Writing Maintainable Automated Acceptance Tests

Dale H. Emery

dale@dhemery.com

http://dhemery.com

This article was originally presented, under a slightly different name, as part of the Agile Testing Workshop at Agile Development Practices 2009. I've made a few minor changes.

Test Automation is Software Development

Test automation is software development[1]. This principle implies that much of what we know about writing software also applies to test automation. And some of the things we know may not be apparent to people with little or no experience writing software.

Much of the cost of software development is maintenance—changing the software after it is written. This single fact accounts for much of the difference between successful and unsuccessful test automation efforts. I've talked to people in many organizations that attempted test automation only to abandon the effort within a few months. When I ask what led them to abandon test automation, the most common answer is that the tests quickly became brittle and too costly to maintain. The slightest change in the implementation of the system—for example, renaming a button—breaks swarms of tests, and fixing the tests is too time consuming.

But some organizations succeed with test automation. Don't they experience maintenance costs, too? Of course they do. An important difference is that where unsuccessful organizations are surprised by the maintenance costs, successful organizations expect them. The difference between success and failure is not the maintenance costs per se, but whether the organization expects them. Successful organizations understand that test automation is software development, that it involves significant maintenance costs, and that they can and must make deliberate, vigilant effort to keep maintenance costs low.

The need to change tests comes from two directions: changes in requirements and changes in the system's implementation. Either kind of change can break any number of automated tests. If the tests become out of sync with either the requirements or the implementation, people stop running the tests or stop trusting the results. To get the tests back in sync, we must change the tests to adapt to the new requirements or the new implementation.

If we can't stop requirements and implementations from changing, the only way to keep the maintenance cost of tests low is to make the tests adaptable to those kinds of changes.

---

[1] I learned this idea from Elisabeth Hendrickson, an extraordinary tester.

Developers have learned—often through painful experience—that two key factors make code difficult to change: Incidental details and duplication. You don't want to learn this the hard way.

Acceptance Tests and System Responsibilities

An acceptance test investigates a system to determine whether it correctly implements a given responsibility. The essence of an acceptance test is the responsibility it investigates, regardless of the technology used to implement the test.

Suppose we are testing a system's account creation feature. The `create` command creates a new account, given a user name and a password. One of the account creation feature's responsibilities is to *validate passwords*. That is, it must accept valid passwords and reject invalid ones. To be valid, a password must be from 6 to 16 characters long and include at least one letter, at least one digit, and at least one punctuation character. If the submitted password is valid, the `create` command creates the account and reports *Account Created*. If the password is invalid, the `create` command refrains from creating the account and reports *Invalid Password*.

That's the essence of the responsibility. No matter how the system is implemented—whether as a web app, a GUI app, a set of commands to be executed on the command line, or a guy named Bruce wielding a huge pair of scissors to snip off the fingers of anyone who submits an invalid password—the system must implement that responsibility.

Incidental Details

Listing 1 shows a poorly written automated acceptance test[2] for the `create` command's password validation responsibility.

[2] The examples presented here run within Robot Framework, an increasingly popular test automation tool that allows you to write tests in a variety of formats. As you will see, Robot Framework offers techniques to write clear, maintainable tests. Robot Framework is free and open source. See http://code.google.com/p/robotframework/ for further information.

Listing 1: A poorly written acceptance test

```
** Test Cases **
The create command validates passwords
    ${status}=  Run  ruby app/cli.rb create fred 1234!@$^
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred abcd!@$^
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred abcd1234
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred !2c45
    Should Be Equal  ${status}  Invalid Password
    ${status}=  Run  ruby app/cli.rb create fred !2c456
    Should Be Equal  ${status}  Account Created
    ${status}=  Run  ruby app/cli.rb create fred !2c4567890123456
    Should Be Equal  ${status}  Account Created
    ${status}=  Run  ruby app/cli.rb create fred !2c45678901234567
    Should Be Equal  ${status}  Invalid Password
```

This test has numerous problems, the most obvious being that it is hard to understand. We can see from the second line—the name of the test—that it tests the `create` command's validation responsibility. But it's hard to make sense of the details of the test among the flurry of words and "syntax junk" such as dollar signs and braces.

With a little study we can pick out the passwords—such as `1234!@$^`. And with a little more study we might notice that some passwords lead to a status of `Invalid Password` and others lead to `Account Created`. On the other hand, we might just as easily not notice that, because the connection between passwords and statuses is buried among the noise of the test. What do dollar signs, braces, and the words `Run`, `Ruby`, and `fred` have to do with passwords and validation? Nothing. Those are all *incidental details*, details required only because of the way we've chosen to implement the system and the test.

Incidental details destroy maintainability. Suppose our security analysts remind us that six-character passwords are inherently insecure. So we change one of the key elements of the responsibility, increasing the minimum length of a password from six to ten. Given this change in requirements, what lines of this test would have to change, and how? It isn't easy to see at a glance.

Let's consider a more challenging requirements change. We want system administrators to be able to configure the minimum and maximum password length for each instance of the system. Now which lines of the test would have to change? Again, the answer isn't easy to see at a glance.

That's because the test does not clearly express the responsibility it is testing. When we cannot see the essence of a test, it's more difficult and costly to understand how to change the test when the system's responsibilities change. Incidental details increase maintenance costs.

So the first step toward improving maintainability is to hide the incidental details, allowing us to more easily see the essence of the test. In this test, most of the details are about how to invoke the `create` command. This system is implemented as a set of command line commands, written in the Ruby programming language. The first highlighted line in the test tells Robot Framework to run the computer's Ruby interpreter, telling it to run the `app/cli.rb` (the system we're testing), and telling *it* in turn to run its `create` command with the user name `fred` and the password `1234!@$^`. And at the end of it all, Robot Framework stuffs the `create` command's output in a variable called `${status}` Whew!

The highlighted second line is easier to understand—it compares the returned status to the required status `Invalid Password`—but it's awkwardly worded and includes distracting syntax junk, a form of incidental detail.

Robot Framework allows us to extract details into *keywords*, which act like subroutines for our tests. A keyword defines how to execute a step in an automated test.

So let's create a keyword to hide some of the incidental details.

One useful approach is to ask yourself: How would I write that first step if I knew nothing about the system's implementation? Even if I knew nothing about the system's implementation, I know it has the responsibility to create accounts—that's the feature we're testing, after all. So I know it will offer the user some way to create an account. *Create Account*, then, is an essential element of the system's responsibilities. I also know (from other requirements) that in order to create an account, the user must submit a user name and a password.

Given all of that, I might write the test step like this:

```
Create Account fred 1234!@$^
```

I still have some concerns with this test step[3], but I'll deal with those later.

Now let's look at the second highlighted step. It seems to be verifying that the `create` command returned the appropriate status: *Invalid Password*. How might I rewrite this step if I knew nothing about the system's implementation? Here's one possibility:

```
Status Should Be Invalid Password
```

So together, those two steps now look like this:

```
Create Account fred 1234!@$^
Status Should Be Invalid Password
```

That's much clearer. Without all of the incidental details, it's easier to spot the connection between the two lines: The system must tells us that the given password is invalid.

[3] My first concern: What's `fred` doing there? That's a user name. I've given a user name because the *Create Account* command (however it's implemented) requires a user name. Still, the user name has no bearing on password validation, so it's extraneous for this test. My second concern is that it isn't immediately obvious what's significant about that specific password.

Now if we try to run the test, it will fail, because Robot Framework doesn't know the meaning of `Create Account` or `Status Should Be`. We haven't defined those keywords yet. Let's do that now:

Listing 2: Keywords to create an account and check the status

```
** Keywords **
Create Account ${user_name} ${password}
    ${status}=  Run  ruby app/cli.rb create ${user_name} ${password}
    Set Test Variable   ${status}

Status Should Be ${required_status}
    Should Be Equal  ${status}  ${required_status}
```

The highlighted line introduces a new keyword called `Create Account`, and describes it as taking two pieces of information as input—a user name and a password. The next two lines tell Robot Framework how to execute the keyword. Notice that the first indented line looks a lot like the first highlighted line of our original test. This is where we hid the incidental details.

You may also notice that we introduced yet more syntax junk, yet more dollar signs and braces. How is this an improvement? The benefit is this: By extracting all of the incidental details out of the test steps and into the keyword, we've cleaned up our test steps, making them easier to understand. The benefit becomes more apparent if we rewrite *all* of our test steps using the new keywords:

Listing 3: The test rewritten to remove incidental details

```
** Test Cases **
The create command validates passwords
    Create Account fred 1234!@$^
    Status Should Be Invalid Password
    Create Account fred abcd!@$^
    Status Should Be Invalid Password
    Create Account fred abcd1234
    Status Should Be Invalid Password
    Create Account fred !2c45
    Status Should Be Invalid Password
    Create Account fred !2c456
    Status Should Be Account Created
    Create Account fred !2c4567890123456
    Status Should Be Account Created
    Create Account fred !2c45678901234567
    Status Should Be Invalid Password
```

Now our test reads much more cleanly. At the expense of a little bit of syntax awkwardness in the keyword definition, we've gained a lot of clarity in the test. It's a tradeoff well worth making.

Duplication

So far we've improved the test noticeably by extracting incidental details into reusable keywords. But there are still problems. One, mentioned earlier, is the troublesome `fred` in every other step. A bigger problem is duplication. Every pair of lines submits an interesting password and verifies that the system emits the appropriate status message. From one pair to the next, only two things change: the password and the desired status. Everything else stays the same. Everything else is duplicated from one pair to the next.

Duplication destroys maintainability. Suppose our usability analysts remind is that none of our other systems ask users to create an account. Instead, they ask users to *register*. So the language of this system—*create account*—is inconsistent with others. The usability analysts insist, and now we need to change our system's terminology.

One possibility is to simply change the name of the command line command from *create* to *register*, and leave our tests the way they are. If we were to do that, then every time we tried to talk about the acceptance tests with the business folks, we would have to translate between the language of the tests and the language of the business. That path leads to confusion.

To keep the language consistent, it would be better to change the tests to use the common terminology. This is where duplication rears its ugly head. We have to scan all of our tests, identify every mention of *create*, and change it to *register*. With our revised test, that's not especially onerous. We mention *create* only ten times—eight[4] times in the test and twice in the keywords. But imagine if we had hundreds of tests[5]. Duplication increases maintenance costs.

Duplication often signals that some important concept lurks unexpressed in the tests. That's especially true when we duplicate not just single steps, but *sequences* of steps. In our test, we duplicate pairs of steps—one step in each pair creates an account with a significant password, and the next checks to see whether the system reported the correct status.

Consider the first two steps in Listing 3. What do they do? What is the essence of those two steps? Taken together, the verify that the `create` command rejects the password `1234!@$^`. How about steps nine and ten? Those two steps verify that the `create` command accepts the password `!2c456`. *Accept* and *reject*. Those concepts are the essence of the responsibility we're testing, yet they're cowering in the shadows of our test steps.

Let's make the concepts explicit by creating two new keywords[6]:

---

[4] I originally counted only seven occurrences, missing the one in the name of the test. That's another challenge with duplication. When you have to change all of the occurrences, it's easy to miss some.

[5] Or count the number of *creates* in the original test in Listing 1. Notice that by extracting incidental details from the test into a keyword, we've also reduced the number of changes we'd have to make if we switched from *create* to *register*. Bonus!

[6] Notice that these new keywords do not depend on any implementation details of the system. They're built entirely on our lower-level keywords. If the implementation details change, these keywords will continue to be valid, and will not require change. Also, I've changed the user name from `fred` to `arbitraryUserName` to help readers understand that, for the purpose of this keyword, there's nothing special about this user name.

Listing 4: Keywords for accepting and rejecting passwords

```
** Keywords **
Accepts Password ${valid_password}
    Create Account arbitraryUserName ${valid_password}
    Status Should Be Account Created

Rejects Password ${invalid_password}
    Create Account arbitraryUserName ${invalid_password}
    Status Should Be Invalid Password
```

These keywords not only allow us to rewrite our test, they also define the *meaning* of accepting and rejecting passwords. To accept a password means that when we try to create an account with the password, the system reports that the account has been created[7]. To reject a password means that when we try to create an account with that password, the system reports that the password is invalid.

Now we can rewrite our test to reduce the duplication, and also to directly express the essential responsibility of accepting and rejecting passwords[8]:

Listing 5: Test rewritten to reduce duplication

```
** Test Cases **
The create command validates passwords
    Rejects Password 1234!@$^
    Rejects Password abcd!@$^
    Rejects Password abcd1234
    Rejects Password !2c45
    Accepts Password !2c456
    Accepts Password !2c4567890123456
    Rejects Password !2c45678901234567
```

By analyzing duplication in the test, we identified two essential system concepts—the system accepts valid passwords and rejects invalid ones. By defining keywords, we *named* those concepts. Then we rewrote the test to refer to the concepts by name. By putting names to those concepts, and using the names throughout the test, we made the test more understandable and thus more maintainable.

---

[7] In the real world, accepting a password means more than simply reporting that the account was created. In addition, the system must of course actually create the account. A complete test would verify those essential results, and not simply take the system's word that it created the account. Systems lie! I've omitted those details to keep the example small enough to talk about.

[8] There is still duplication here. We could reduce it further, perhaps by creating a `Rejects Passwords` keyword that takes a *list* of invalid passwords and checks whether the system rejects each one. Would that make the test clearer or more maintainable? My guess is no, but it's worth considering. Try it for yourself and see.

Naming the Essence

Now that the test more clearly talk about accepting and rejecting passwords, one last bit of unclarity becomes more apparent. As we look at each invalid password, it isn't immediately obvious what's invalid about it. And what about the valid passwords? Why do we test two passwords? And why those two? What's so special about them? With time you could figure out the answers to those questions. But here's a key point: *Any* time spent puzzling out the meaning and significance of a test is maintenance cost. This may seem like a trivial cost, but multiply that by however many tests you need to change the next time someone changes a requirement. As many of my clients have discovered, these "trivial" maintenance costs add up, and they kill test automation efforts.

As I designed the test, I chose each password for a specific purpose. The essence of each password is that it tells me something specific that I want to know about the system. Take the password `1234!@$^` as an example. I chose this password because it is missing one of the required character types: it contains no letters. The essence of this password is that it lacks letters.

I'd like to give that essence a name. Robot Framework offers a feature to do that: variables. I can create a variable, give it an expressive name, and assign it a value that embodies that name. Here's how to create a variable:

```
** Variables **
${aPasswordWithNoLetters}        1234!@$^
```

Now I can use that variable in my test. In the interest of space, let's assume that I've created variables for all of the passwords, each named to express its essence, its significance in the test[9]:

Listing 6: Test rewritten to name significant values

```
** Test Cases **
The create command validates passwords
    Rejects Password ${aPasswordWithNoLetters}
    Rejects Password ${aPasswordWithNoDigits}
    Rejects Password ${aPasswordWithNoPunctuation}
    Rejects Password ${aTooShortPassword}
    Accepts Password ${aMinimumLengthPassword}
    Accepts Password ${aMaximumLengthPassword}
    Rejects Password ${aTooLongPassword}
```

Now the test is nearly as clear as we can make it. I'll take one more step, and break the test into multiple tests, each focused on a particular element of password validation:

[9] Yes, using variables does require us include distracting dollar signs and braces in our test. Does the clarity of the names outweighs the distraction of the syntax junk?

Listing 7: Test rewritten to name significant values

```
** Test Cases **
Rejects passwords that omit required character types
    Rejects Password    ${aPasswordWithNoLetters}
    Rejects Password    ${aPasswordWithNoDigits}
    Rejects Password    ${aPasswordWithNoPunctuation}

Rejects passwords with bad lengths
    Rejects Password    ${aTooShortPassword}
    Rejects Password    ${aTooLongPassword}

Accepts minimum and maximum length passwords
    Accepts Password    ${aMinimumLengthPassword}
    Accepts Password    ${aMaximumLengthPassword}
```

Now when I read these tests, I can understand *at a glance* the meaning and significance of each test and each step. Each important requirements concept is expressed clearly, and expressed once.

Now suppose we change the requirements for minimum and maximum password length. Because each requirements concept is expressed clearly in the tests, I can quickly identify which tests would have to change. And because each concept is defined once—and given a name—I can quickly change the tests.

Putting the Tests to the Test: A Major Implementation Change

So all of our work has made the tests more adaptable to requirements changes. But how about implementation changes? To find out, let's change a few implementation details of the system and see how our tests fare. By "a few implementation details," I mean let's rewrite entire system as a web app. Now, instead of typing the `create` command on the command line, users visit the account creation web page, type the user name and password into text fields on a web form, and click the *Create Account* button. And the system, instead of printing the status to the command line, forwards the user to a web page that displays the status.

The big question: How would our tests have to change?

Remember that earlier we hid many incidental details inside keywords—`Create Account` and `Status Should Be`. Those keywords still contain the arcane steps to issue commands on the command line. So clearly those keywords will have to change. Let's rewrite those keywords to invoke the web app instead of the command line app[10]:

[10] These steps use another tool, Selenium, to drive a web browser and to interact with the web app. To start up Selenium and a browser to run the tests, and to shut down Selenium and the browser after the tests, we have to add another few geeky lines to our tests. In the interest of staying focused, I won't include those lines here, but it's a grand total of eleven lines of test code.

Listing 8: Rewriting keywords to invoke the new web app

```
Create Account ${username} ${password}
    Go To    http://localhost:4567/create
    Input Text    username    ${username}
    Input Text    password    ${password}
    Submit Form

Status Should Be ${required_status}
    ${status}=          Get Text    status
    Should Be Equal     ${required_status}  ${status}
```

Okay, so we've changed the keywords that directly interact with the system. And we've added another eleven lines of test code as described in the footnote. What's next? What else do we have to change?

Nothing. We're done.

We've changed a few lines of test code, and our tests now run just fine against a new implementation of the system using entirely changed technology.

Meanwhile, Back in the Real World

In the real world, you will likely have more work to do to respond to such a major implementation change. For example, you will have to change more than two keywords. But if you've created low-level keywords that isolate the rest of your test code from the details of how to interact with the system, you will have to change only those low-level keywords. The tests themselves continue to work, unchanged.

And real world implementation changes may require more radical changes in the tools you use to run the tests.

But even when that's true, you can still use modern open source testing tools[11] to remove duplication from your tests, and to write tests that clearly and directly express the essence of the system responsibilities they are testing.

The bottom line is this: If you write automated tests so that they express system responsibilities clearly and directly, and if you remove duplication, you will significantly reduce the maintenance costs that arise from both changes in requirements and changes in system implementation. That could mean the difference between successful test automation and failure.

[11] Though these tests use Robot Framework's particular test format, many other open source test tools—Fit, FitNesse, and Cucumber being among the more popular—offer similar ways to express the essence of your tests and to hide implementation details.