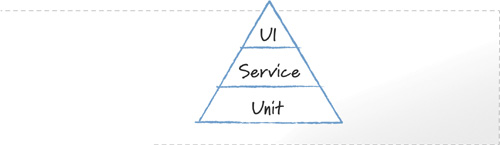
**AUTOMATE AT DIFFERENT LEVELS**

Even before the ascendancy of agile methodologies like Scrum, we knew we should automate our tests. But we didn’t. Automated tests were considered expensive to write and were often written months, or in some cases years, after a feature had been programmed. One reason teams found it difficult to write tests sooner was because they were automating at the wrong level. An effective test automation strategy calls for automating tests at three different levels, as shown in [Figure 16.1](https://www.safaribooksonline.com/library/view/succeeding-with-agile/9780321660534/ch16.html#ch16fig01), which depicts the *test automation pyramid*.

**Figure 16.1** The test automation pyramid.



At the base of the test automation pyramid is unit testing. Unit testing should be the foundation of a solid test automation strategy and as such represents the largest part of the pyramid. Automated unit tests are wonderful because they give specific data to a programmer—there is a bug and it’s on line 47. Programmers have learned that the bug may really be on line 51 or 42, but it’s much nicer to have an automated unit test narrow it down than it is to have a tester say, “There’s a bug in how you’re retrieving member records from the database,” which might represent 1,000 or more lines of code. Also, because unit tests are usually written in the same language as the system, programmers are often most comfortable writing them.

Let’s skip for a moment the middle of the test automation pyramid and jump right to the top; the user interface level. Automated user interface testing is placed at the top of the test automation pyramid because we want to do as little of it as possible. We want this because user interface tests often have the following negative attributes:

• **Brittle.** A small change in the user interface can break many tests. When this is repeated many times over the course of a project, teams simply give up and stop correcting tests every time the user interface changes.

• **Expensive to write.** A quick capture-and-playback approach to recording user interface tests can work, but tests recorded this way are usually the most brittle. Writing a good user interface test that will remain useful and valid takes time.

• **Time consuming.** Tests run through the user interface often take a long time to run. I’ve seen numerous teams with impressive suites of automated user interface tests that take so long to run they cannot be run every night, much less multiple times per day.

Suppose we wish to test a very simple calculator that allows a user to enter two integers, click either a *multiply* or *divide* button, and then see the result of that operation. To test this through the user interface, we would script a series of tests to drive the user interface, type the appropriate values into the fields, press the multiply or divide button, and then compare expected and actual values. Testing in this manner would certainly work but would be prone to the brittleness and expense problems previously noted.

Additionally, testing an application this way is partially redundant—think about how many times a suite of tests like this will test the user interface. Each test case will invoke the code that connects the multiply or divide button to the code in the guts of the application that does the math. Each test case will also test the code that displays results. And so on. Testing through the user interface like this is expensive and should be minimized. Although there are many test cases that need to be invoked, not all need to be run through the user interface.

And this is where the *service layer* of the test automation pyramid comes in.

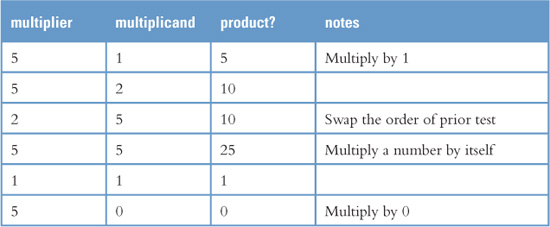
Although I refer to the middle layer of the test automation pyramid as the service layer, I am not restricting us to using only a service-oriented architecture. All applications are made up of various services. In the way I’m using it, a service is something the application does in response to some input or set of inputs. Our example calculator involves two services: multiply and divide.

SEE ALSO

Service-level testing was also described as a technique for specifying the behavior of a system through examples in [Chapter 13](https://www.safaribooksonline.com/library/view/succeeding-with-agile/9780321660534/ch13.html#ch13), “[The Product Backlog](https://www.safaribooksonline.com/library/view/succeeding-with-agile/9780321660534/ch13.html#ch13).”

Service-level testing is about testing the services of an application separately from its user interface. So instead of running a dozen or so multiplication test cases through the calculator’s user interface, we instead perform those tests at the service level. To see how this might work, suppose we create a spreadsheet like [Table 16.1](https://www.safaribooksonline.com/library/view/succeeding-with-agile/9780321660534/ch16.html#ch16tab01), where each row represents one test case. The first two columns represent the numbers to be multiplied, the third column is the expected result, and the fourth column contains explanatory notes that will not be used by the test but that make the tests more readable.

**Table 16.1** A spreadsheet showing a subset of the multiplication service tests.



What’s needed next is a simple program that can read the rows of this spreadsheet, pass the data columns to the right service within your application, and verify that the right results occur. Despite this simplistic example where the result is simple calculation, the result could be anything—data updated in the database, an e-mail sent to a specific recipient, money transferred between bank accounts, and so on.

NOTE

Although writing a tool to read a spreadsheet and pass data to specific services within your application is something the programmers on the team could easily write, there are already excellent tools to do this. FitNesse, available at [www.fitnesse.org](http://www.fitnesse.org/), is the most popular such tool.