

Testing Calculations with ColumnFixture Tables

We often want to test that calculations are being carried out correctly, according to some business rule. Tables of concrete examples help us to understand business needs and communicate what's required. Here, we will focus on how to read such Fit tests; later, we will show how to write them.

We begin with two simple examples that test calculations by using tables of type `ColumnFixture`, which is designed to do this.

These tests are rather abstract in that they say nothing about how someone using the system under test will see the consequences of this business rule. In Chapter 4, we'll see tests that are more aligned to the step-by-step use of the system under test.

3.1 Calculating Discount

The business rule for our first example follows.

A 5 percent discount is provided whenever the total purchase is greater than \$1,000.

We use examples in a table to help define the relationship between the amount and the discount for several cases. For example, when the amount is \$1,100, the discount is \$55. Figure 3.1 shows a simple set of tests for the part of a system that calculates discounts.

The *fixture*, `CalculateDiscount`, is named in the first row of the table. The fixture determines how the examples in the table are to be tested automatically against the system under test.

As this is a `ColumnFixture` table, the second row is a *header* row, which labels the *given* and *calculated* value columns. Here, the *amount* column holds the *given* values, and the *discount()* column holds the *calculated* results that are expected.¹

¹ The label for a *calculated* result column has `()` after the name.

Column labels in the header row serve two purposes.

1. They help us to understand the examples in the table.
2. They are used when the examples are automatically tested by Fit.

CalculateDiscount	
<i>amount</i>	<i>discount()</i>
0.00	0.00
100.00	0.00
999.00	0.00
1000.00	0.00
1010.00	50.50
1100.00	55.00
1200.00	60.00
2000.00	100.00

Figure 3.1 Fit Table for Testing Discount (with an Error)

The remaining eight *test rows* of the table are our test cases, which are checked one at a time. For the first test case, the given value under *amount* is 0, and so a *calculated* value of 0 under *discount()* is expected. The second test case is independent of the first; the *given amount* is 100, and so the *calculated discount()* is expected to be 0. And so on.

Fit runs the tests, one row at a time, against the system under test. Fit begins with the third row of the table, providing the *given* value of 0.00 to the application. Fit checks that the result calculated by the application, based on that *amount*, is 0.00, as *expected*.

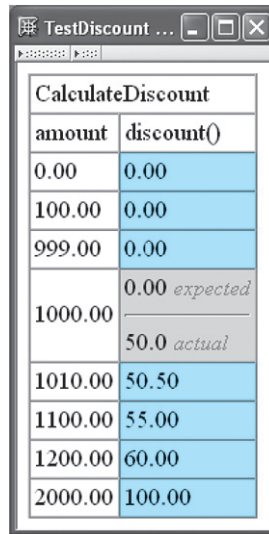
3.2 Reports: Traffic Lights

When Fit is run, it produces a report. The table cells are colored to show the results of the tests, so that we can tell which tests passed and which failed. This makes it easy to see which tests went right and which went wrong. Fit uses a traffic light metaphor for the colors in the report.

- *Green*: Test passed.
- *Red*: Test failed to meet expectations. Additional information may be provided.
- *Yellow*: Part of a test is not completely implemented, or something else went wrong.
- *Gray*: Parts of tables that have not been processed for some known reason, perhaps because a test is not completely implemented.

Other cells are left unmarked unless something goes wrong. A reported table may have rows added to it to provide extra information about failing tests.

The results of the tests in Figure 3.1 are reported as shown in Figure 3.2 (see also Plate 1). The passing tests, whose calculated value of *discount()* is as expected, are green. The report contains a single failing test. This failed test is red because the system under test is incorrect when the amount is 1,000.00. The actual value that was calculated by the system under test (50.0) is added to the report.



CalculateDiscount	
amount	discount()
0.00	0.00
100.00	0.00
999.00	0.00
1000.00	0.00 <i>expected</i>
	50.0 <i>actual</i>
1010.00	50.50
1100.00	55.00
1200.00	60.00
2000.00	100.00

Figure 3.2 Fit Report for TestDiscount

Tip

The ordering of the test rows in this particular table is not important for automated testing, as the tests are independent. However, the order is important to the people who read and write such tests.

Are there any questions at this stage?

Questions & Answers

Are these tests, or are they requirements?

They're both. The concrete examples help us to think about and communicate the important requirements to software developers. As the examples are in a suitable form, they can also be used to automatically test that the resulting software application does what's expected.

But the tests can't *define* all situations.

That's right; we use concrete examples to give an idea of the general case. They need to be augmented with other forms of communication: general statements of the underlying business rules and discussion between the people concerned.

What if the business rules are not well understood or are changing?

Creating the examples helps to focus on the essence of the rules. Examples serve as a way to sort out how we want to talk about the important issues that a system has to deal with. It can be easier to discuss business issues in the context of specific examples rather than trying to deal with generalities. The examples help to break down the important cases so that we don't have to deal with everything all at once. (We cover the dynamics of this process in detail in Part II.)

If the business rules are changing, we need a way to express those changes and to talk about them. That may mean simply adding new examples to cover new cases. Or it may mean changing our terminology because the essence of the rules has to be altered. Either way, as we see in Chapter 18, it's important that we can change the Fit tests as the world changes.

What happens if you put something other than a number in a cell?

When the wrong sort of data is in a table cell, an *error* is marked with yellow in the report that Fit produces.

Can we use values other than numbers?

Yes; in general, a cell table can contain any sort of values, such as text strings, dates, times, and simple lists. See Section 3.4 for an example that uses simple lists.

Programmers: The fixture code for the tests here is given in Section 21.1 on p. 179.

3.3 Calculating Credit

Here's the business rule for our second example of using a `ColumnFixture` table to test calculations.

Credit is allowed, up to an amount of \$1,000, for a customer who has been trading with us for more than 12 months, has paid reliably over that period, and has a balance owing of less than \$6,000.

Again, we use a table to define whether credit will be extended, based on the values of the various given characteristics of our customers. A small set of tests for this business rule is shown in Figure 3.3.

The first row of the table names `CalculateCredit`, the *fixture*, which determines how the examples in the table are to be tested automatically against the system under test. In this book, we follow a convention that fixtures that test calculations have *Calculate* in their name.

CalculateCredit				
<i>months</i>	<i>reliable</i>	<i>balance</i>	<i>allow credit()</i>	<i>credit limit()</i>
14	true	5000.00	true	1000.00
0	true	0.00	false	0.00
24	false	0.00	false	0.00
18	true	6000.00	false	0.00
12	true	5500.00	true	1000.00

Figure 3.3 Fit Table for Testing Credit

The second (header) row of this table, as usual, labels the *given* and *calculated* value columns for testing creditworthiness. Two columns have *calculated* values: *allow credit()* and *credit limit()*. The given value of *reliable* and the calculated value of *allow credit()* will be either true or false. As is usual with `ColumnFixture` tables, the test rows are independent of one another.

In Figure 3.3, the four cell values in *italics* indicate that credit is refused. Test writers can format Fit tables to better organize them, as we'll see later, and format cells values of special interest to highlight them.

Fit reports the results as shown in Figure 3.4 (see also Plate 2). In each *test row* of the report, the two cells for the calculated results that are expected are marked. The tests in the first four rows are *right* and are marked in green. The last row of the table fails, with two cells marked *wrong*, in red.

CalculateCredit										
<i>months</i>	<i>reliable</i>	<i>balance</i>	<i>allow credit()</i>	<i>credit limit()</i>						
14	true	5000.00	true	1000.00						
0	true	0.00	false	0.00						
24	false	0.00	false	0.00						
18	true	6000.00	false	0.00						
12	true	5500.00	<table border="1"> <tr> <td>true expected</td> <td>1000.00 expected</td> </tr> <tr> <td>false actual</td> <td>0.0 actual</td> </tr> </table>	true expected	1000.00 expected	false actual	0.0 actual	<table border="1"> <tr> <td>1000.00 expected</td> <td>0.0 actual</td> </tr> </table>	1000.00 expected	0.0 actual
true expected	1000.00 expected									
false actual	0.0 actual									
1000.00 expected	0.0 actual									

Figure 3.4 Fit Report for TestCredit

In general, a test can fail for several reasons. For example, in the report in Figure 3.4, the failing test itself could be incorrect, or the business rule may be stated incorrectly for the case of exactly 12 months.

Looking Ahead

Note that we don't test the business rule with specific customers; we have abstracted away from the table how we determine those characteristics for particular customers. We discuss such design issues for tables in Chapter 18.

This set of examples is fine for giving a sense of the business rule. However, they are incomplete from a tester's perspective. We'd need more tests to make sure that the business rule has been understood and implemented correctly in software. Testers have expertise in the art of choosing such test cases.

These tests are also badly organized. It's not easy to see what the rows are testing and what is missing. We take up this issue in Chapter 18.

You may have noticed the subtle redundancy between the two calculated fields. The value of `credit limit()` always depends, effectively, on the value of `allow credit()`. *Redundancy*, and why we want to avoid it in the design of tables, is discussed in Chapter 18.

Finally, in some situations, the rows of a `ColumnFixture` table are not independent, by choice, as we show in Section 4.1 on p. 23 and in Chapter 13.

Questions & Answers

What if we want to use `yes` and `no` instead of `true` and `false`?

They can be used instead in FitNesse (Chapter 8), where `yes`, `y`, `1`, and `+` can also be used for true. Anything else is false.

What if a `given` number is negative, which is not valid in the application?

We take up this issue in Chapter 9.

What's a `fixture`?

A fixture is the "glue" that connects the tests in a Fit table to your application.

From a programmer's point of view, a fixture names the software that is responsible for running the tests in the table against the system under test. Different fixtures may do different sorts of tests. Part III covers this in detail.

Will a whole test sequence be included in one table? Some of our tests go on for many pages.

No, not necessarily. We'll see in Chapter 6 that a test sequence may consist of many tables of varying types.

Programmers: The fixture code for the tests here is given in Section 21.2 on p. 182.

3.4 Selecting a Phone Number

Our third example illustrates the use of simple lists in tests. The business rule for this example is very simple.

The first phone number in the list of supplied phone numbers will normally be used for communicating with the client.

A single test is shown in Figure 3.5. The *phones* column contains a comma-separated list of phone numbers. The *calculated* column *first()* selects the first one in the list.

CalculateFirstPhoneNumber	
<i>phones</i>	<i>first()</i>
(209)373 7453, (209)373 7454	(209)373 7453

Figure 3.5 Fit Table for Testing First Phone

When the elements of a list are more complex, such as containing details of each order item within an order, this approach doesn't work. A `RowFixture` table can be used instead, as discussed in Chapter 5.

Questions & Answers

What if an element in the list contains a comma (,)?

In that case, a `RowFixture` tables could be used, as covered in Chapter 5. If you wanted to use such lists in a `ColumnFixture`, a programmer would need to write some fixture code to handle that specially, as we discuss in Chapter 25.

What happens if the phone list is empty?

The business rule doesn't cover that situation, but it does need to be defined. We take up this issue in Chapter 9, when we talk about handling various expected errors.

Programmers: The fixture code for the tests here is given in Section 21.3 on p. 184.

3.5 Summary

- Fit tests are defined in tables.
- `ColumnFixture` tables are good for specifying the expected *calculated* value based on the *given* value in a row.
- The test writer gets to choose the names of the labels in the header row, such as *phones* and *first()*.
- Each row of a `ColumnFixture` table represents an independent test when it's used for testing the same calculations on different data. We'll see `ColumnFixture` tables in Section 4.1, where the rows are not independent, and so their order is important.
- When Fit runs a test, it produces a report. This report gives feedback on the test by marking parts of the input table according to what passed.
- We've seen several examples of `ColumnFixture` tables; the second table had two *calculated* columns.

The next chapter introduces action-style tables. However, if you'd like to start using Fit on `ColumnFixture` examples first, you may like to look now at Chapter 7.

Questions & Answers

[We have lots of tests in text files. We wouldn't want to make tables from them.](#)

You don't need to do so. As we'll see in Chapter 7, tests in other input formats can be fed into Fit. However, some straightforward programming will be required for custom data formats (as discussed in Chapter 39).

[Our existing tests all center on the process of using the system. There are no "calculation" tests.](#)

Often, tests are written in this way because the only way to check them has been to run the system through the user interface and carry out a sequence of steps. When we look more closely at those tests, we often get an inkling of the underlying business rules.

As discussed in Part II and especially Chapter 18, we aim to extract those essential business rules and express them independently of the work flow. That extraction has several benefits. We gain clarity about the rules. By expressing them succinctly, we can more easily discuss the rule and see what cases (examples) we may need to add or change as the business changes. It is much faster to create the (short) examples, as they can ignore the workflow. The tests are not dependent on the user interface, which is often the part of a system that changes the most. Those tests can be made to run much faster, if necessary.

[But that assumes that we can test the business rules in isolation!](#)

Yes, it seems to assume that. We discuss various ways for programmers to manage this in Chapter 33.

How do we decide on suitable tests for our application?

We'll cover that in some depth in Part II, once we've seen several examples of various sorts of tests. The process of writing tests usually starts from thinking about the important things in your business domain and what you want to say about them. That's what we're calling "business rules." In some cases, they're quite obvious; in others, they come out of clear thinking about what's needed by exploring with concrete examples.

If you now want to see how Fit tests are created, skip ahead and read Chapters 12 and 13. The chapters after those two may not make complete sense; you will probably need to come back and read more of the chapters in this part first.

How do the programmers know what to do with a Fit table I've written?

To write the fixture code, they may need to talk to you about what you intend with the test. After that, they'll know what to do with similar tables.

3.6 Exercises

Answers to selected exercises are available on the book Web site; see Appendix B for details.

1. The business rule from Section 3.1 on p. 13 is as follows: "A 5 percent discount is provided whenever the total purchase is greater than \$1,000." Using this business rule, color in Figure 3.6, based on the traffic light colors in Section 3.2 on p. 14. Do so as it would be reported by Fit, according to whether the tests pass. You don't need to bother to insert *actual* values where a test value is *wrong*.

CalculateDiscount	
<i>amount</i>	<i>discount()</i>
1.00	0.00
10.00	0.50
1400.00	75.00
20000.00	200.00

Figure 3.6 Color in the Table

2. The business rule from Section 3.3 on p. 16 is as follows: "Credit is allowed, up to an amount of \$1,000, for a customer who has been trading with us for more than 12 months, has paid reliably over that period, and has a balance owing of less than \$6,000." Color in Figure 3.7, using this business rule, as in the previous exercise.
3. Create a `ColumnFixture` table for the following.

Percentages of dollar amounts are rounded to the nearest cent.

CalculateCredit				
<i>months</i>	<i>reliable</i>	<i>balance</i>	<i>allow credit()</i>	<i>credit limit()</i>
13	true	5900.00	true	1000.00
12	true	0.00	false	1000.00
24	false	1000.00	false	0.00
18	false	6000.00	true	1000.00

Figure 3.7 Color in the Table

4. Create a `ColumnFixture` table for this business rule.
We charge for time, based on the charge rate of the consultant, in units of whole hours and with a minimum charge of \$5,000.
5. Create a `ColumnFixture` table for a simple business rule that is relevant to you and that involves simple calculations.