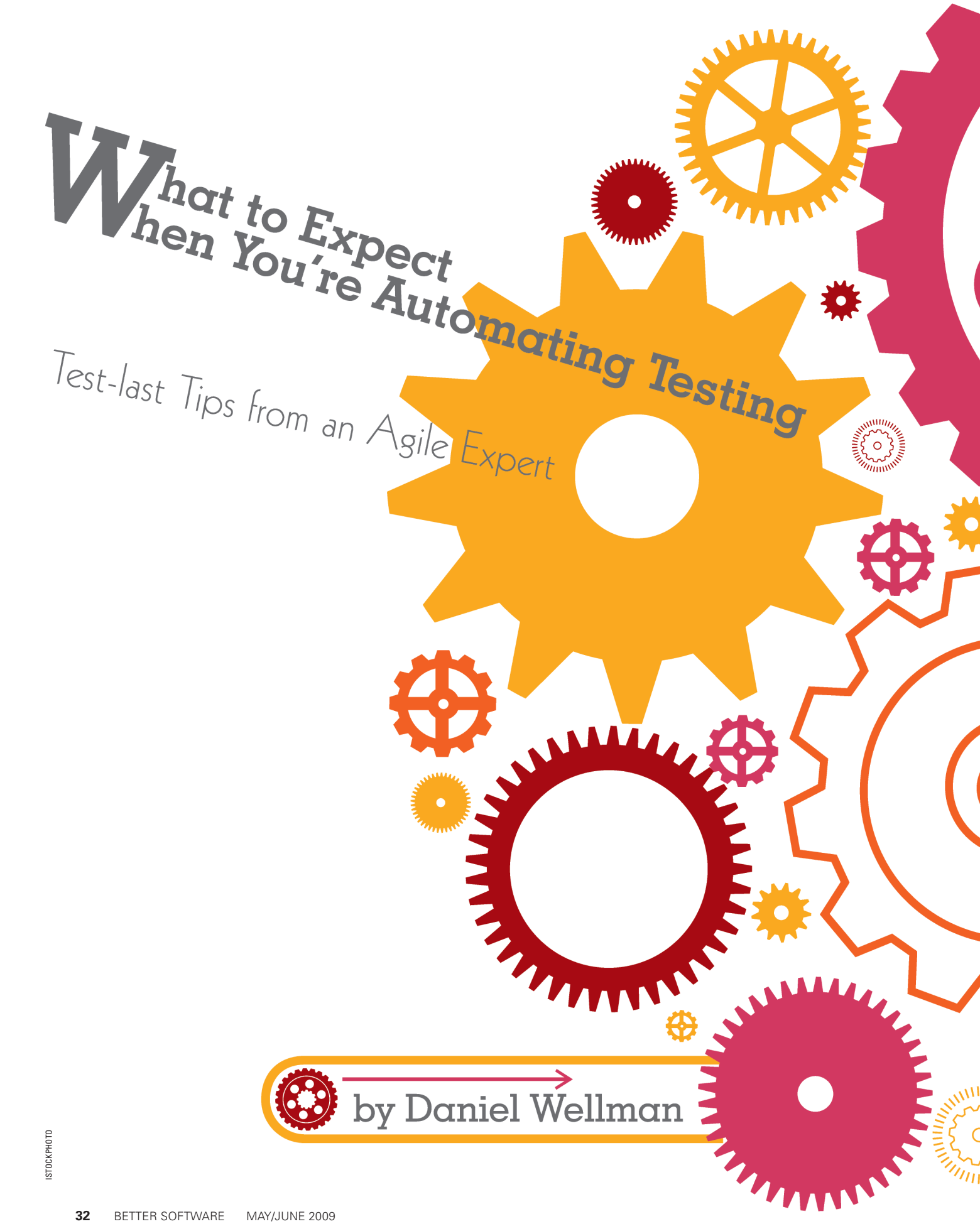
What to Expect When You're Automating Testing

Test-last Tips from an Agile Expert



by Daniel Wellman

ISTOCKPHOTO

Congratulations, you've decided to start writing automated tests for your application. Maybe tests are a new requirement for your team, maybe you've been burned by bugs that keep reappearing, or maybe you were just curious about the buzz surrounding automated testing. However you got to this point, a good suite of automated tests can make your development life more productive and peaceful.

If you are a developer, perhaps you've read an introductory article on testing with JUnit (or NUnit, Test::Unit, or your programming language's flavor of the xUnit test framework), and you understand the syntax and fundamentals of writing tests. But going from test-driving a stack data structure (a typical book example) to testing your living, complex production application can seem like a daunting challenge. In this article, I'll suggest what to start testing in your application, how to get started, and some problems you may encounter along the way.

It's important to note that writing automated tests can be practiced in any software process or methodology, whether it's Scrum, waterfall, RUP, Extreme Programming (XP), or your organization's own custom blend. While XP practitioners write their tests before the production code, this practice can be difficult to start—and not everyone prefers to work in this manner (though I'd encourage everyone to give it a try before dismissing it). Test-last development—testing after the production code is written—is a way that many teams start and practice automated testing.

What Do I Test?

Like starting anything new, it can be difficult to decide how and where to begin adding tests. While sitting down and writing tests for the first code you see may feel productive, there are some strategies to get more immediate value out of your tests. When starting with no existing tests, you want to get the most value out of your time and effort. You want to avoid writing tests for code that never has broken and probably never will break. Here are a few questions I ask to discover some possible starting points:

- Are there any existing or recently fixed bugs?
- What features have you just finished?
- What are you planning to work on next?

Are there any existing or recently fixed bugs?

Tests that expose a bug are the most immediately valuable tests. These demonstrate a real fault in the system, provide feedback for when you have finished fixing the problem (the test passes), and act as an automated alarm if the defect ever gets reintroduced.

If you have just found a bug, write a test that reproduces the failure. Fix the bug, rerun the test to see it pass, and refactor the code to clean it up. This rhythm is the Holy Grail of test-driven development—the "Red-Green-Refactor" cycle.

What do you do if you don't have any bugs (that you know about)? Pick a bug that you just fixed or one that was a real zinger. Write a test for that bug—but here's the catch—roll back the production code to a point where the bug still existed. Now run the test to confirm the test exposes the bug, then restore the production code to the current working state. This step is critical—you want to make sure you write a test that actually catches the bug. When you have code that already works, it's easy to write a test that never actually fails, even if the bug gets reintroduced.

What features have you just finished?

When do you best remember the code you've written? Right after you write it! Use that mental clarity to write tests that exercise core aspects of the feature, using the tests to document how things are supposed to work. Now is the perfect opportunity to write some executable documentation that demonstrates any unusual corner cases of the particular business rule you've just implemented. Then when you need to modify it in six months, you've left some breadcrumbs to remind you of all the particular nuances.

One other practical reason to work on code you just finished is that team members sometimes get grumpy when people change their code or even insinuate that it might be wrong and require testing (perish the thought!). If you're on a team like this, practice introspection and test your own code; once you do it enough, your teammates might take notice.

Another good reason to test new code is that it may be easier to get the business sponsors to buy-in on the effort. If you tell a business user that you want to spend some time defect-proofing a feature, he'll probably like the idea. If you tell him you want to work on some other feature that's not being worked on anymore, he'll probably balk at the idea. Instead, focus on the current features being developed.

What are you planning to work on next?

When you add new features to an application, there is some risk of breaking existing functionality. This is a fact of life, but tests can help. There are a couple of options to consider: add regression tests to prevent introducing new bug, and to understand how some existing code works.

If you're about to start a new piece of functionality, use your current knowledge of the system to figure out what might break and write tests to cover those cases. Once you complete your task, use these tests to ensure you haven't broken anything. This is especially true if you're about to start refactoring code. Make sure you write tests to ensure you don't accidentally change the code's behavior.

But what if you didn't write the code in the first place and don't really know what it's doing? Write tests to demonstrate and discover the code's behavior. In his book *Working Effectively with Legacy Code*, Michael Feathers calls these *characterization tests*. The theory is that if a system is working, the correct behavior doesn't come from some requirements specification document; it comes from whatever the code is doing right then. Characterization tests help ensure that the code's behavior stays consistent after you've made your new changes.

Once you've figured out what to test, you need to think about how to get started.

Getting Started

High-level and low-level tests

For this article, I'll use the term "high level" to describe tests that exercise top-level classes (e.g., Web framework actions) or public APIs, and "low level" for tests that utilize the individual objects or components (e.g., a sales tax calculator) of your system in isolation. The effectiveness of these test types can be measured in terms of *depth*—how many different components are exercised—and *breadth*—the number of different paths or data combinations executed by the test.

High-level tests provide deep depth and narrow breadth: deep depth because they exercise many layers of the system together, but narrow breadth because they normally exercise just a few paths through the code. But this depth comes at a cost. You need to manage a lot of dependencies and setup before each test. Furthermore, high-level test failures may be hard to diagnose—you might get feedback that a record didn't appear in the database, but you won't easily know if it was because of bad input data, a database problem, or some logic error in any of the collaborating objects.

If you have no tests at all, high-level tests that use a real database exercise a lot of the system and can provide confidence that the system is wired together correctly. These tests usually start just below the user interface by accessing the Web framework actions or services directly. If at all possible, avoid testing directly through the GUI since the user

interface typically changes frequently and leads to a lot of incorrect tests and test maintenance.

Low-level tests are the opposite: shallow depth and wide breadth. Low-level tests are easier to set up with a variety of scenarios since you are working with objects in isolation and can more directly specify desired test setup behavior, such as creating test-only objects that always throw exceptions. This makes low-level tests better for isolating behavior and diagnosing failures. However, these tests typically exercise one layer of your application (e.g., business objects or services) and provide shallow depth.

Ultimately, a well-tested system has a combination of both test types: high level to ensure the system is wired correctly and low level to ensure all the business cases are covered.

For most developers, focused low-level tests are the best way to get started. They're quick to write and run, which means they're more likely to be run frequently, and that means more good feedback and a lot of little successes. Taken on their own, these little tests may seem trivial, but put them all together and you've got the beginnings of a regression suite. Build experience writing tests while learning about the properties of good tests. The Pragmatic Programmers publish an excellent book—*Pragmatic Unit Testing*—with editions for Java/JUnit and C#/NUnit, which does a great job teaching the fundamentals.

After you're comfortable writing low-level tests, move on to high-level tests. If you use a database or external systems as part of these tests, you'll need to ensure these are set up in a repeatable fashion before every test. This may require work other than coding, such as creating your own database schema that you can change as part of your tests, figuring out how to start up your own local copy of a dependent server, or creating mock versions of databases or servers.

Once you've gained experience writing both high-level and low-level tests, you'll learn that starting at the lowest level of your application and moving your way up to high-level tests isn't always the right strategy. Writing low-level tests offers quick rewards, but it can be exhausting to climb level after level of your application. Sometimes it's a good strategy to start with a high-level test to ensure the feature is working end to end, then flesh out the details with low-level tests as necessary.

I faced a similar scenario with a developer. We started writing tests for his new feature at the lowest level of the application. We wrote tests, moving up one layer at a time. After a morning of doing this, we were both exhausted and hadn't yet written a high-level test to verify that the feature worked! In retrospect, it would have been better to start with a high-level test; that way we would have had time to focus on other scenarios to test or refactor the code while the task was still fresh in our minds.

```
public void testSave() {
    action = // ... setup code omitted
    action.setFullName("Nigel Tufnel");
    action.execute();
}
```

Listing 1: A bad test

```
public void testSaveShouldCreatePerson() {
    action = // ... setup code omitted
    action.setFullName("Nigel Tufnel");
    String result = action.execute();

    assertEquals(SUCCESS, result);
    assertPersonCreatedWithName("Nigel Tufnel");
}
```

Listing 2: A better test

There's a special kind of high-level test I briefly mentioned earlier that controls the application through the user interface. For a Web application, this means automating a Web browser with a tool like Selenium or Watir (see the StickyNotes for links to tools). For a desktop application, this means using libraries like Abbot or White. If you're just learning how to write tests, starting with automated GUI tests is almost always a mistake. They're difficult to reproduce because it's hard to set up the data, they're slow to execute, and they're the most brittle and costly tests. However, these tools are great for smoke tests used to ensure your application installs and operates correctly end to end. They are not a replacement for a curious—and perhaps devious—tester who can use the application in ways that were never anticipated. For example, it's easy to write a test that fills in invalid form data in a Web page, but a tester might try to click the back button, open a separate copy of the page in another window, and access your application in two windows simultaneously. Instead, start out by automating the simple, repeatable cases just beneath the GUI layer.

Common Problems

Tests aren't catching regression bugs!

Good tests catch bugs; bad tests let them slip by undetected. To ensure your tests are good, make some devious changes in your production code and ensure the tests fail in the way you expect. Invert some boolean conditional tests, do one less iteration in a loop, swap some assignment operators with equality checks (= for ==), don't actually save a record to the database, or include whatever sort of mistake might make sense. Do your tests catch the error? If not, evaluate whether you're missing a test or if one of the tests is missing some key detail.

For example, imagine you're writing a test for a customer management Web application. Listing 1 shows an example of a bad test for a "Create a Person" action—the test never checks to see if the person was actually created. In fact, this test will fail only if the code throws an exception. Listing 2 shows a better test, which verifies the action's result code and ensures that a person was created. Note also that we've created a custom assertion method to make the test easier to read. This method could check a real database or a faster in-memory version specifically used for tests.

I can't change the database since it affects the rest of the company

If you are writing tests that use the database, it's much easier to have your own private instance that you can set up and tear down at will. This also means

```
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index ||  size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

Figure 1: Sample code coverage annotated by EclEmma

that the build machine should have its own instance that it can use, just like each developer. If it's hard to get the database schema set up reliably in an automated fashion, consider versioning your database changes. Add a version table to keep track of the current schema version, and then create all your changes in incremental SQL scripts that update the schema version after they run. This allows you automatically to re-create or update any database, whether it's the local copy on your workstation or in production. Ruby on Rails uses this technique, as do tools like dbdeploy and migratordotnet.

If you can't get your own instance of the database, there are some techniques you can use to help isolate yourself from the rest of the company (but fight tooth and nail for your own private instance or schema, otherwise your build might occasionally fail because someone else botched the schema). In his book *xUnit Test Patterns*, Gerard Meszaros describes several options for using reusable database fixtures. The premise is that your tests either insert unique data on every run or depend on some data that is always expected to be present. One pattern starts a transaction at the beginning of the test, exercises the system, and then rolls back at the end of the test, preventing any changes from persisting and altering the test database.

How can my team measure its progress?

Code coverage is a measure of how much production code your automated tests exercise, stated as a percentage (for example, "65 percent of my production code is exercised by tests"). While code coverage alone is not a useful metric to determine how safe your application is from accidental regressions (there might be tests, but they might be bad tests), it is a great motivator and teaching aid. Since you're starting out, your coverage number will be low, so set a goal to always have code coverage increase rather than targeting an arbitrary percentage.

Code coverage tools like EclEmma can display production code coverage in your IDE by coloring tested lines of code green and untested lines red. There is no underestimating the "wow" factor of seeing code you've written turn green or red, and it provides great immediate feedback. See Figure 1 for an example.

It's hard to test my objects in isolation

One of the most difficult aspects of starting testing can be getting your code into a test harness—a repeatable configuration of tests with several different scenarios. If your objects under test talk directly to other difficult-to-control components, use dependency injection (see the StickyNotes for a link) to pass in your own test-specific versions. For example, to avoid sending emails to your operations staff every time you run a unit test that verifies a system outage procedure, pass in a fake mail server connection and ensure that a message gets sent via the fake server. Your test will be much more repeatable, and you won't get nasty letters from operations.

By writing more tests, you'll start to learn how to write production code that is easier to test in isolation. However, it won't happen overnight, and it probably will mean making some changes to your existing code to expose testing hooks. Think of these as small steps on the way to a better design. If you have a class that is hard to test, try making a testing-specific subclass that overrides the problematic behavior. It may feel strange to change your production code in awkward ways just to make testing easier, but if you can use it to write automated tests that provide a safety net, you may be able to change your design to remove that test-specific class entirely. Again, see Feathers's excellent *Working Effectively with Legacy Code* for a wealth of techniques for wrangling code into a more testable state.

One Step at a Time...

Just as writing clean code takes discipline, so, too, does writing good automated tests. It's a rewarding practice, but be prepared: It's going to be difficult at first. Start out slow—small, focused low-level tests. Set realistic goals for yourself, and celebrate your successes, such as your first test that runs in the automated build, the first time a test catches a regression bug, or the first test that uses the database. Along the way, verify that your tests are really delivering value—break the production code and ensure a test catches the error.

And, when you become comfortable with writing tests last, I recommend at least trying to write your tests before the production code. I personally find the process of working in small steps with frequent feedback a rewarding and energizing experience. But, some developers love it and some hate it, so decide for yourself. Rather than debating the merits of Test First versus Test Last, let's all celebrate that a good test suite provides us confidence that our code is doing what we expect, and that means we write code more confidently and sleep easier at night.

That sounds like a good life to me. **{end}**