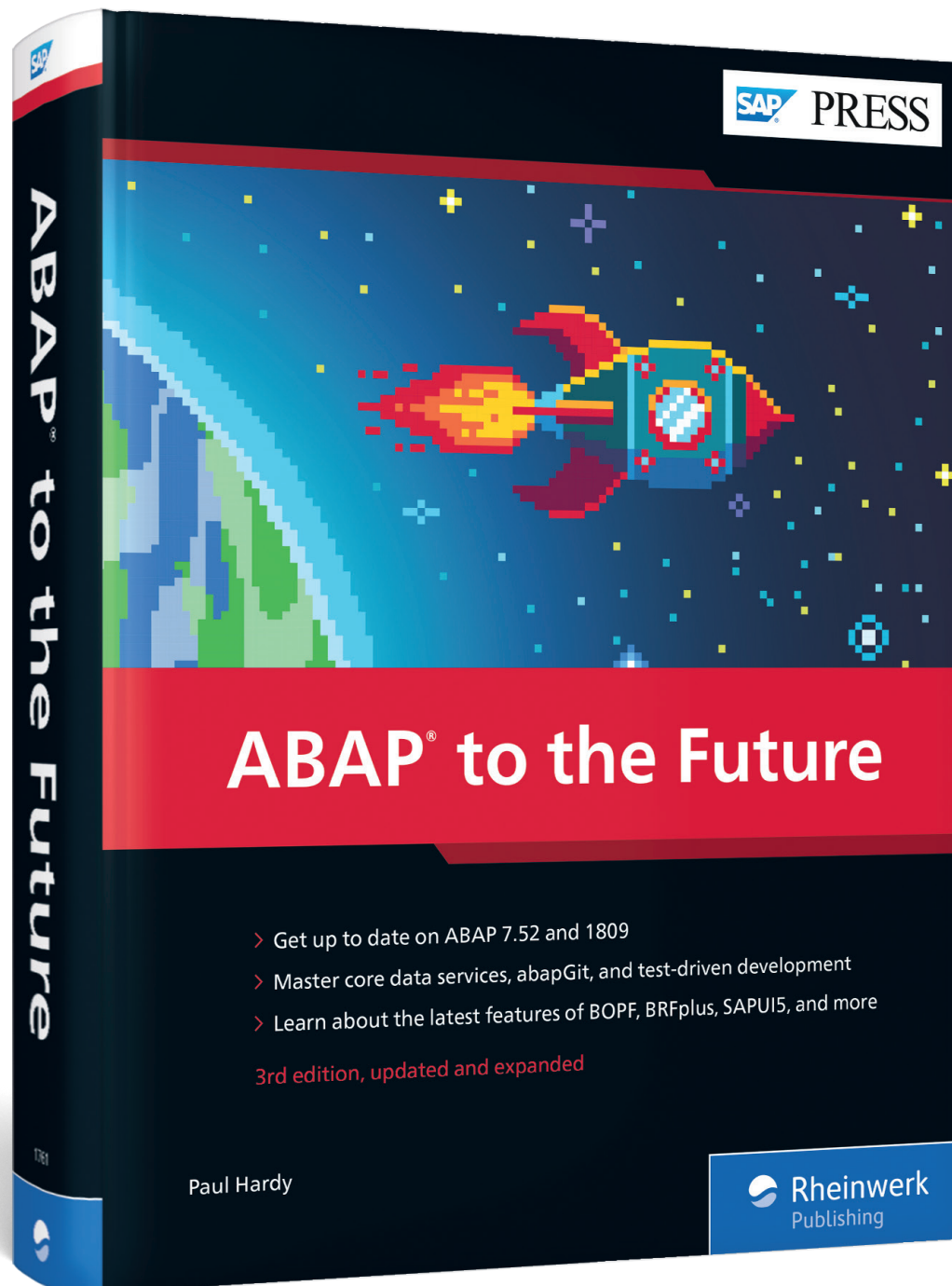
Paul Hardy

ABAP to the Future

864 Pages, 2019, \$79.95

ISBN 978-1-4932-1761-8

 www.sap-press.com/4751



Chapter 5

ABAP Unit and Test-Driven Development

“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*

Programs have traditionally been *fragile*; that is, any sort of change was likely to break something, potentially causing massive damage. You want to reverse this situation so that you can make as many changes as the business requires as fast as you can with zero risk. The way to do this is via test-driven development (TDD), supported by the ABAP Unit tool. 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. But if they can summon the courage to stay in the room and learn about TDD, then they’ll 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 5.1). This way, when the time comes to move the created or changed code into test, you can be confident that it's correct.

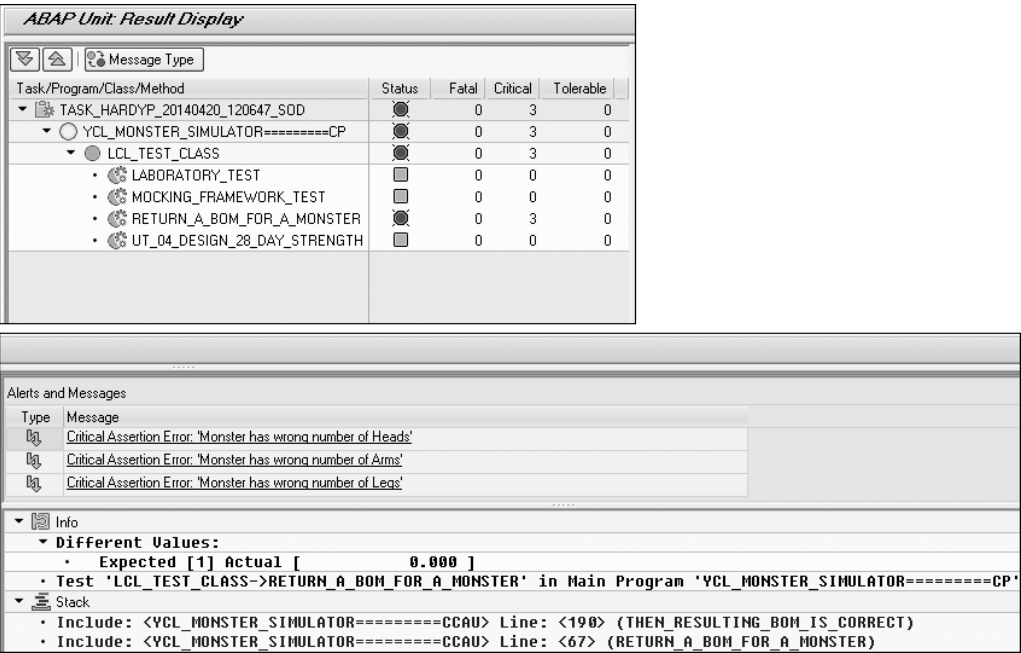


Figure 5.1 ABAP Unit Results Screen

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 (seams) in your existing programs
- 2. Implementing test doubles 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, which includes a look at how to run large numbers of unit tests on different data sets without having to hard-code them all.

Finally, after you've covered how to unit-test enable existing code, you'll see an explanation of why using TDD via the tools explained in this chapter on new code is such a good idea.

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'll 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're 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, 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.

5.1 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 can't or don't want to do in the development environment.

All Is Not as It Seams

Another term often used to describe a dependency is a *seam*. In both cases, the term means a place in your code that isn't testable due to a call to something outside the code to be tested. I personally consider dependency to be a more accurate term, but you need to be aware of the alternative.

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 won't 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.

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 test double 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).

5.1.1 Identifying Dependencies

Listing 5.1 shows an example of a common ABAP application. In this case, our good old friend the baron (remember him?) doesn't want any neighboring mad scientists building monsters and thus encroaching on his market, so as soon as he hears about such a competitor he drops a nuclear bomb on him (as any good mad scientist

would). Listing 5.1 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 5.1 Common ABAP Application

In this code, you want to test two things:

1. That you direct the missile to the correct target
2. That if the user aborts halfway through, the missile doesn't 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*. So long as they're part of your code, you can't implement automated unit tests.

Unfortunately, most ABAP code traditionally looks like the example in Listing 5.1; that is, the code is like a scrambled egg with all sorts of disparate things mixed together in the same routine. Thus, when preparing an existing program to be unit tested, the first thing to do is make a list of anything that isn't pure business logic (i.e., calls to the database, user input, calls to external systems, etc.), exactly as in the preceding bullet list. What you're doing is *unscrambling the egg*.

5.1.2 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 as of 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's 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 5.1.3 and doing some serious rearranging of the code.

Warning: Houston, We Have a Problem

The TEST-SEAM concept should never be used *ever*, on pain of death, because it's an abomination that horrifies all serious programmers (they would say production code should be unaware what parts of it are going to be tested) and was designed purely as an interim measure to use while redesigning old, badly written programs.

However, even that didn't make any sense because introducing a test seam means changing the production code, and if you're going to change the production code there's nothing that can't be wrapped in a class that's then replaced by a test double by the mechanism described in Section 5.2.

In the example in this section, we've moved the whole program into a function module so that 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 5.2).

```
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 5.2 Surrounding Dependencies with Test Seams

Highlighting the dependencies in this way not only allows unit testing (as we’ll cover in Section 5.3) 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’re the exact reverse.

5.1.3 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 doesn't 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 result will look like Listing 5.3.

```
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 5.3 Calling Methods of Classes

Note that the functionality hasn't 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 test double objects.

5.2 Implementing Test Doubles

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 *test double* 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. SAP likens them to stunt doubles or body doubles in movies. They look and act like the star but they're actually someone else.
- 2. Make sure that all the classes under test (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 test double objects instead of the real objects, but only when a test is underway. This is known as *injection*.

Test Double Terminology

When talking about test double objects, the terms *stub* and *mock* and *spy* and *fake* and *dummy* and *stalking horse* and *invasion of the body snatchers* and about a million others are often used. They're all variants of the same thing (you use an identical-looking duplicate for testing), and although all those terms are all in fact subtly different, trying to get your head around all the terminology just gives you a headache for no real benefit.

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

5.2.1 Test Injection for Test Seams

Test injection for test seams is the poor man's version of implementing test double objects. Instead of replacing entire routines with a duplicate inside a test double class, you flag sections (one or more lines of code) inside of such routines, as shown in Section 5.1.2, so that 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 5.4.

```
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 5.4 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.

5.2.2 Creating Test Doubles

For testing purposes, what you actually want is to define test double classes and test double objects. *Test double 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. Test double 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 test double monster object is an instance of a test double 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 local class that implements the interface of the real database access class. This local test double class 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 5.5, you'll see some possible redefined implementations of methods in test double subclasses that could replace the real classes in the example.

```
METHOD read_customising. "test double 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 test double database implementation

METHOD popup_to_confirm. "test double user interface implementation
*-----*
* RETURNING rd_answer TYPE char01
*-----*

  rd_answer = '1'."Yes
```



```
ENDMETHOD. "test double user interface implementation

METHOD fire_missile. "Test double External Interface Implementation

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

ENDMETHOD. "Fire Missile - Test double Ext Interface - Implementation
```

Listing 5.5 Test Double 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.

As of ABAP 7.51, the way you create test doubles for database access becomes a special case, as you'll see in Chapter 7 about SAP HANA programming. This is because SAP has developed a special framework for testing database access to test that your SQL statements are written correctly—something that wasn't possible previously.

Object-Oriented Recommendation

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 test double class be a local class that implements that interface. In the latter case, before ABAP 7.4 you'd have to create blank implementations for the methods you weren't 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.

Moreover, if your test double is a subclass and the real class gets a new method, then the test double won't know about it, leading to your unit tests using the actual production code for the new method and potentially accessing the real database or some such. If your test double isn't a subclass but implements an interface, then adding a new method to the interface doesn't cause such problems because the test double knows nothing about the implementation of the new method in the real class.

The implication is obvious: all classes should have their public methods/attributes defined via an interface because you never know when you might want to replace them with a test double.

5.2.3 Injection: Good Method

In well-designed programs, classes don't do everything themselves, but instead 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 5.6.

```
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 5.6 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 5.7.

```
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 5.7 Variables Set Up during Construction of Object Instance

However, as you can see, this design doesn't include any test double 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 test double 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 test double 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 5.8.

```
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 5.8 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 to read the database, write to a log, or

communicate with any other external party as needed. When running a unit test, you pass in (*inject*) test double objects to 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 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.

This technique works wonderfully, but the next section discusses an alternative method to achieve the same thing, which may in fact work even better.

5.2.4 Injection: Better Method

It has been argued that injection in any form is a lot of work and involves writing complicated constructor methods. I personally don't think it's such a big deal, but I do like an alternate method that was suggested by the SAP instructors in their open course about unit testing: *dependency lookup*.

Let's look at how dependency lookup works using the example of the monster-making machine (MMM) interface. In our unit tests, we don't actually want to connect to the monster-making machine and create actual monsters, so we need a test double.

We also don't want the production code to know about the existence of test doubles. The most elegant way to achieve this is to change the production code so that every time it wants a new instance of an object (that can be doubled), it doesn't do a CREATE OBJECT or rely on injection, but instead calls a factory method to get that instance.

Listing 5.9 compares the three ways of getting an instance—that is, direct creation via `CREATE OBJECT`, creation via some sort of injection mechanism, and creation via a factory class.

```
METHOD make_monster.  
* Three possible ways to generate an instance  
* (1) Direct Creation  
  CREATE OBJECT mo_monster_machine TYPE zcl_monster_making_machine.  
  
* (2) Parameter Injection  
  mo_monster_machine = io_machine.  
  
* (3) Instance Creation via Factory  
  mo_monster_machine = mo_factory->get_monster_machine( ).  
  
ENDMETHOD.
```

Listing 5.9 Three Different Ways to Get New Object Instance

Sometimes each class has its own factory method, but in this case we want one big factory method for the whole application that returns instances of all the objects we could need doubles for. In this example, we only have one method because we only have one object to create a test double for. There’s also the constructor to create a singleton for the object factory itself. SAP recommends making the whole thing static, but I think static things are evil, and quite frankly they smell, as you can’t override them (and the factory is production code, so someone someday may want to override it). The definition of the monster object factory is shown in Listing 5.10.

```
CLASS zcl_monster_object_factory DEFINITION  
  PUBLIC  
  CREATE PUBLIC .  
  
  PUBLIC SECTION.  
  
  METHODS get_monster_making_machine  
    RETURNING  
      VALUE(ro_monster_making_machine)  
        TYPE REF TO zif_monster_making_machine .  
  CLASS-METHODS get_instance  
    RETURNING
```

```
    VALUE(ro_factory) TYPE REF TO zcl_monster_object_factory .  
  PROTECTED SECTION.  
  PRIVATE SECTION.  
  
  DATA mo_monster_making_machine  
    TYPE REF TO zif_monster_making_machine .  
  CLASS-DATA mo_factory TYPE REF TO zcl_monster_object_factory .  
ENDCLASS.
```

Listing 5.10 Monster Object Factory Class Definition

The next stage is to create the monster-making machine class, which needs to be a “friend” of the factory class. This “friendship” means the factory can access the private attributes of the MMM class. This is needed because the next step is to change the MMM class so that its instance creation setting is private. This means that calling programs now can’t use a `CREATE OBJECT` statement to generate an instance of the class. They’re 100% forced to use the factory to get an instance. The factory is allowed to use a `CREATE OBJECT` statement to generate that instance because it’s a friend of the MMM class.

Why all this complexity? As you may have guessed, it’s so that the factory can return test doubles in tests and real instances in production, with the minimum needed changes to the production code to allow testing—calling a factory method instead of `CREATE OBJECT` replaces a line of code rather than adding extra ones as most other injection techniques do—and you want to keep your code as compact as possible.

So how does the factory know when to pass back a test double and when to pass back a real instance? The answer is that it doesn’t! The factory class is production code and production code should know nothing about testing, which is why `TEST-SEAMS` are so evil.

We do have to break this rule ever so slightly though. We’re going to create a special injector class, and once it exists, we’ll change the factory class so that the injector class is its friend. The injector class is rather like a cuckoo that throws out the eggs in another bird’s nest and replaces them with her own eggs. The mother bird raises the cuckoo chicks as her own and can’t tell the difference.

You can see the code for the injector class in Listing 5.11. The first and most vital thing is that the class is defined as `FOR TESTING` so that it can only run inside the unit test framework and most especially never in production. That way the factory class is safe from being meddled with in the live environment.

The injector class is called during the `SETUP` phase of a test method (more on this in Section 5.3, coming up next) and injects its egg (test double) into the factory method. Then when the factory method runs inside the unit test it passes back the test double; when it runs for real there's no injected double, so a real instance is passed back.

```
* <SIGNATURE>-----+
* | Instance Public Method ZCL_MONSTER_OBJECT_INJECTOR->INJECT_MONSTER_MAKING_
MACHINE
* +-----+
* | [-> IO_MONSTER_MAKING_MACHINE TYPE REF TO ZIF_MONSTER_MAKING_MACHINE
* +-----</SIGNATURE>
METHOD inject_monster_making_machine.

mo_factory->mo_monster_making_machine = io_monster_making_machine.

ENDMETHOD.
```

Listing 5.11 Monster Object Injector

When you look at the factory code (Listing 5.12), you can see the singleton pattern, which when used outside of unit testing means an instance of the class is created only once when the factory method is first called. But here this logic is interfered with in unit tests because the factory is fooled into thinking it's been called before and has already created an object instance.

Sometimes factory methods have really complicated logic for determining what exact subclass gets returned. In Listing 5.12, the exact subclass type for creation is hard-coded, but that doesn't need to be the case.

```
METHOD get_monster_making_machine.

IF mo_monster_making_machine IS NOT BOUND.
  "Logic could be really complicated here
  CREATE OBJECT mo_monster_making_machine
  TYPE zcl_monster_making_machine.
ENDIF.

ro_monster_making_machine = mo_monster_making_machine.

ENDMETHOD.
```

Listing 5.12 Monster Object Factory Returning Instance

Later, in Step 1 within Section 5.3.3, you'll see how during a unit test the injector class does what it says on the tin and injects test doubles during a run of the unit test framework, as shown in Listing 5.13. To use another horrible analogy from Mother Nature, this is like when a so-called zombie flea injects its eggs into a honey bee and takes over its brain.

```
"Injection
CREATE OBJECT lo_injector.
lo_injector->inject_pers_layer( mo_mock_pers_layer ).
lo_injector->inject_global_customising( mo_mock_customising ).
lo_injector->inject_monster_making_machine(mo_mock_mmm ).
lo_injector->inject_error_handler( mo_mock_error_handler ).
```

Listing 5.13 Dependency Lookup: Injection

Now we've got to the stage where we've changed the production code enough to inject test doubles in all the right places (seams), the production code is officially judged testable, and we can proceed with writing unit tests for it.

5.3 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 5.3.1.
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 5.3.2.

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're 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 isn't 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's only so much that you can write on the back of a sticky note.)

However, before we get down to the nitty-gritty of writing and implementing unit tests, let's introduce the concepts behind test-driven development (TDD).

5.3.1 Test-Driven Development

Most of this chapter focuses on the mechanics of the ABAP Unit framework—in other words, how to add unit tests to existing code, be it procedural or OO. When people first hear about this, they say it's a lot of effort to change existing code—and indeed it is, even if the effort pays you back a thousandfold.

However, what I really want to advocate is that when you write new code or make changes to existing code, the approach you should follow is to write the unit test *before* making the code change. That seems crazy at first glance: of course the test will fail because you're either testing a fix you haven't made yet or testing new developments for which you haven't even written the new production code!

But that's the entire point: you can't prove you've written code that does whatever it is you're trying to do unless you first prove the existing code base *doesn't do* whatever it is, and then you write code until you're 100% rock-solid sure that the problem has been solved—with tool support.

In normal development, you write code to solve the problem, and until you do a manual test of some sort you're guessing whether what you've done is correct. TDD takes the guesswork out of the equation.

TDD follows a circular pattern: the red stage then the green stage then the blue stage, then start again with the next test:

- **Red stage**
In the red stage, you have a problem to be solved and write a unit test to demonstrate that the current code doesn't do whatever's needed.
- **Green stage**
In the green stage, you write the minimum amount of code—and no more—to get the test to pass. Why the minimum? This is because developers tend to add all sorts of extra features that are never going to be used, per the YAGNI problem (You Ain't Gonna Need It).

- **Blue stage**
Once the test passes, you get to the blue stage, in which you're free to improve (refactor) your initial code as much as you want: first make it work, then make it good. You'll now be able to tell if your improvements break the fix as the test will start to fail.

I've done this in real life and I can tell you it works like a charm, even if the whole idea blows the minds of traditional ABAP developers.

I'm now going to make an even more bizarre claim: using TDD will make your programs better even if you never run the tests. That sounds even madder than before, right? What's the point of writing testable code if you aren't going to run the tests?

Well, obviously I do want you to run the tests, which enables regression testing, but the point is that testable code is good code. Writing code with tests in mind forces you to use the so-called SOLID principles of OO programming and stops you dead from cutting corners.

So I've put this challenge out on the Internet: The next time you have to change an existing program or create a new one, write the test first. You'll no doubt hate the idea initially, but I didn't like the look of pickled red cabbage at first, and now it's my favorite food. That might not be the best analogy, but I'm sure you get the idea.

Developers in other languages (e.g., Java, JavaScript, .NET) have been doing test-driven development from time immemorial, and SAP itself has adopted this approach since about 2009, which the company claims has sped them up dramatically.

It's been argued that the fact that 95% of global developers, including those at SAP, take this approach is no reason for "customer" ABAP developers to follow suit, as in, "If all those developers jumped off a cliff, would you do the same?"

This is a cliff I'm more than willing to jump off, and it's my goal in life to spread the message to as many SAP customer organizations as I can and get them all to jump off the TDD cliff like a swarm of lemmings.

5.3.2 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 5.14, 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 5.14 Defining Test Class

A lot of people say that testing private methods 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 5.15.

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

Listing 5.15 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’re 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 can’t 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 5.2.3 to take in a database layer and a logger as constructor parameters so that you can pass in test double objects that will pretend to handle interactions with the database and logging mechanism. Because you’re going to be passing in such test double objects to a constructor, you need to declare instance variables based on those test double classes (Listing 5.16).

```
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 5.16 Defining Test Double Classes for Injecting into Test Class

In Listing 5.8, you saw how in the constructor in production the class would create the real classes, but during a unit test test double classes are passed into the constructor of the class under test by the `SETUP` method, which runs at the start of each test method, or are injected into a factory class as you saw in Section 5.2.4.

The full list of the data definitions in the test class is shown in Listing 5.17. In addition to the test double 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 zvc_monster_input_data,  
      mt_bom_data        TYPE ztt_monster_items,  
      md_creator         TYPE zde_monster_creator_name.
```

Listing 5.17 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); 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 5.18). 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 5.18 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'll 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. This means the name of the test class should represent the `IT`. Thus the names tell a story: "the missile firer application should fire a nuclear missile" becomes `LTC_MISSILE_FIRER` for the test class name and, as we've said, `FIRE A NUCLEAR MISSILE` for one of the test methods.

In ABAP, you're 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 can 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 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 5.19.

```
*-----*
* Specifications for application LTC_MONSTER_CONTROLLER (IT)
*-----*
"IT SHOULD.....
"User Acceptance Tests
make_the_monster_sing FOR TESTING
make_the_monster_dance FOR TESTING
make_the_monster_go_to_france FOR TESTING
```

Listing 5.19 Test Methods that Describe What This 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's 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 also may 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 5.20.

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

```
"THEN.....  
then_resulting_bom_is_correct,
```

Listing 5.20 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.

5.3.3 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 5.21). Note that in the first method that starts with GIVEN_CUSTOMIZING, we’ve worked around the thirty-character limit by using parameters to make things look even more like plain English.

```
METHOD return_a_bom_for_a_monster.  
  given_customizing_that_says( for_evil_castle      = 'PL10'  
                              default_graveyard_is = '91' ).  
  
  given_monster_details_entered( ).  
  
  when_bom_is_calculated( ).
```

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

Listing 5.21 Implementation of Test Class

If you have parameters in your test methods, you can loop over an internal table of test inputs and outputs, testing a variety of test cases, and the code still is crystal clear as to what is being tested.

On introduction to unit testing, some developers get paralyzed and have no idea what it is they should be testing for. To decide in what order to convert the requirements in the specification to test classes, ask yourself this question: What’s the most important thing the program should be doing but doesn’t yet do? That important thing will be composed of several smaller steps, so start writing tests for them. Then move on to the next most important thing, and so on.

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, you’ll find out how to reduce the amount of code needed to do this.

Because 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 5.22.

```
METHOD: setup.  
*-----*  
* Called before every test  
*-----*  
  
CREATE OBJECT mo_class_under_test.  
"Create Test Doubles
```

```
CREATE OBJECT mo_mock_monster_pers_layer.
CREATE OBJECT mo_mock_logger.
CREATE OBJECT mo_mock_monster_making_machine
"Injection
lo_injector = NEW zcl_monster_object_injector( ).
lo_injector->inject_pers_layer( mo_mock_monster_pers_layer ).
lo_injector->inject_logger( mo_mock_logger ).
lo_injector->inject_mmm( mo_mock_monster_making_machine ).
"Clear Member Variables
CLEAR: ms_input_data,
      md_creator.

ENDMETHOD. "setup
```

Listing 5.22 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 5.23). 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 many 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 5.23 Preparing Test Data by Simulating External Input

Step 3: Calling the Production Code to be Tested

The time has come to 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 test double 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 5.24).

```
"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 5.24 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're correct or not. There generally are two types of tests:

1. Absolutely basic tests

In Chapter 4, Section 4.3, you 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 framework when the standard mechanisms don't do everything you want. Finally, you'll see how to achieve 100% test coverage.

Testing for Really Basic Failures

In Chapter 4, you saw 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 5.25 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 5.25 Unit Test to Check for Basic Errors

The structure in Listing 5.25 is totally generic. In TDD, you create the structure before writing the actual code, so the nature of the pre- and postconditions only will 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 5.26—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 5.26).

```
"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 5.26 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're 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 5.26, 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 ASSERT_THAT assertion in action. First, create a local class that implements the interface needed when using the ASSERT_THAT method (Listing 5.27).

```
CLASS lcl_my_constraint DEFINITION.

PUBLIC SECTION.
    INTERFACES if_constraint.

ENDCLASS.

"lcl_my_constraint DEFINITION
```

Listing 5.27 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 5.28 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 5.28 Implementation of Custom Constraint Class

All that remains is to call the assertion in the THEN part of a test method, as shown in Listing 5.29.

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

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

Listing 5.29 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 ten 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.

5.4 Optimizing the Test Process

In the example presented in this chapter, the code was deliberately simple 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 in the previous sections, you’ll see that this can lead to quite a large effort in writing the test code. This is what puts people off unit testing and TDD in general: they perceive this extra effort as a showstopper. My position is that even if there were no tools to make things easier for you, you should still

do TDD anyway because the benefits are so huge. Luckily, there are tools to help, and in great abundance.

In this section, you'll see that both the Eclipse IDE and the ABAP language in general bend over backward to help you write unit tests. Then you'll see that in addition there's a framework to automate creating test double objects, the aptly named ABAP Test Double Framework.

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, and how to reduce that number of test situations while still covering everything important.

5.4.1 Eclipse Support for the Unit Test Process

One thing will become very clear to you once you start writing unit tests and following the TDD methodology: you'll find that the process is a million times easier if you develop using ABAP in Eclipse instead of SE80. In this section, we'll talk about how Eclipse automates what are labor-intensive manual steps in SE80 and touch on the concept of "continuous integration" and how it's applied in the ABAP world in general and Eclipse in particular.

Autogeneration of Method Definitions/Implementations

In SE80, there's a tool to create test classes based on existing routines, but that's the wrong way around. With TDD, the idea is to write the test before the production code. So you create your test class first, and it contains a reference to the class under test. Then you write some test methods in which the `GIVEN` section contains a call to a method that doesn't yet exist in the class under test. Naturally, the test fails; it can't even compile.

So you click the nonexistent method and press `CTRL+1`. You're asked if you want to create a blank definition and implementation in the class under test. Yes you do—and half a second later the new method exists. If you have parameters, they're even typed correctly (most of the time). You just can't do that in SE80, and moreover this is the right way round: create the test first and then the production code based upon the test.

Autoupdate of Test Doubles with Interfaces

As you saw in Section 5.2, the best way to code test doubles is to have them implement the same interface as the real classes they're standing in for. You can't use

PARTIALLY IMPLEMENTED on interface declarations outside of test classes, so each time the interface gets a new method added your test double class will stop compiling properly, producing a warning that the new method hasn't been implemented in the double. In SE80, you would have to add each such implementation by hand, even if each implementation was blank (it's quite common to have lots of blank implementation methods in a test double), just to get the code working again.

In Eclipse, you just put your cursor on the name of the interface and press `CTRL+1`, and you're asked if you want to generate empty implementations for all the missing methods. Once again, you say yes—and half a second later the problem has gone away.

Autorun of ABAP Unit Runner (Continuous Integration)

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 accidentally breaks something else and doesn't bother to run the unit tests and thus doesn't realize that he's broken something?

Continuous integration is a concept that comes from the non-ABAP world, in which developers code on their local machines and then deploy their changes to the main code base. At that point, there's usually some sort of automated process in which tests run to prove the change hasn't broken anything—and if the tests fail, then the change is rejected. You learned about this back in Chapter 2 when we talked about `abapGit`, and in Chapter 1 when you installed the Continuous Integration plug-in for Eclipse.

The ABAP equivalent is report `RS_AUCV_RUNNER`, which is started via the ABAP Test Cockpit (Transaction `ATC`). This can run large amounts of unit tests and should be scheduled to run on a regular basis; emails can be sent automatically in the event of test failure. It's the job of all the ABAP developers to keep all the tests green. There are two options:

- 1. Inside Eclipse, ABAP Unit Runner (`CTRL+SHIFT+F10`) displays the test methods and shows which tests have failed.
- 2. Likewise, ABAP Coverage View (`CTRL+SHIFT+F11`) displays the statistics on how much of the code has been covered by the tests.

Both of those options are available in SE80; they just look a lot better in Eclipse.

5.4.2 ABAP Support for the Unit Test Process

Some of the newer features in ABAP make unit testing just that little bit easier. Specifically, let’s go over how to create test double classes when the original class implemented an interface, how to code the contents of a test method faster, and how to create complicated test double objects with fewer lines of code.

Creating Test Doubles

As you’ll recall from Chapter 3, interface methods can now be made optional in ABAP, so you don’t have to redefine them all. The same type of change has been made to the definition and implementation of test classes, but here the syntax is a little different. Prior to 7.4, you had to create an empty implementation for every method defined in the interface, even if you were never going to call that method at any point in your test class. An example of this is shown in Listing 5.30.

```
CLASS lcl_mock_monster DEFINITION FOR TESTING.  
  PUBLIC SECTION.  
    INTERFACES if_really_big_standard_interface.  
ENDCLASS.  
  
CLASS lcl_mock_monster IMPLEMENTATION.  
  METHOD one_i_want_to_use.  
  ENDMETHOD.  
  METHOD one_i_do_not_want.  
  ENDMETHOD.  
  METHOD another_i_do_not_want.  
  ENDMETHOD.  
  Etc.
```

Listing 5.30 Defining and Implementing Interface in Test Class before 7.4

From 7.4 on, however, you can write something like Listing 5.31 in your test class. Now you only have to create implementations of the methods the real class actually uses—that is, not all the blank ones. The PARTIALLY IMPLEMENTED statement only works inside test classes, which is as it should be. In real life, an interface is a guarantee to the outside world that a class can provide all the functionality in that interface.

```
CLASS lcl_mock_monster DEFINITION FOR TESTING.  
  PUBLIC SECTION.  
    INTERFACES if_really_big_standard_interface
```

```
  PARTIALLY IMPLEMENTED.  
ENDCLASS.  
CLASS lcl_mock_monster IMPLEMENTATION.  
  METHOD one_i_want_to_use.  
  ENDMETHOD.  
ENDCLASS.
```

Listing 5.31 Defining and Implementing Interface in Test Class in 7.4

Coding Return Values from Test Doubles

Usually, when performing unit tests, you have some test doubles (test double objects) that return hard-coded values based upon other hard-coded values. Prior to ABAP 7.4, you would end up with something that looks like the code in Listing 5.32.

```
METHOD get_monster_chemical_dose.  
  
  IF id_chemical_group = 'TRANSYLVANIA' AND  
    id_monster_strength = 35 AND  
    id_chemical_type = 'SNAILS'.  
    rd_chemical_dosage = '350'.  
  ENDIF.  
  
  IF id_chemcial_type = 'PUPPY_DOG_TAILS'.  
    rd_chemical_dosage = 80.  
  ENDIF.  
  
ENDMETHOD.
```

Listing 5.32 Test Double Objects

However, by using the COND construct you learned about in Chapter 3, such code can be simplified to the code shown in Listing 5.33. For your return value, instead of a never-ending string of IF/ELSE statements specifying the RD_CHEMICAL_DOSAGE variable inside each branch, you have a slightly more compact structure.

```
METHOD get_monster_chemical_dose.  
  
  rd_chemical_dosage = COND zde_chemical_dosage(  
    WHEN id_chemical_group = 'TRANSYLVANIA' AND  
      id_monster_strength = 35      AND
```

```
id_chemical_type = 'SNAILS'.
THEN '350'
WHEN id_chemcial_type = 'PUPPY_DOG_TAILS' THEN 80 ).

ENDMETHOD.
```

Listing 5.33 COND Method

Because such methods tend to have long strings of hard-coded logic like this, even such small reductions in code for each IF branch can all add up.

Creating Test Doubles Related to Complex Objects

In the sections earlier in this chapter, you read about injection in the context of unit testing. *Injection* makes it easier to set up objects that need to have other objects passed into them upon creation (and some of those objects need other objects passed into them upon creation, and so on). A pre-7.4 example of such an object setup is shown in Listing 5.34.

```
DATA: model      TYPE REF TO zcl_monster_model,
      view       TYPE REF TO lcl_view,
      controller TYPE REF TO lcl_controller,
      logger     TYPE REF TO zcl_bc_logger,
      pers_layer TYPE REF TO lcl_mock_pers_layer.

CREATE OBJECT logger.
CREATE OBJECT mock_pers_layer.
CREATE OBJECT model
EXPORTING
  io_logger = logger
  io_pers_layer = mock_pers_layer
CREATE OBJECT view.
CREATE OBJECT controller
EXPORTING
  io_model = model
  io_view = view.
```

Listing 5.34 Building Up Complex Object before 7.4

Well-designed OO programs need a lot of small classes that work together to make them more resistant to change—but what a lot of lines of code you need to create an

instance of your application controller! If you’ll forgive the pun, with release 7.4 of ABAP you can replace all this with one *monster* line of code, as shown in Listing 5.35. The CREATE OBJECT statements can be replaced by the NEW constructor operator each time you want to pass in a parameter to an object’s constructor—and you no longer need data declarations because the type of the newly created object is taken from the definition of the importing parameter of the constructor.

```
controller = NEW lcl_controller(
model      = NEW zcl_monster_model(
logger     = NEW zcl_logger( )
pers_layer = NEW lcl_mock_pers_layer( ) )
view       = NEW lcl_view ( )).
```

Listing 5.35 Building Up Complex Object in 7.4

This time, the new process knocked out almost two out of every three lines of code: the data declarations and the CREATE OBJECT statements.

5.4.3 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. Test double objects are a great example of this: many different test double 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 test double 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 to say that it 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.

Why Interfaces Are Better than Subclasses during Testing

When your test double class is a subclass of the dependency to be mocked, then you redefine all the methods called by the production code to return hard-coded values. If you have your test double class not be a subclass but instead implement an interface, it appears you do the exact same thing—that is, define the methods as returning hard-coded values.

The difference is that when the real class that’s being doubled adds a new method, the test double that is a subclass would use the real production code and thus potentially break the tests, but a double with an interface could never use the new production code in a million years. It would give you a syntax error—but at least that’s in your face.

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 test double. 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 5.36. In our example, the class to be test doubled is `ZCL_MONSTER_SIMULATOR`, which implements interface `ZIF_MONSTER_SIMULATOR`. Listing 5.36 demonstrates several concepts; let’s examine them one at a time before looking at the listing as a whole.

First you create the test double 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 test doubled is `CALCULATE_SCARINESS`. This may seem odd, but the method name isn’t 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. The fact that you specify the result before saying what method gives that result is a Bad Thing and highly illogical, Captain, but that’s the way the framework works.

Next you set up the input data. As mentioned earlier, in Listing 5.20 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 test double 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 test double 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 5.36 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 test doubling 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 5.36 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 test double. 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 so 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 test double method doesn't do anything at all in a test situation, you still want to be sure that it's been called.

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's 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 5.37.

```
METHOD test standing in for_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 test standing in for 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."Test doubleing Exception Test
```

Listing 5.37 Test Doubling Exception Using ATDF

Note that to test double an exception being raised, the exception being tested for has to be declared in the signature of the method being test doubled. 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 the URL of 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 test double 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 test double class that implements an interface—which is the better way to go. But sometimes this is even more work because in earlier versions of ABAP you needed an implementation for every method in the interface.

5.4.4 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, 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 hunch-back 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 named 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 URL 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 Code Exchange section of the SAP Community 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, then update the open-source project so that the whole SAP community benefits.

As mentioned in Chapter 2, all these open source ABAP projects—and this one in particular—now live on GitHub and can be installed using abapGit. This project in particular seems to be getting new features on a regular basis, at least in 2018.

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 5.38 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 SMWO, 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 5.38 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 SMWO  
  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 5.38 Test Method to Load Multiple Test Cases

Every so often, a new problem will arise in production; you just need to 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 test double 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 test double uploader.

As Snoopy would say, “You see how it all comes together?”

5.4.5 Combinatorial Test Design

In the last section, we talked about how to handle massive amounts of data, situations in which you have massive amounts of test cases and each line of your test data spreadsheet is a different situation that needs its own unit test.

The situation in which you have massive amounts of test cases arises usually when you have a large number of input parameters, each with many possible values, and before you know it your ten input parameters are generating 10 x 5 x 2 x 2 x 2 x 3 different possible combinations of input data—so to exhaustively test every one, you need 1,200 different test cases.

In fact, you don’t need anywhere near that number because there are two ways to reduce the number of needed test cases without sacrificing the accuracy of the test cover. In the Recommended Reading at the end of the chapter you’ll see a URL for a series of YouTube videos about *combinatorial test design* (does a video count as reading?).

The videos are very comprehensive, but here I’ll give my own summary of the basic idea, plus of course I’ve written my own ABAP program to help in this area. First, one input field might be a number, and naturally there could be an infinite number of values a user can enter, so you can’t test them all. You have to restrict the test cases to the following:

- A negative number
- Zero
- A positive number
- The user typing in “HELLO” or some other character value instead of a number
- Boundary numbers

Straight away, you’re asking, “What in the world is a boundary number?” If you’re testing for ages between fifteen and sixty, say, you test with *boundary numbers* like fourteen, sixteen, fifty-nine, or sixty-one to see what happens. This is because a very common error is using GE (>=) instead of GT (>) or vice versa in comparison conditions.

On its own, that technique dramatically reduces the number of possible test cases, and then you can use the *all pairs technique* to reduce the number still further. At this point you’re wondering what “all pairs” is and maybe thinking it involves a teenager

from a different country coming to your house and looking after your children—but no, that’s an *au pair*, which is something different.

When it comes to “all pairs,” research has shown that the vast bulk of errors in code come from a specific combination of two input values. Therefore, the idea is that you only need to test each pair of input values. One Recommended Reading article at the end of the chapter points you to a multitude of tools with which you can input all your input parameters and their possible values and get a list of test cases.

Naturally, I had to write the same tool in ABAP—using test-driven development—and you can download program ZALL_PAIRS from the abapGit repository for this book (as well as the book’s webpage at www.sap-press.com/4751).

The result is that you have an automated tool to produce the large amount of test cases you can use to help in your unit testing via a framework like that described in Section 5.4.4.

5.5 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 the TDD 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.

Framework Specific Test Tools

Here you have learned about how to do TDD/unit testing for ABAP code in general. There is a bucket load of framework-specific unit test tools to extend this concept for specific tools, such as CDS views, BOPF, WDA, and so on. These will be addressed in the relevant chapters.

Once you’ve started doing TDD for real, you’ll find the cycle goes something like this: write the failing test, write the production code, the test still fails, so you need to debug the production code.

You want to spend as little time in the debugger as possible by jumping straight to the cause of your problem. This brings us nicely to the next chapter, which is all about the wonderful enhancements SAP has made to the debugger in recent years—in particular, debugger scripting.

Recommended Reading

- **Head First Design Patterns**
Eric Freeman, Elisabeth Robson, Bert Bates, and 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**
<https://blogs.sap.com/2013/08/28/dependency-injection-for-abap/> (Jack Stewart)
- **ABAP Test Double Framework**
<https://blogs.sap.com/2015/01/05/abap-test-double-framework-an-introduction/> (Parjul Meyana)
- **ZMOCKUP_LOADER**
https://blogs.sap.com/2018/09/02/zmockup_loader-and-unit-tests-with-interface-stubbing/ (Alexander Tsybulsky)
- **Combinatorial Test Design**
https://www.youtube.com/playlist?list=PLQeCIR5qhUI_jkEnQV3LQOoY73SbySWyA

Contents

Acknowledgments	21
Introduction	23

1

ABAP in Eclipse

35

1.1	Installation	37
1.1.1	Installing Eclipse	38
1.1.2	Installing SAP-Specific Add-Ons	40
1.1.3	Connecting Eclipse to a Backend SAP System	42
1.2	ABAP-Specific Features	43
1.2.1	Working on Multiple Objects at the Same Time	48
1.2.2	Bookmarking	50
1.2.3	Creating a Method from the Calling Code	51
1.2.4	Extracting a Method	56
1.2.5	Refactoring: Moving Methods and Attributes	61
1.2.6	Deleting Unused Variables	62
1.2.7	Creating Instance Attributes and Method Parameters	63
1.2.8	Creating Class Constructors	63
1.2.9	ABAP 7.5+ Features	65
1.3	Eclipse-Specific Features	73
1.3.1	Neon (2016)	76
1.3.2	Oxygen (2017)	78
1.3.3	Photon (2018)	78
1.4	Testing and Troubleshooting	79
1.4.1	Unit Testing Code Coverage	79
1.4.2	Debugging	83
1.4.3	Runtime Analysis	86
1.4.4	Dynamic Log Points	87
1.5	Customization Options with User-Defined Plug-Ins	89
1.5.1	Favorites List	90
1.5.2	Automatic Pretty Print	92
1.5.3	System Coloring	94

1.6	Eclipse in the Cloud	95
1.7	Summary	96
2	abapGit	97
2.1	Theory	98
2.2	Installation	99
2.2.1	Installing the abapGit Repository in Your SAP System	99
2.2.2	Keeping Your abapGit Version Up to Date	105
2.2.3	Watching the abapGit Repository	107
2.3	Storing and Moving Objects	108
2.3.1	abapGit versus SAPlink	108
2.3.2	Using Online Repositories	109
2.3.3	Using Offline Repositories	115
2.4	Branching	120
2.4.1	Project Collaboration	121
2.4.2	Production Support	127
2.4.3	Utopian Dream	135
2.5	Summary	136
3	New Language Features in ABAP	137
3.1	Declaring and Creating Variables	138
3.1.1	Omitting Data Type Declarations	139
3.1.2	Creating Objects Using NEW	140
3.1.3	Filling Structures and Internal Tables While Creating Them Using VALUE	140
3.1.4	Filling Internal Tables from Other Tables Using FOR	142
3.1.5	Creating Short-Lived Variables Using LET	143
3.1.6	Enumerations	144
3.1.7	New Mathematical Operators	147
3.2	String Processing	148

3.3	Calling Functions	149
3.3.1	Avoiding Type Mismatch Dumps when Calling Functions	149
3.3.2	Using Constructor Operators to Convert Strings	151
3.3.3	Functions that Expect TYPE REF TO DATA	152
3.4	Conditional Logic	153
3.4.1	Omitting ABAP_TRUE	153
3.4.2	Using XSDBOOL as a Workaround for BOOLC	155
3.4.3	The SWITCH Statement as a Replacement for CASE	157
3.4.4	The COND Statement as a Replacement for IF/ELSE	158
3.5	Internal Tables	160
3.5.1	Table Work Areas	160
3.5.2	Reading from a Table	162
3.5.3	CORRESPONDING for Normal Internal Tables	164
3.5.4	MOVE-CORRESPONDING for Internal Tables with Deep Structures	165
3.5.5	Dynamic MOVE-CORRESPONDING	169
3.5.6	New Functions for Common Internal Table Tasks	171
3.5.7	Internal Table Queries with REDUCE	174
3.5.8	Grouping Internal Tables	175
3.5.9	Extracting One Table from Another	178
3.5.10	Virtual Sorting of Internal Tables	180
3.6	Object-Oriented Programming	182
3.6.1	Upcasting/Downcasting with CAST	182
3.6.2	Finding the Subclass of an Object Instance	183
3.6.3	CHANGING and EXPORTING Parameters	185
3.6.4	Changes to Interfaces	186
3.7	Search Helps	187
3.7.1	Predictive Search Helps	188
3.7.2	Search Help in SE80	189
3.8	Summary	189
4	Exception Classes and Design by Contract	191
4.1	Types of Exception Classes	193
4.1.1	Static Check (Local or Nearby Handling)	194

4.1.2	Dynamic Check (Local or Nearby Handling)	195
4.1.3	No Check (Remote Handling)	196
4.1.4	Deciding Which Type of Exception Class to Use	198
4.2	Designing Exception Classes	199
4.2.1	Creating the Exception	199
4.2.2	Declaring the Exception	201
4.2.3	Raising the Exception	202
4.2.4	Cleaning Up after the Exception Is Raised	207
4.2.5	Error Handling with RETRY and RESUME	210
4.3	Design by Contract	214
4.3.1	Preconditions and Postconditions	216
4.3.2	Class Invariants	218
4.3.3	Handling DBC Violations	221
4.4	Summary	222

5 ABAP Unit and Test-Driven Development 223

5.1	Eliminating Dependencies	225
5.1.1	Identifying Dependencies	226
5.1.2	Breaking Up Dependencies Using Test Seams	228
5.1.3	Breaking Up Dependencies Properly	231
5.2	Implementing Test Doubles	233
5.2.1	Test Injection for Test Seams	234
5.2.2	Creating Test Doubles	235
5.2.3	Injection: Good Method	237
5.2.4	Injection: Better Method	239
5.3	Writing and Implementing Unit Tests	243
5.3.1	Test-Driven Development	244
5.3.2	Defining Test Classes	245
5.3.3	Implementing Test Classes	252
5.4	Optimizing the Test Process	261
5.4.1	Eclipse Support for the Unit Test Process	262
5.4.2	ABAP Support for the Unit Test Process	264
5.4.3	Test Double Framework	267

5.4.4	Unit Tests with Massive Amounts of Data	272
5.4.5	Combinatorial Test Design	277
5.5	Summary	278

6 Debugger Scripting 281

6.1	Writing a Debugger Script Program	282
6.2	Coding the SCRIPT Method	287
6.3	Coding the INIT and END Methods	293
6.4	Real-World Examples	300
6.5	Summary	302

7 Database Programming with SAP HANA 305

7.1	The Three Faces of Code Pushdown	306
7.2	ABAP SQL	307
7.2.1	New Commands in ABAP SQL	308
7.2.2	Creating while Reading	314
7.2.3	Buffering Improvements	316
7.2.4	INNER JOIN Improvements	318
7.2.5	UNION	320
7.2.6	CROSS JOINS	320
7.2.7	Code Completion in SELECT Statements	322
7.2.8	Filling a Database Table with Summarized Data	322
7.2.9	Common Table Expressions	323
7.2.10	Stricter Syntax Check	324
7.2.11	Unit Testing ABAP SQL Statements	324
7.3	CDS Views	327
7.3.1	Creating a CDS View in Eclipse	329
7.3.2	Coding a CDS View in Eclipse	332
7.3.3	Classifying a CDS View	345
7.3.4	Adding Authority Checks to a CDS View	347

7.3.5	Reading a CDS View from an ABAP Program	348
7.3.6	Unit Testing CDS Views	351
7.4	ABAP Managed Database Procedures	354
7.4.1	Defining an AMDP in Eclipse	354
7.4.2	Implementing an ADMP in Eclipse	355
7.4.3	Calling an AMDP from an ABAP Program	360
7.4.4	Calling an AMDP from Inside a CDS View	360
7.5	Locating and Pushing Down Code	364
7.5.1	Finding Custom Code that Needs to Be Pushed Down	364
7.5.2	Which Technique to Use to Push Code Down	366
7.5.3	Example	369
7.6	Summary	375
8	Business Object Processing Framework	377
8.1	Manually Defining a Business Object	379
8.1.1	Creating the Object	380
8.1.2	Creating a Header Node	382
8.1.3	Creating an Item Node	384
8.2	Generating a Business Object from a CDS View	386
8.3	Using BOPF to Write a Dynpro-Style Program	389
8.3.1	Creating Model Classes	390
8.3.2	Creating or Changing Objects	394
8.3.3	Locking Objects	407
8.3.4	Performing Authority Checks	409
8.3.5	Setting Display Text Using Determinations	410
8.3.6	Disabling Specific Commands Using Validations	424
8.3.7	Checking Data Integrity Using Validations	426
8.3.8	Responding to User Input via Actions	433
8.3.9	Saving to the Database	445
8.3.10	Tracking Changes in BOPF Objects	452
8.4	Unit Testing BOPF Objects with BUnit	461
8.5	Custom Enhancements	464
8.5.1	Enhancing Standard SAP Objects	465

8.5.2	Using a Custom Interface (Wrapper)	466
8.6	Summary	468
9	BRFplus	469
9.1	The Historic Location of Rules	472
9.1.1	Rules in People's Heads	472
9.1.2	Rules in Customizing Tables	474
9.1.3	Rules in ABAP	476
9.2	Creating Rules in BRFplus: Basic Example	477
9.2.1	Creating a BRFplus Application	477
9.2.2	Adding Rule Logic	486
9.2.3	BRFplus Rules in ABAP	498
9.3	Creating Rules in BRFplus: Complex Example	502
9.4	Simulations	509
9.5	Unit Testing	511
9.6	SAP Business Workflow Integration	512
9.7	Options for Enhancements	516
9.7.1	Procedure Expressions	517
9.7.2	Application Exits	517
9.7.3	Custom Frontends	518
9.7.4	Custom Extensions	518
9.7.5	Traces and Versions	519
9.8	Rules Frameworks	519
9.9	Summary	520
10	ALV SALV Reporting Framework	523
10.1	Getting Started	526
10.1.1	Defining an SALV-Specific (Concrete) Class	527
10.1.2	Coding a Program to Call a Report	528

10.2 Designing a Report Interface	531
10.2.1 Report Flow Step 1: Creating a Container (Generic/Optional)	534
10.2.2 Report Flow Step 2: Initializing a Report (Generic)	534
10.2.3 Report Flow Step 3: Making Application-Specific Changes (Specific)	542
10.2.4 Report Flow Step 4: Displaying the Report (Generic)	556
10.3 Adding Custom Command Icons with Programming	561
10.3.1 Creating a Method to Automatically Create a Container	563
10.3.2 Changing ZCL_BC_VIEW_SALV_TABLE to Fill the Container	564
10.3.3 Changing the INITIALIZE Method	565
10.3.4 Adding the Custom Commands to the Toolbar	566
10.3.5 Sending User Commands from the Calling Program	567
10.3.6 Adding Separators	568
10.4 Editing Data	569
10.4.1 Creating a Custom Class to Hold the Standard SALV Model Class	571
10.4.2 Changing the Initialization Method of ZCL_BC_VIEW_SALV_TABLE	571
10.4.3 Adding a Method to Retrieve the Underlying Grid Object	576
10.4.4 Changing the Calling Program	579
10.4.5 Coding User Command Handling	579
10.5 Handling Large Internal Tables with CL_SALV_GUI_TABLE_IDA	583
10.6 Open-Source Fast ALV Grid Object	586
10.7 Making SAP GUI Look Like SAP Fiori	587
10.8 Summary	588

11 Web Dynpro ABAP and Floorplan Manager 589

11.1 The Model-View-Controller Concept	590
11.1.1 Model	591
11.1.2 View	593
11.1.3 Controller	596
11.2 Building the WDA Application	597
11.2.1 Creating a Web Dynpro Component	599
11.2.2 Declaring Data Structures for the Controller	600
11.2.3 Establishing View Settings	604

11.2.4 Defining the Windows	612
11.2.5 Navigating between Views inside the Window	614
11.2.6 Enabling the Application to be Called	616
11.3 Coding the WDA Application	617
11.3.1 Linking the Controller to the Model	618
11.3.2 Selecting Monster Records	618
11.3.3 Navigating to the Single-Record View	624
11.4 Using Floorplan Manager to Create WDA Applications	628
11.4.1 Creating an Application Using Floorplan Manager	630
11.4.2 Integrating BOPF with Floorplan Manager	643
11.5 Unit Testing WDA Applications	651
11.6 Making WDA Look Like SAP Fiori	654
11.7 Summary	655

12 SAPUI5 657

12.1 Architecture	659
12.1.1 Frontend: What SAPUI5 Is	660
12.1.2 Backend: What SAP Gateway Is	661
12.2 Prerequisites	661
12.2.1 Requirements in SAP	662
12.2.2 Requirements on Your Local Machine	662
12.3 Backend Tasks: Creating the Model Manually Using SAP Gateway	663
12.3.1 Configuration	663
12.3.2 Coding	678
12.4 Backend Tasks: Automatically Generating the Model	691
12.4.1 Creating an SAP Gateway Service by Pulling from a CDS View	691
12.4.2 Creating an SAP Gateway Service by Pushing from a CDS View	694
12.5 Frontend Tasks: Creating the View and Controller Using SAPUI5	699
12.5.1 First Steps	699
12.5.2 View	704
12.5.3 Controller	718
12.5.4 Testing Your Application	724

12.6	Generating SAPUI5 Applications from SAP Web IDE Templates	726
12.7	Generating SAPUI5 Applications from SAP Build	733
12.8	Adding Elements with OpenUI5	743
12.9	Importing SAPUI5 Applications to SAP ERP	748
12.9.1	Storing the Application inside SAP	748
12.9.2	Testing the SAPUI5 Application from within SAP ERP	750
12.10	Unit Testing SAPUI5 Applications	753
12.10.1	ESLint	753
12.10.2	JUnit	754
12.10.3	OPA	755
12.10.4	Gherkin	756
12.11	SAPUI5 vs. SAP Fiori	757
12.12	Summary	759
 13 ABAP Channels		761
13.1	General Concepts	762
13.1.1	ABAP Messaging Channels	763
13.1.2	ABAP Push Channels	764
13.1.3	ABAP Daemons	765
13.2	ABAP Messaging Channels: SAP GUI Example	767
13.2.1	Coding the Sending Application	770
13.2.2	Coding the Receiving Application	775
13.2.3	Watching the Applications Communicate	779
13.3	ABAP Push Channels: SAPUI5 Example	782
13.3.1	Coding the Receiving (Backend) Components	783
13.3.2	Coding the Sending (Frontend) Application	792
13.4	Internet of Things Relevance	793
13.5	Summary	794

14 The RESTful ABAP Programming Model		797
14.1	ABAP in the Cloud	798
14.2	Programming Model Changes	802
14.2.1	Changes to Database Layer	802
14.2.2	Changes to Business Logic Layer	803
14.2.3	Changes to User Interface Layer	805
14.3	Application Structure	806
14.4	Coding a Transactional Business Object Application	807
14.4.1	Coding Business Object CDS Views	807
14.4.2	Coding Behavior Definition	811
14.4.3	Coding Behavior Implementation	813
14.4.4	Saving the Changed Business Objects	837
14.5	Service Definitions and Bindings	839
14.5.1	Creating a Service Definition	839
14.5.2	Creating a Service Binding	840
14.5.3	Creating an SAPUI5 App to Consume the Application	842
14.6	Calling CRUD Operations from ABAP	846
14.7	Summary	847
 Conclusion		849
The Author		851
Index		853

Index

A

ABAP	
<i>constructs</i>	50
<i>development system</i>	47
<i>event mechanism</i>	558
<i>packages</i>	44
<i>Quick Assist</i>	56
ABAP 1809	65, 313, 353
ABAP 7.3	261
ABAP 7.4	264, 311
ABAP 7.5	56
<i>features</i>	65
<i>new features</i>	137
ABAP Channels	761
<i>general concept</i>	762
ABAP Console	73
ABAP Doc	71
ABAP Editor	282
ABAP in Eclipse	
<i>ADT</i>	36
<i>connection</i>	42
<i>global structures</i>	67
<i>missing method</i>	52
ABAP in the cloud	798
ABAP Managed Database Procedures	
(ADMP)	306, 328, 336, 354
<i>ABAP program</i>	360
<i>CDS view</i>	360
<i>Eclipse</i>	355, 357
<i>method definition</i>	373
ABAP Messaging Channels	762–763, 768
<i>coding the receiving application</i>	775
<i>coding the sending application</i>	770
<i>example</i>	779
<i>framework</i>	772
<i>SAP GUI</i>	767
<i>warning</i>	769
ABAP programming model for	
SAP S/4HANA	327, 802
ABAP Push Channels	762, 764–765, 783
<i>coding the sending application</i>	792
<i>SAPUI5</i>	782
<i>test tool</i>	791
ABAP SQL	306–307
<i>new commands</i>	308
ABAP Test Cockpit	73, 93, 364, 366, 753
ABAP Test Double Framework	
(ATDF)	262, 267, 270
ABAP Unit	223
ABAP Unit Framework	753
ABAP Unit Test Double Framework	755
ABAP Workbench	35–36, 45, 54, 83, 471, 680
ABAP_TRUE	153–154
abapGit	97
<i>documentation</i>	100
<i>installation</i>	99
<i>packages</i>	112
<i>production support</i>	127
<i>project collaboration</i>	121
<i>repository</i>	99
<i>SAP versions</i>	100
<i>staging screen</i>	129
<i>storing and moving objects</i>	108
<i>transaction</i>	100
<i>versions</i>	105
<i>versus SAPLink</i>	108
<i>watch repository</i>	107
abapLint	126
Access condition parameter	348
Actions	829
Adapter pattern	468, 559
Agile development	733
Alias	335, 337
ALPHA formatting option	149
ALV	289
<i>application</i>	767
<i>function modules</i>	556
<i>grid</i>	597, 610, 620
<i>interface</i>	772
<i>list program</i>	196
<i>report</i>	461, 696, 761
<i>SALV</i>	523
<i>screen</i>	708
Annotation	69, 333, 386, 696
ANSI-standard SQL	357
Antifragile	623

Application configurations 631, 641
Application log 300
Application log object 293
Application model 536
Application-defined function 540
Artifacts 50
Assemble/act/assert test 251
ASSERT 215, 259
Association 332, 647, 715
Asterisks 319
Authority checks 301, 347–348, 410
Autosave 76

B

BAPEX structure 824
Behavior definition 811
Behavior implementation 813
Behavior-driven development 250, 279
Belize 587
Big data 138
Bill of materials (BOM) 379
BOOLC 155
Boolean logic 155
Boolean variable 156
BOPF 35, 151, 377–378, 622
 action validations 439
 actions 433–434
 authority checks 409
 callback subclass 457
 change document subnode 456
 configuration class 402
 create header node 382
 create item node 384
 create model classes 390
 create object 380
 creating an action 433
 creating/changing objects 394
 CRUD 446
 custom enhancements 464
 custom queries 395–396
 delegated objects 455
 determinations 410
 FPM 643
 header record 402
 locking objects 407

BOPF (Cont.)
 persistence class 393
 read object 417
 recommended reading 468
 service manager 404
 testing 460
 tracking changes 452
 unit testing 461
 validations 424, 426
 wrappers 466
BOR object 665
Branching 120
 create new 132
 master branch 131
 reverting 133
 switch 132
Breakpoints 285–286
 user-specific 294
BRFplus 424, 469, 477, 483
 application settings 479
 BOPF integration 502
 call function from ABAP 501
 call in ABAP 498
 create application 477–478
 decision table 490, 505
 decision trees 486
 enhancements 516
 example 502
 recommended reading 521
 rule logic 486
 SAP Business Workflow 512
 simulations 509
 unit testing 511
 workflow task 514
BSP application 733
Buffer class 817
Buffering 316–317
BUnit 461
 test definition 463
 test implementation 464
Business definition 804
Business object 395, 465, 804
 CDS views 386
 manual definition 379
 saving changes 837

Business Rule Framework (BRF) 469
Business rule management system
 (BRMS) 469, 486
Business rules 469, 472
 ABAP 476
 BRFplus 477
 customizing tables 474
Business transaction events 766

C

Calling code 51
Calling program 199, 528
Cardinality 340
CASE 153, 157, 185, 310, 325, 337
CASE statement 308, 310, 337
Case-insensitive search 313
CDS test double framework 351
CDS views 69, 306, 327, 331, 387, 411, 586,
 691, 698, 726, 730, 806–807
 BOPF annotations 388
 buffering 334
 building 329
 classify 345
 define 343
 definition 344
 Eclipse 350
 extend view 344
 header settings 335
 open 368
 parameters 344
 read from ABAP 348
 unit testing 351
Centralized version-control system 120
Certificates 102
Change document 457
Changing parameter 185, 536
Channel extension 773
CHECK 415, 420, 429
Check method 260
CHECK_DELTA 415, 417, 429
CL_SALV_TABLE 314, 525, 527
Class invariants 219
Class under test 233
Class-based exception 201, 204

CLEANUP 207–208
Clover 81
Code generator 619
Code Inspector 73, 131
Code pushdown 306, 364
 AMDP 372
 CDS views 372
 example 369
 locating code 366
 OpenSQL 371
 techniques 366
Combinatorial test design 277
Combined structure 385
Complex objects 266
Component 596
Component configuration 632, 636
Component controller 618
COMPONENTCONTROLLER 627
Conceptual thinking 406
COND 158
Conditional logic 143, 153
Configuration table 459
Consistency validation 444
Constructor expression 143
Constructor injection 238
 arguments against 239
Constructor operator 151, 159, 165
Containers 514, 562, 564
 create automatically 563
Context parameter 488
Contract violation 271, 584
CORRESPONDING 164
CREATE method 821
Creating while reading 314
CROSS JOINS 320
Cross-origin resource sharing
 (CORS) 681, 725
CRUD 397, 445, 679, 846
CRUD methods 815
Customer requirements 480, 492
Customizing settings 227
Customizing table 474
CX_DYNAMIC_CHECK 195–196
CX_NO_CHECK 197
CX_STATIC_CHECK 194–195

D

Daemons 765
 class methods 766
Data changed event 560
Data class 345
Data declaration 161
Data definition 247–248
Data dictionary 69
Data element, creation 65
Data provider class 679
Data request flow 622
Data source 335
Data type declaration 139
Data validation test 256
Data values 411
Database access class 236
Database layer 307
DCL (Data Control Language) 347
DCL source 348
DDIC 396
 configuration table 496
 data element 482
 descriptions 633
 field 668
 objects 383
 structure 482, 600, 665
 table 69, 334, 668
DDL 329, 333, 338, 372
 definition 361, 695
 source 362
Debugger 84
Debugging 177, 281
 changing variables 291
 control 289
 manual 290
 saving script 299
 script program 282
 text note 292
 trigger settings 285
Decision logic 469
Decision tables 491, 497, 507
 create 491
Decision tree 486, 488, 490
Deep structures 165
Delegated object 454
DELETE method 827

Dependencies 224, 228, 351
 breaking up 228, 231
 eliminating 225
 identifying 226
 inversion 415
 lookup 239
Design by contract 191, 214, 216, 256, 549, 584
 class invariants 218
 postconditions 216
 preconditions 216
 violations 221
Design mode 706
Design Patterns—Elements of Resuable
 Object-Oriented Software 378
Determination 418
Dialog box 722
Distributed version-control system 121
Domains 66, 482
Downcast 182
Draft document 387
Dropdown menu creation 746
Duplicate code 56
Dynamic check 195–196
Dynamic exception 196
Dynamic log point 87–88
Dynamic SQL 634
Dynpro 378, 389, 524, 589, 594, 612
 UI framework 390
Dynpro Screen Painter 605, 607

E

Eclipse 35, 45, 76, 329, 389, 814
 AMDP 357
 bookmarking 50
 CDS view 332
 class constructors 63
 connect to backend system 42
 create attributes 63
 create parameters 63
 debugging 83
 extract method 56
 Extract Method Wizard 61, 73
 features 43
 help 74

Eclipse (Cont.)
 in the cloud 95
 installation 37, 39
 multiple objects 48
 Neon 76
 Oxygen 78
 Photon 38
 plug-ins 89
 prerequisites 37
 Quick Assist 54
 recommended reading 96
 refactoring 63
 release cycle 35, 47
 runtime analysis 86
 SAP add-ons 40
 SAP HANA 329
 SAPUI5 699
 SDK 89
 unit tests 79
 unused variables 62
 word wrap 77
Eiffel 216, 218
Ellison, Larry 305
ELSE clause 337
END method 284, 293
Entities 664
Entity Manipulation Language (EML) 846
Entity set 666
Enumeration 144, 146
Error function 723
Error handling 203, 210, 678
 method 558
 RESUME 212
 RETRY 210
Errors 294, 724
ESLint 753
Exception 70, 191, 193, 690
 examples 191
 method clean up 209
 raising 193, 202
 recommended reading 222
Exception classes 191, 193, 195, 429, 690
 choosing type 198
 constructor 201
 creation 199
 declaring 201

Exception classes (Cont.)
 design 199
 types 193
Exception handling 193
Exception object 193, 204
EXECUTE 415, 421, 436
Existence check 317
Export parameter 185, 216
Expression type 486
Extended syntax check 62
External breakpoint 688, 793

F

Factory class 562
Factory method 64, 240, 393
Fast ALV Grid Object (FALV) 586
Favorites list 90
Feeder 632
Feeder class 632
 methods 633
Field catalog 554, 574
Field symbol 171
FILTER 178
Filter structure 396
FitNesse 251
Floorplan Manager 35, 465, 589, 628
 BOPF 643
 floorplans 629
 GUIBBs 631
 Guided Activity Floorplan 629
 Overview Floorplan 629
 Quick Activity Floorplan 629
 recommended reading 655
 UIBBs 631
Flowchart 473
FLUID tool 650
FOR 142
Foreign key 340
FORM routine 46, 51
Form-based approach 67
Fragment 704
Function 479, 492
Function module 202, 204–205, 564
 signature 204
Functional methods 355

G

Generic method 467

Generic User Interface Building Blocks

 (GUIBB) 631, 643

components 638

GET_ENTITY_SET 680, 686, 688

Gherkin 756

Git 98

GitHub 98, 103, 129

create project 110

errors 103

GIVEN method 652

Global class 80

God class 527

Gorilla testing 122

GROUP BY 175

GUID 381, 404

key 394

GuiXT 758

H

Hard-coded restrictions 337

Hashed key 179

Head First Design Patterns 236

Helper class 392, 399

Helper methods 250, 547

Helper variable 164

Hollywood Principle 763

Host name 725

Hotspot 539

HTTP authentication 106

Hungarian notation 499

I

IDocs 661

IF/ELSE 158

IF/THEN 153

Importing parameters 361

Importing table 535

Inbound plug 616, 627

Index file 750

Information/Warning/Error 205

INIT method 284, 293

INITIALIZE 540, 565

INITIALIZE method 572

Injection 233, 237, 239

Injector 241

INNER JOIN 318

Input parameter 481

Integrated data access 583

Interfaces 186

Internal tables 160, 404

grouping 175

new functions 171

Internet of Things (IoT) 762, 793

Isolation policy 763

J

Java 35, 140, 657

Java EE perspective 702

JavaScript 657, 700

library 660, 743

JavaScript program 48

Joe Dolce principle 267

JSON Voorhees 771

L

Layout data property 609

Lead selection 619, 625

LET 143

LINE_EXISTS 173

LINE_INDEX 172

Local host 724

Local variable 359

LOCK method 835

Logging 292

Logging class 64

Logical condition 488–489, 495

Logical unit of work 450

M

Mapping object 170

Master data 340

Message object 430, 774

Message producer 774

Method 56, 203

Method autocreation 53

Method call 60, 220

Method definition 533

Meyer, Bertrand 218

Microsoft Outlook 668

MIME repository 274

Mock class 235

Mock objects 248

Mockup Loader 273–274

Model 691

Model class 378, 391, 397, 772

Model provider class 679

Module pool transaction 600

MOVE-CORRESPONDING 164–165

dynamic 169

MVC pattern 378, 389, 526, 542, 558, 590, 643, 660

controller 526, 596

location of model 591

model 526, 542, 591

model as an assistance class 592

model declared in the controller 593

model inside the view 592

view 526, 593

N

NativeSQL 308

NEW 140

No check 196

Node structure 602

Nuggets 109

O

Object authorization class 409

Object Linking and Embedding (OLE) 768

object_configuration 393

Object-oriented programming (OOP) .. 51, 182, 192, 204, 235, 378, 466, 525–526

OData 661, 678

documentation 684

service 697, 729

Offline repositories 115

create new 116

On start message 790

Online repositories 109

OPA 755

Open SQL Test Double Framework 325

Open-closed principle 87, 360

OpenSQL 307, 338

query 309

OpenUI5 743

Outbound plug 614

parameter 615

Overlap check 508

Overview page 640

P

Pace layering 391

Package assignment 71

Parameter ID 770

Parent node 384

Patterns 150

PBO processing 424

Perspective 88, 703

Post exit 569

Postcondition 217

Precondition 217

PREPARE 435

Pretty print 92

Private method 59, 246

Procedural programming 225

Procedure call 517

Processing block 177

Program assumptions 214

Projects 664

PROLOGUE method 284

Prototype screen 733

Proxy 725

calls 661

servlet 725

settings 726

Public method 199

Publish and subscribe 783

Pull requests 126

Push Channel Protocol (PCP) 766, 771, 773

interface 776

Q

Query logic 396

JUnit 754

R

RAISE SHORTDUMP 221

RAP

application structure 806

business logical layer 803

database layer 802

user interface layer 805

READ method 832

READ TABLE 162

Read-only mode 579

REDUCE 174

Refresh display 560

Regular expressions 163

Remote function call (RFC) 661

Report programming 523

Repository object 104

REST 661

RESTful ABAP programming model

 (RAP) 797

Result method 260

RETRIEVE DEFAULT PARAM 435

Return parameter 515

Return values 265

RevTrac 135

RFC function module 355

Rule logic 486

Rules engines 470

Rulesets 483

create 484

definition screen 485

S

SALV 524–525

add custom icons 561

application-specific changes 542

CL_SALV_GUI_TABLE_IDA 583

concrete class 527

create container 534, 563

design report interface 531

display report 556

editable fields 575

editing data 569

event handling 538

framework 557

grid refresh 573

SALV (Cont.)

grids 572

initialize report 534

object editability 571

recommended reading 588

report 551

 SAP HANA 586

with IDA 583

SAP Build

prototype 739

templates 737

SAP Business Workflow 391, 512–513

SAP Cloud Appliance Library 798

SAP Cloud Platform 741, 844

SAP Cloud Platform cockpit 728

SAP Cloud Platform Connectivity's cloud

connector 728

SAP Cloud Platform, ABAP environment ... 138, 799

SAP Code Exchange 273

SAP Community 36

SAP Decision Service Management 471, 512

SAP EarlyWatch Check 761

SAP ERP 750

SAP Fiori 587, 654, 729, 757, 843

SAP Gateway 657, 661, 752

coding 678

configuration 663

create model 663

create service 671

creating entities 665

creating services and classes 670

data provider class 679

error handling 690

model provider class 679

Service Builder 664

service implementation 678

testing 676

SAP GUI 87, 188, 523, 733, 763

embedded 85

 SAP Fiori 587

SAP HANA 322, 586, 660, 685, 769

 AMDP 328

 CDS views 327–328

code pushdown 306, 364

database views 306

 DDL 329

 Eclipse 329

SAP HANA (Cont.)

recommended reading 376

stored procedure 354

SAP HANA rules framework 519

SAP Messaging Channels

activity scope 771

receiver object 777

subscriber object 778

SAP NetWeaver Development Tools for ABAP

 (ADT) 36, 695

SAP Process Integration

 (SAP PI) 384, 661, 768, 776

SAP Push Channels

code for incoming messages 785

coding receiving components 783

testing the APC service 790

SAP S/4HANA 305

migration 73

SAP S/4HANA Cloud 798

SAP Screen Personas 758

SAP Web IDE 729, 736, 758, 844

cloud version 728

menu 732

templates 726, 845

SAPlink 105, 109, 128, 587

SAPUI5 35, 386, 465, 590, 657, 763, 792

architecture 659

browser support 678

buttons 722

consume application 842

controller 718

design mode 708

 Eclipse 662, 699

fragment XML file 713

function for testing 724

functions 720–721

HTML file 704

importing applications 748

JavaScript 660

prerequisites 661

recommended reading 759

search button 709

storing applications 748

testing 724, 750

view 704

view and controller 699

XML file 706

Scalar functions 355

Screen Painter 736

SCRIPT method 282, 284, 287, 293, 298

Script Wizard 287, 290–291, 294, 296, 300

options 288

Search configuration 637

Search helps 187

predictive 188

Search UIBB 636, 641

SELECT 308, 313, 322

Selection criteria 637

Separation of concerns 231, 623

Separators 568

Service adaptation definition language

 (SADL) 694

Service bindings 839

Service Builder 664

Service definitions 823, 839

service_manager 393

SET_COLUMN_ATTRIBUTES_METHOD 546

SETUP 249

Short dump 70, 149, 151, 221

SICF framework 691

SICF service node 680

Signature definition 151

Single responsibility principle 231

Single-record view 624

Slinkees 109

Smart templates 727

SOLID principles 245

Sort criteria 554

Sort order 555

Sorted key 179

Source code view 46

Source mode 706

SQL 685

calculations 311

queries 310

SQL for the web 685

SQL Monitor 364

SQL Performance Tuning Worklist 364, 366

SQL trace 374

SQL view 333

SQL-92 standard 309

SQLScript 328, 354, 357–358

SSL 101

SSL client 102

Staging	124
Stateful	785
Stateless	785
Static check	194
Static method	362, 777
Stored procedure	307
String processing	148
Structured variable	436
Structures	67
Stub objects	233
Subclassing	458
Subnodes	385
SWITCH	157
Syntax check	194, 299, 324
System alias	671
System coloring	94
SY-TABIX	174
T	
Table join	320
Table work areas	160
TCP protocol	794
Technical columns	546
Template	
<i>customization</i>	730
<i>views</i>	332
Test class	243, 245, 252, 462
<i>definition</i>	246
Test code	261
Test data	254
Test doubles	264
Test injection	234
Test methods	249
Test seams	228
Test value	509
Test-driven development (TDD)	79, 81, 223, 244, 257, 652
<i>pattern</i>	244
The Pragmatic Programmer	256
THEN method	652
Tooltip	547, 553
Trace file	87
Tracing	301
Transaction	616
<i>/BOBF/TEST_UI</i>	424
<i>/BOBF/CONF_UI</i>	455

Transaction (Cont.)	
<i>/BOBF/TEST_UI</i>	438
<i>/IWFND/ERROR_LOG</i>	688
<i>/IWFND/MAINT_SERVICE</i>	672, 675, 686
<i>BOB</i>	379–380, 395
<i>BOBF</i>	379
<i>BOBF/TEST_UI</i>	460
<i>BOBX</i>	379, 462
<i>BOPF_EWB</i>	465
<i>CICO</i>	761
<i>COO8</i>	424
<i>FLUID</i>	647
<i>FPM_WB</i>	630, 644
<i>FPM_WDA</i>	638
<i>IWFND/MAINT_SERVICE</i>	698
<i>MRRL</i>	322
<i>RSSCD100</i>	461
<i>SALV</i>	772
<i>SAMC</i>	770, 773, 784
<i>SAPC</i>	790
<i>SAT</i>	86
<i>SATC</i>	364
<i>SCDO</i>	453, 458
<i>SE03</i>	71
<i>SE11</i>	68, 188, 328, 336
<i>SE24</i>	182, 199–200, 471, 670
<i>SE37</i>	471
<i>SE38</i>	46
<i>SE80</i>	35, 43, 45, 81, 86, 104, 189, 471, 599, 639
<i>SEGW</i>	664, 672, 691
<i>SICF</i>	90, 670, 676–677, 751, 783, 790
<i>SIMGH</i>	672
<i>SLIN</i>	62
<i>SM12</i>	408
<i>SM36</i>	106
<i>SMWO</i>	274
<i>SQLM</i>	364
<i>ST05</i>	316, 348, 364
<i>ST22</i>	200
<i>STRUST</i>	101, 103
<i>SU01</i>	106
<i>SU53</i>	348
<i>SWLT</i>	364
<i>SWO1</i>	665
<i>ZMAM</i>	775

transaction_manager	393
Transactional business object	
application	807
Transactional view	386
Transient structure	383, 412
Transport request	59, 134, 514
TRUE/FALSE	156
TRY/CATCH/CLEANUP	203
TYPE definition	54
TYPE REF TO DATA	152
U	
UMAP	90
Underlying grid object	576
UNION	320
Unit testing	81, 105, 231, 243, 301, 651
<i>ABAP 7.4</i>	189
<i>ABAP SQL</i>	324
<i>automation</i>	267
<i>executable specifications</i>	243
<i>massive data amounts</i>	272
<i>mockA</i>	267
<i>recommended reading</i>	279
UPDATE method	827
User acceptance testing (UAT)	251
User command handling	579
User command routine	772
User commands	536, 567
<i>processing</i>	197
<i>toolbar</i>	566
User exits	302, 517
User Interface Building Blocks	
(UIBB)	630, 643
<i>freestyle</i>	631
V	
Validation	426
<i>category</i>	427
<i>coding</i>	429
<i>creation</i>	427
<i>overview</i>	428
Validation logic	441
VALUE	140
Variables	138, 289, 291

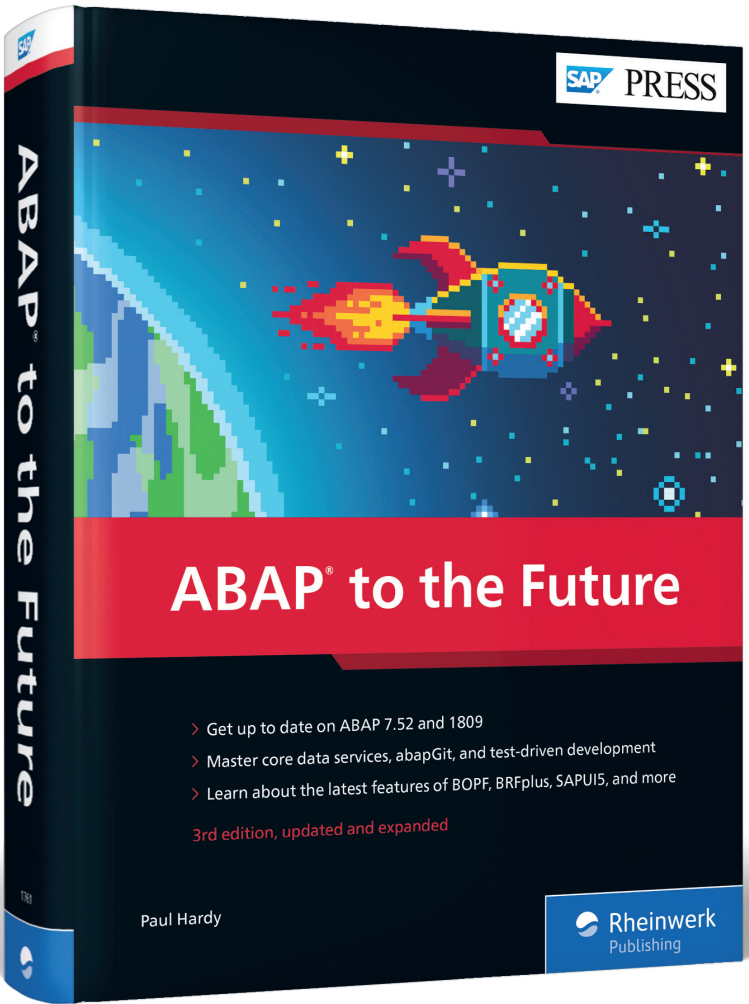
Variant Configuration	503
Version control	98
View controller	596
Views	612, 619
<i>inside windows</i>	613
<i>navigation</i>	614
Violated postcondition	256
Violated precondition	549
W	
Watchpoints	285–286, 295
<i>field symbol</i>	301
Web Dynpro ABAP	35, 151, 518, 589
<i>ALV grid</i>	589
<i>application building</i>	597
<i>calling application</i>	616
<i>Code Wizard</i>	618
<i>coding</i>	617
<i>component controller</i>	596
<i>create component</i>	599
<i>data structures</i>	600
<i>defining view</i>	609
<i>defining windows</i>	612
<i>graphical screen painter</i>	594
<i>interface controller</i>	600
<i>linking the controller</i>	618
<i>nodes</i>	603
<i>PAI</i>	595
<i>PBO</i>	595
<i>recommended reading</i>	655
<i>SAP Fiori</i>	654
<i>selecting records</i>	618
<i>standard elements</i>	605
<i>storing data</i>	594
<i>unit test</i>	653
<i>unit test framework</i>	651
<i>view settings</i>	604
WebSocket	762, 764, 785, 790
<i>connect from SAPPUI5</i>	792
WHEN method	652
WHERE clause	312, 341
Window controller	596
Work area	162
Workflow	512
Workflow Builder	513, 516

X

XML 700, 704
 tree 382
XSDBOOL 155

Z

Z aggregated storage table 322
Z class 44, 246, 293, 394, 517, 572, 660
Z table 505, 799
ZABAPGIT 100
ZCL_BC_VIEW_SALV_TABLE 564, 571
ZCX_NO_CHECK 196



Paul Hardy

ABAP to the Future

864 Pages, 2019, \$79.95

ISBN 978-1-4932-1761-8

 www.sap-press.com/4751



Paul Hardy is a senior ABAP developer at Hanson and has worked on SAP rollouts at multiple companies all over the world. He joined Heidelberg Cement in the UK in 1990 and, for the first seven years, 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 SAP Community 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).

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.