Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

9

Automating validation without changing specifications

After we refine the specification of a feature, it becomes a clear target for implementation and a precise way to measure when we've finished. The refined specification also allows us to check in the future whether our system still has the required functionality, every time we change it. Because of the level of detail that we get from illustrating the specifications using examples, it becomes impossible to manually run all the checks within short iterations, even for midsize projects. The solution is obvious: We have to automate as many of these checks as possible.

The right automation for validating specifications with examples is quite different from traditional test automation in software projects. If we have to significantly change the specification while automating it, the telephone game starts all over again and the value of refining the specification is lost. Ideally, we should automate the validation processes for our specifications without distorting any information. This introduces an additional set of challenges on top of the usual test automation issues.

In this chapter I present advice on how to get started with automating validation of specifications without changing them and how to control the long-term maintenance costs of automation. I then cover the two areas that caused the most problems for the teams I interviewed: automating user interfaces and managing data for automated test runs. The practices I present here apply to any tool. I won't be discussing individual tools, but if you're interested in researching more about that topic visit http://specificationbyexample.com and download additional articles. Before we start with that, I'll address a question that's often raised in mailing lists and online forums—do we need this new type of automation at all?

Because automation is a highly technical problem, this chapter is going to be more technical than the others. If you're not a programmer or an automation specialist, you might find it hard to follow some parts. I suggest you read the first two sections and then skip the rest of this chapter. You won't lose anything that interests you.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How does this work?

All the most popular tools for automating executable specifications work with two types of artifacts: specifications in a human-readable form and automation code in a programming language. Depending on the tool, the specifications are in plain text, HTML, or some other human-readable format. The tools know how to extract inputs and expected outputs from those specifications, so that they can pass that on to the automation code and evaluate whether the expectations matched the results. The automation code, with some tools called *fixtures* or *step definitions*, calls the application APIs, interacts with the database, or executes actions through the application user interface.

The automation code depends on the specifications but not the other way around. That's how these tools allow us to automate the validation of specifications without changing them.

Some tools require teams to store examples in programming language code and produce human-readable specifications from that. Technically they achieve the same effect, but such tools effectively prevent anyone who's not proficient with programming language code from writing or updating a specification.

licensed by YEONG SHENG TAN 2440622

Is automation required at all?

The long-term maintenance cost of executable specifications is one of the biggest issues that teams face today when implementing Specification by Example. The tools for automating executable specifications are improving rapidly, but they're still far from more established unit-testing tools in terms of the ease of maintenance and development tool integration. Automation also introduces additional work for the team. This frequently causes discussions as to whether automation is required at all and whether it costs more than it's worth.

The argument against automation is that it increases the amount of work to develop and maintain software and that teams can get a shared understanding of what needs to be done by illustrating using examples and not automating them at all. Phil Cowans said that this view neglects the long-term benefits of Specification by Example:

> "It feels like you're writing twice as much code to build the same functionality. But the number of lines of code is probably not the limiting factor in your development process, so this is quite naive. You're not taking into account the fact that you spend less time maintaining what you've already built or dealing with miscommunication between your testing and development."

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Automation in general is important for larger teams because it ensures that we have an impartial, objective measurement of when we're finished. Ian Cooper has a nice analogy for this:

> When I'm tired, and I wash the dishes, I don't want to dry the dishes. I've washed everything; it's almost done. Missus looks at that and doesn't think that I'm done. For her, "done" is when the dishes have been dried and put away and the sink is clean. It [automation] is forcing developers to be honest. They can't do just the bits that interest them.

Automation is also very important long term because it enables us to check more cases more frequently. Pierre Veragen said that the managers in his company quickly understood this value:

> All of the sudden the managers realized that instead of having something that checks two or three numbers during a test, now we had something that checked 20 or 30 numbers more and we could pinpoint problems easier.

Some teams reduce the cost of automation by moving it to technical tools. While I was preparing for interviews for this book, I was very surprised that Jim Shore, one of the thought leaders in the agile community and an early adopter of Specification by Example, actually gave up on automating executable specifications because of that cost.[1] Shore wrote that in his experience, illustrating using examples brings more value than automating validation without changing the specifications:

> My experience with FIT and other agile acceptance testing tools is that they cost more than they're worth. There's a lot of value in getting concrete examples from real customers and business experts, not so much value in using "natural language" tools like FIT and similar.

From my experience, this push back on automation of executable specifications can save time in the short term but prevents the team from getting some of the most important long-term benefits of Specification by Example.

[1] Parts of our email conversation are published online. See http://jamesshore.com/Blog/Alternatives-to-Acceptance-Testing.html, http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html, and http://gojko.net/2010/03/01/are-tools-necessary-for-acceptance-testing-or-are-they-just-evil/. You'll also find the links to opinions of other community members on this topic in those articles. I strongly suggest reading those articles, especially Shore's discussion about alternatives to automation of executable specifications.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

When deciding whether to automate the validation of specifications using a technical tool or one for executable specifications, think about which benefits you want to get out of it. If we automate examples with a technical tool, we get easier automation and cheaper maintenance but lose the ability to use them for communication with business users later. We get very good regression tests, but the specifications will be accessible only to developers. Depending on your context, this might or might not be acceptable.

Automating validation of specifications without changing them is a key part of getting to living documentation. Without it, we can't guarantee the correctness of human-readable specifications. For many teams, the long-term benefit of Specification by Example comes from living documentation. Instead of dropping automation that preserves the original specifications, we can work on controlling the cost of maintenance. You'll find many good techniques that teams used to reduce long-term maintenance costs in the "Managing the automation layer" section later in this chapter, as well as in chapter 10.

Starting with automation

Automated validation of executable specifications is quite different from unit testing, recorded and scripted functional automation that developers and testers are often used to. Automating something but keeping it human readable requires teams to learn how to use new tools and to discover the best way of hooking the automation into their system. Here are some good ideas on how to start to implement the automation process and a look at some mistakes the teams I interviewed commonly made while doing that.

To learn about tools, try a simple project first

When: Working on a legacy system

Several teams used a simple project or a spike to learn how to use a new automation tool. If you have a small, relatively isolated piece of work in the pipeline, that might be a good strategy.

A small project minimizes risk and helps you focus on learning how to use a tool instead of dealing with complex integrations and business rules.

This approach is especially effective if you want to implement Specification by Example at the same time as moving to an agile development process.

At uSwitch, they took this approach when introducing Cucumber, another popular tool for automation of executable specifications. They got the entire development team to start converting

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

existing tests to the new tool. This gave everyone on the team some experience with the new tool quickly. Stephen Lloyd says that it also showed them the power of executable specifications:

> We realized that there is a whole extra level of testing that needed to be done, and that testing at the end of the cycle didn't make sense.

A mini-project gives you a way to learn and practice new skills without much risk to ongoing development, so it might be a lot easier to get approval for that than to experiment with something that's much riskier.

It might be a good idea to get the result reviewed by an external consultant. Once you have done something that can be reviewed, external consultants will be able to provide much more meaningful feedback and discuss better solutions. By then, your team would also have had a chance to play with the tool and go over some basics, so they'll be able to understand more advanced techniques and get more value out of the consultant's time.

Plan for automation upfront

Teams that work on systems that haven't been designed up front for automated testing should expect their productivity to actually drop at first when they start automating executable specifications.

Even without learning how to use a new tool, automated validation initially adds a significant overhead on a project. Automation is front loaded—a disproportional amount of work has to be done when you start automating. This includes creating basic automation components, deciding on the best format of executable specifications and the best way to integrate with the system, resolving issues with test stability and dedicated environments, and many others. We deal with most of those issues in this chapter and in chapter 10, but for now it's important to understand that productivity will drop when you start automating.

Once those issues are resolved and the structure of the executable specifications stabilizes, we can reuse the basic automation components when working on new specifications. As the project matures, the automation effort drops significantly and the productivity surges.

In several projects, developers didn't consider this when estimating the effort required to implement a story. But when the automation is just starting, it might take much more effort to automate an executable specification than to implement the required change in production code.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Make sure to plan for a drop in productivity up front. The team should undercommit on scope to allow for automation to be done within an iteration.

Unless this planning is done, automation is going to overrun into the next iteration and interrupt the flow. This is one of the key lessons André Brissette, the Talia product director at Pyxis Technologies, learned:

> If I had to do it all over again, from the start I would be more directive about the need to write tests [executable specifications]. I knew that it was a challenge for a team to write these kinds of tests, so I was patient. Also I could have let more room in the sprint [iteration] for making tests. We'd start, and then I'd talk about executable specifications and the team would say, "We don't really have the time for this learning curve because we're pretty loaded in this sprint." In fact, one of the reasons why they were pretty loaded was because I was filling the sprint with a lot of features. Because of this, it started slowly, and it took many iterations before having a decent set of specifications.
>
> Maybe it would pay off more to break that wall at the beginning and decide to do fewer features at the start. That's the choice that I would make the next time. When you spread the integration of that kind of practice over a long time, you have the cost of it without having the benefit. It ends up being more expensive.

One idea to ensure that the initial automation effort is planned for is to consider the automation toolkit as a separate product with its own backlog and then devote a certain percentage of team's time for that product. Just to make things clear, both the primary product and the automation framework should still be developed and delivered by the same team, in order to ensure that the team is familiar with the automation later. I advise treating it as a separate product purely to limit the impact on the delivery of the primary work.

Don't postpone or delegate automation

Because of the automation overhead, some teams delayed it. They described specifications with examples and then wrote code, leaving the automation for later. This seems to be related to projects where the development and test automation teams were separate or where external consultants were automating tests. This caused a lot of rework and churn.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Developers were marking user stories as done without having an objective automated criterion for that. When the acceptance tests finally got automated, problems often popped out and stories had to be sent back for fixing.

When automation is done along with implementation, developers have to design the system to make it testable. When automation is delegated to testers or consultants, developers don't take care to implement the system in a way that makes it easy to validate. This leads to more costly and more difficult automation. It also causes tests to slip into the next iteration, interrupting the flow when problems come back.

Instead of delaying automation of executable specifications because of the overhead, deal with the automation problems so that the task becomes easier to do later.

Postponing automation is just a local optimization. You might get through the stories quicker from the initial development perspective, but they'll come back for fixing down the road. David Evans often illustrates this with an analogy of a city bus: A bus can go a lot faster if it doesn't have to stop to pick up passengers, but it isn't really doing its job then.

Avoid automating existing manual test scripts

Creating an executable specification from existing manual test scripts might seem to be a logical thing to do when starting out. Such scripts already describe what the system does and the testers are running them anyway, so automation will surely help, right? Not really—in fact, this is one of the most common failure patterns.

Manual and automated checks are affected by a completely different set of constraints. The time spent preparing the context is often a key bottleneck in manual testing. With automated testing, people spend the most time trying to understand what's wrong when a test fails.

For example, to prepare for a test script that checks user account management rules, a tester might have to log on to an administrative application, create a user, log on as that new user to the client application, and change the password after first use. To avoid doing this several times during the test, a tester will reuse the context for several manual scripts. So she would create the user once, block that account and verify that the user can't log on, reset the password to verify that it is reenabled, and then set some user preferences and verify that they change the home page correctly. This approach helps the tester run through the script more quickly.

With automated testing, the time spent on setting up the user is no longer a problem. Automated tests generally go through many more cases than manual tests. When

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

they run correctly, nobody is really looking at them. Once an automated test fails, someone has to go in and figure out what went wrong. If the test is described as a sequence of interdependent steps, it will be very hard to understand what exactly caused the problem, because the context changes throughout the script.

A single script checking 10 different things is more likely to fail than a smaller and more focused test, because it's affected by lots of different areas of code. In the previous example with user account management, if the password reset function stops working, we won't be able to set the user preferences correctly. A consequence is that the check for home page changes will also fail. If we had 10 different, smaller, focused, and independent tests instead of one big script, a bug in the password reset function wouldn't affect the test results for user preferences. That makes tests more resilient to change and reduces the cost of maintenance. It also helps us pinpoint the problems more quickly.

Instead of plainly automating manual test scripts, think about what the script is testing and describe that with a group of independent, focused tests. This will significantly reduce the automation overhead and maintenance costs.

Gain trust with user interface tests

When: Team members are skeptical about executable specifications

Many tools for automating executable specifications allow us to integrate with software below the user interface. This reduces the cost of maintenance, makes the automation easier to implement, and provides quicker feedback (see the "Automate below the skin of the application" section later in this chapter).

But business users and testers might not trust such automation initially. Without seeing the screens moving with their own eyes, they don't believe that the right code is actually being exercised.

When you're starting out with Specification by Example, if your team members doubt the automation, try to execute the specifications through the user interface. Note that you shouldn't change the specifications to describe the user interface interactions, but you can hide those activities in the automation layer.

Getting the business users to trust executable specifications was one of the key challenges on the Norwegian Dairy Herd Recording System project. Børge Lotre, a manager at Bekk Consulting who worked on that project, says that they built the trust gradually as the number of checks in executable specifications increased:

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

> They [business users] used to insist on manual testing in addition to Cucumber. I think they are seeing the value of the Cucumber tests because they are not capable of [manually] testing the old requirements each time we add new functionality.

Executable specifications should generally be automated through the user interface only as a last resort, because user interface automation slows down feedback and significantly increases the complexity of the automation layer. On the other hand, executing automated specifications through a user interface might be a good solution to gain trust from the nontechnical users initially. Make the automation layer flexible so that you can switch to integrating below the skin of the application later.

Running executable specifications through the user interface is also a good option when working with a legacy system that doesn't have a clean integration API (in which case the only way to automate tests is end to end, starting with the front-end user interface and validating the results either in the database or by using the user interface again). Making the automation layer flexible is a good idea in this case as well, because you'll probably want to move it below the user interface once the architecture becomes more testable.

Apart from gaining trust, allowing people to see the application screens during automated testing sometimes helps them think about additional examples.

According to my experience and in many of the case studies for this book, executing tests through a user interface doesn't scale well. You might want to reduce the number of tests executed through the UI later, once you gain the trust of the stakeholders.

If you decide to automate specifications through a user interface, apply the ideas described in the "Automating user interfaces" section later in this chapter to get the most out of it and to ensure that you'll be able to move the automation below the user interface when needed.

Managing the automation layer

Controlling the cost of maintenance for a living documentation system is one of the biggest challenges the teams I interviewed faced in the long term. A huge factor in that is managing the automation effectively.

In this section, I present some good ideas that the teams used to reduce the long-term maintenance cost of their automation layers. The advice in this section applies regardless of the tool you choose for automation.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Don't treat automation code as second-grade code

One of the most common mistakes that teams made was treating specifications or related automation code as less important than production code. Examples of this are giving the automation tasks to less-capable developers and testers and not maintaining the automation layer with the same kind of effort applied to production code.

In many cases, this came from the misperception that Specification by Example is just about functional test automation (hence the aliases *agile acceptance testing* and *Acceptance Test-Driven Development*), with developers thinking that test code isn't that important.

Wes Williams said that this reminded him of his early experiences with unit-testing tools:

> I guess it's a similar learning curve to writing JUnit. We started doing the same thing with JUnit tests and then everyone started writing, "Hey guys, JUnit is code; it should be clean." You ran into maintainability problems if you didn't do that. The next thing we learned was that the test pages [executable specifications] themselves are "code."

Phil Cowans listed this as one of the biggest mistakes his team made early on when implementing Specification by Example at Songkick. He added:

> Your test suite is a first-class part of the code that needs to be maintained as much as the regular code of the application. I now think of [acceptance] tests as first class and the [production] code itself as less than first class. The tests are a canonical description of what the application does.
>
> Ultimately the success is more about building the right thing than building it well. If the tests are your description of what the code does, they are not just a very important part of your development process but a very important part of building the product and understanding what you built and keeping the complexity under control. It probably took us a year to realize this.

Clare McLennan says that it's crucial to get the most capable people on the task of designing and building the automation layer:

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

> When I went back the other day, one of the other developers said that the design of the test integration framework is almost more important than the design of the actual product. In other words, the testing framework needs to have as good a design as the actual product because it needs to be maintainable. Part of the reason why the test system succeeded was that I knew about the structure and I could read the code.
>
> What typically happens on projects is they put a junior programmer to write the tests and the test system. However, automated test systems are difficult to get right. Junior programmers tend to choose the wrong approximations and build something less reliable. Put your best architects on it. They have the power to say: If we change this in our design, it will make it much better and easier to get tested.

I wouldn't go as far as saying that the automation code is more important than production code. At the end of the day, the software is built because that production code will help reach some business goal. The best automation framework in the world can't make the project succeed without good production code.

Specifications with examples—those that end up in the living documentation—are much longer lived than the production code. A good living documentation system is crucial when completely rewriting production code in a better technology. It will outlive any code.

Describe validation processes in the automation layer

Most tools for automating executable specifications work with specifications in plain text or HTML formats. This allows us to change the specifications without recompiling or redeploying any programming language code. The automation layer, on the other hand, is programming language code that needs to be recompiled and redeployed if we change it.

Many teams have tried to make the automation layer generic in order to avoid having to change it frequently. They created only low-level reusable components in the automation layer, such as UI automation commands, and then scripted the validation processes, such as website workflows, with these commands. A telling sign for this issue is specifications that contain user interface concepts (such as clicking links or opening windows) or, even worse, low-level automation commands such as Selenium operations.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

For example, the Global Talent Management team at Ultimate Software decided at some point to push all workflow out of the automation layer and into test specifications. They were using a custom-built, open source UI automation tool called SWAT, so they exposed SWAT commands directly as fixtures. They grouped SWAT commands together into meaningful domain workflows for specifications. This approach made writing specifications easier at first but caused many maintenance issues later, according to Scott Berger and Maykel Suarez:

> There is a central team that maintains SWAT and writes macros. At some point it was impossible to maintain. We were using macros based on macros. This made it hard to refactor [tests] and it was a nightmare. A given [test context] would be a collapsible region, but if you expanded it, it would be huge. We moved to implementing the workflow in fixtures. For every page [specification], we have a fixture behind.

Instead of describing validation processes in specifications, we should capture them in the automation layer. The resulting specifications will be more focused and easier to understand.

Describing validation processes (how we test something as opposed to what's being tested) in the automation layer makes that layer more complex and harder to maintain, but programming tools such as IDEs make that task easier. When Berger's team described workflows as reusable components in plain-text specifications, they were essentially programming in plain text without the support of any development tools.

We can use programming tools to maintain the implementation of validation processes more efficiently than if they were described in plain text. We can also reuse the automated validation process for other related specifications more easily. See the sidebar "Three levels of user interface automation" later in this chapter for more information on this topic.

Don't replicate business logic in the test automation layer

Emulating parts of the application business flow or logic in the automation layer can make the tests easier to automate, but it will make the automation layer more complex and harder to maintain. Even worse, it makes the test results unreliable.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The real production flow might have a problem that wasn't replicated in the automation layer. An example that depends on that flow would fail when executed against a real system, but the automated tests would pass, giving the team false assurance that everything is okay.

This is one of the most important early lessons for Tim Andersen at Iowa Student Loan:

> Instead of creating a fake loan from test-helper code, we modified our test code to leverage our application to set up a loan in a valid state. We were able to delete nearly a third of our test code [automation layer] once we had our test abstraction layer using personas to leverage our application. The lesson here is don't fake state; fantasy state is prone to bugs and has a higher maintenance cost. Use the real system to create your state. We had a bunch of tests break. We looked at them and discovered that with this new approach, our existing tests exposed bugs.

On legacy systems, using production code in automation can sometimes lead to very bad hacks. For example, one of my clients extended a third-party product that mixed business logic with user interface code, but we couldn't do anything about that. My clients had read-only access to the source code for third-party components. Someone originally copied and pasted parts of the third-party functionality into test fixtures, removing all user interface bindings. This caused issues when the third-party supplier updated their classes.

I rewrote those fixtures to initialize third-party window classes and access private variables using reflection to run through the real business workflow. I'd never do anything like that while developing production code, but this was the lesser of the two evils. We deleted 90% of the fixture code and occasionally had to fix the automation when the third-party provider changed the way private variables are used, but this was a lot less work than copying and modifying huge chunks of code all the time. It also made tests reliable.

Automate along system boundaries

When: Complex integrations

If you work on a complex heterogeneous system, it's important to understand where the boundaries of your responsibility lie. Specify and automate tests along those boundaries.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

With complex heterogeneous systems, it might be hard or even impossible to include the entire end-to-end flow in an automated test. When I interviewed Rob Park, his team was working on an integration with an external system that converts voice to data. Going through the entire flow for every automated case would be impractical, if not impossible. But they weren't developing voice recognition, just integrating with such a system.

Their responsibilities are in the context of what happens to voice messages after they get converted to data. Park says that they decided to isolate the system and provide an alternative input path to make it easier to automate:

> Now we're writing a feature for Interactive Voice Response. Policy numbers and identification get automatically transferred to the application from an IVR system, so the screens come up prepopulated. After the first Three Amigos conversation, it became obvious to have a test page that prepares the data sent by the IVR.

Instead of automating such examples end to end including the external systems, Park's team decoupled the external inputs from their system and automated the validation for the part of the system that they're responsible for. This enabled them to validate all the important business rules using executable specifications.

Business users naturally will think about acceptance end to end. Automated tests that don't include the external systems won't give them the confidence that the feature is working fully. That should be handled by separate technical integration tests. In this case, playing a simple prerecorded message and checking that it goes through fully would do the trick. That test would verify that all the components talk to each other correctly. Because all the business rules are specified and tested separately, we don't need to run high-level integration tests for all important use cases.

For more tips on how to deal with large complex infrastructures, see the next chapter.

Don't check business logic through the user interface

Traditional test automation tools mostly work by manipulating user interface objects. Most automation tools for executable specifications can go below the user interface and talk to application programming interfaces directly.

Unless the only way to get confidence out of automated specifications for a feature is to run them end to end through the user interface, don't do it.

User interface automation is typically much slower and much more expensive to maintain than automation at the service or API level. With the exception of using

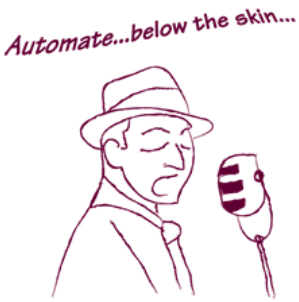
Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

visible user interface automation to gain trust (as described earlier in this chapter), going below the user interface is often a much better solution to verifying business logic whenever possible.

Automate below the skin of the application

When: Checking session and workflow constraints

Workflow and session rules can often be checked only against the user interface layer. But that doesn't mean that the only option to automate those checks is to launch a browser. Instead of automating the specifications through a browser, several teams developing web applications saved a lot of time and effort going right below the skin of the application—to the HTTP layer. Tim Andersen explains this approach:



> We'd send a hash-map that looks a lot like the HTTP request. We have default values that would be rewritten with what's important for the test, and we were testing by basically going right where our HTTP requests were going. That's how our personas [fixtures] worked, by making HTTP requests with an object. That's how they used real state and used real objects.

Not running a browser allows automated checks to execute in parallel and run much faster. Christian Hassa used a similar approach but went one level lower, to the web controllers inside the application. This avoided the HTTP calls as well and made the feedback even faster. He explains this approach:

> We bound parts [of a specification] directly to the UI with Selenium but other parts directly to a MVC controller. It was a significant overhead to bind directly to the UI, and I don't think that this is the primary value of this technique. If I could choose binding all specifications to the controller or a limited set of specifications to the UI, I would always choose executing all the specifications to the controller. Binding to the UI is optional to me; not binding all specifications that are relevant to the system is not an option. And binding to the UI costs significantly more.

Automating just below the skin of the application is a good way to reuse real business flows and avoid duplication in the automation layer. Executing the checks directly using HTTP calls—not through a browser—speeds up validation significantly and makes it possible to run checks in parallel.

Browser automation libraries are often slow and lock user profiles, so only one such check can run at any given time on a single machine. There are many tools and libraries for

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

direct HTTP automation, such as *WebRat*,[2] *Twill*,[3] and the *Selenium 2.0 HtmlUnit driver*.[4] Many modern MVC frameworks allow automation below the HTTP layer, making such checks even more efficient. These tools allow us to execute tests in parallel, faster, and more reliably because they have fewer moving parts than browser automation.

Choosing what to automate

In *Bridging the Communication Gap*, I advised automating all the specifications. After talking to many different teams while preparing this book, I now know that there are situations where automation would not pay off. Gaspar Nagy gave me two good examples:

> "If the automation cost would be too high compared to the benefit of that acceptance criteria—for example, displaying in a sortable grid. The user interface control [widget] will support sorting out of the box. To check whether the data is really sorted you need lots of test data edge cases. This is best left to a quick manual check.
>
> Our application required offline functionality as well. Very special offline edge cases might be hard to automate, and testing manually is probably good enough."

In both these cases, a quick manual check can give the team a level of confidence in the system that was acceptable to their customers. Automation would cost much more than the time it would save long term.

Checking layout examples is, in most cases, a bad choice to automate. Automating them is technically possible, but for many teams the benefits of that wouldn't justify the costs. Automating reference usability examples (such as the ones suggested in the "Build a reference example" section in chapter 7) is practically impossible. Usability and fun require a human eye and a subjective measurement. Other good examples of checks that are probably not worth automating are intuitiveness or asserting how good something looks or how easy it is to use. This doesn't mean that such examples aren't useful to discuss, illustrate with examples, or store in a specification system; quite the contrary. Discussing examples will ensure that everyone has the same understanding, but we can check the result more efficiently by hand.

Automating as much as we can around those functions can help us focus manual checks only on the very few aspects where initial automation or long-term maintenance would be costly.

[2] http://wiki.github.com/brynary/webrat
[3] http://twill.idyll.org
[4] http://seleniumhq.org/docs/09_webdriver.html#htmlunit-driver

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Although I've mostly presented web applications as examples when talking about user interfaces, the same advice is applicable to other types of user interfaces. Automating just below the skin of the application allows us to validate workflow and session constraints but still shorten the feedback time compared to running tests through the user interface. After looking into managing automation in general, it's time to cover two specific areas that caused automation problems for many teams: user interfaces and data management.

Automating user interfaces

When it comes to automation, dealing with user interfaces was the most challenging aspect of Specification by Example for the teams covered by my research. Almost all the teams I interviewed made the same mistake early on. They specified tests intended to be automated through user interfaces as series of technical steps, often directly writing user interface automation commands in their specifications.

User interface automation libraries work in the language of screen objects, essentially software design. Describing specifications in that language directly contradicts the key ideas of refining the specification (see the "Scripts are not specifications" and "Specifications should be about business functionality, not software design" sections in chapter 8). In addition to making specifications hard to understand, this makes automated tests incredibly hard to maintain long term. Pierre Veragen worked on a team that had to throw away all the tests after a small change to the user interface:

> "User interface tests were task oriented (click, point) and therefore tightly coupled to the implementation of the GUI, rather than activity oriented. There was a lot of duplication in tests. FitNesse tests were organized according to the way UI was set up. When the UI was updated, all these tests had to be updated. The translation from conceptual to technical changed. A small change to the GUI, adding a ribbon control, broke everything. There was no way we could update the tests."

The investment they put into tests up to that point was wasted, because it was easier for them to throw away all those tests than to update them. The team decided to invest in restructuring the architecture of the application to enable easier testing.

If you decide to automate validation for some of your specifications through a user interface, managing that automation layer efficiently is probably going to be one of the key activities for your team. Here are some good ideas on how to automate tests through a user interface and still keep them easy to maintain.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Specify user interface functionality at a higher level of abstraction

Pushing the translation from the business language to the language of user interface objects into the automation layer helps to avoid long-term maintenance problems. This essentially means specifying user interface tests at a higher level of abstraction. Aslak Hellesøy says that this was one of the key lessons he learned early on:

> We realized that if we could write tests on a higher level, we could achieve a lot of benefits. This allowed us to change the implementation without having to change a lot of feature scripts. The tests were a lot easier to read, because they were shorter. We had hundreds of these tests, and just by glancing over them it was much easier to see where the things were. They were much more resilient to change.

Lance Walton had a similar experience, which resulted in creating classes in the integration layer that represented operations of user interface screens and then raising the level of abstraction to workflows and finally to higher-level activities. He explains:

> We went through the predictable path of writing tests in "type this, click this button" style with lots of repetition between tests. We had a natural instinct to refactor and realized we needed a representation of the screens. I very much go with the early XP rules: If you have a small expression that has a meaning, refactor it to a method and give it a name. It was predictable that we'll have to log in for every single test, and that should be reusable. I didn't quite know how to do it, but I knew that was going to happen. So we came up with screen classes.
>
> The next thing to realize was that we kept going through the same sequence of pages—it was a workflow. The next stage was to understand that the workflow still had to do with the solution we designed, so actually let's forget about workflow and focus on what the user is trying to achieve.
>
> So we had pages that contained the details, then we had the task level above that, then we had the whole workflow on top of that, and then we finally had the goal that the user is trying to achieve. When we got to that level, the tests could be composed very quickly, and they were robust against the changes.

Reorganizing the automation layer to handle activities—and focusing tests on specifications, not scripts—helped reduce the maintenance costs of automated tests significantly, Walton said:

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

> Early on you had to log in to see anything. At one point there was a notion that you could see a whole bunch of stuff before logging in, and you would only be asked to log in when you followed a link. If you have a whole lot of tests that log in at the start, the first problem you have is that, until you remove the login step, all your tests break. But you have to log in after you follow a link, so a whole bunch of tests would break because of that. If you have abstracted that away, the fact that your test is logging in as a particular person doesn't mean that it's doing that immediately—you just store that information and use it when asked to log in.
>
> The tests move smoothly through. Of course, you need additional tests to check when you are required to log in, but this is a different concern. All the tests that are about testing whether the users can achieve their goal are robust even with that fairly significant change. It was surprising and impressive to me that we could make this change so easily. I truly began to see the power we have to control this stuff.

The fact that a user had to be logged in for a particular action was separated from the actual activity of filling in the login form, submitting it, and logging in. The automation layer decided when to perform that action in the workflow (and if it needed to be performed at all). This made the tests based on the specifications much more resilient to change. It also raised the level of abstraction for user interface actions, allowing the readers to understand the entire specification easier.

Specifying user interface functionality from a higher level of abstraction allows teams to avoid the translation between business and user interface concepts. It also makes the acceptance tests easier to understand and more resilient to change, reducing the long-term maintenance costs.

See the sidebar "Three levels of user interface automation" later in this chapter for an idea how to organize UI test automation to keep all the benefits of refining the specification and reduce long-term maintenance costs.

Check only UI functionality with UI specifications

When: User interface contains complex logic

If your executable specifications are described as interactions with user interface elements, specify only user interface functionality.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The only example where tests described at a lower technical level didn't cause huge maintenance problems later on was the one I saw from the Sierra team at BNP Paribasin London. They had a set of executable specifications described as interactions with user interface elements. The difference between this case and all the other stories, where such tests caused headaches, was that the Sierra team specifies only user interface functionality, not the underlying domain business logic. For example, their tests check for mandatory form fields and functionality implemented in JavaScript. All their business logic specifications are automated below the user interface.

Raising the level of abstraction would certainly make such tests easier to read and maintain. On the other hand, that would complicate the automation layer significantly. Because they have relatively few of these tests, creating and maintaining a smart automation layer would probably take more time than just changing the scripts when the user interface changes. It's also important to understand that they maintain a back-office user interface where the layout doesn't change as much as in public-facing websites, where the user interface is a shopping window.

Avoid recorded UI tests

Many traditional test automation tools offer record-and-replay user interface automation. Although this sounds compelling for initial automation, record-and-replay is a terrible choice for Specification by Example. This is one of the areas where automation of executable specifications is quite different than traditional automated regression testing.

Avoid recording user interface automation if you can. Apart from being almost impossible to understand, recorded scripts are difficult to maintain. They reduce the cost of creating a script but significantly increase the cost of maintenance.

Pierre Veragen's team had 70,000 lines of recorded scripts for user interface regression tests. It took several people six months to re-record them to keep up with significant user interface changes. Such slow feedback would completely invalidate any benefits of executable specifications. In addition to that, record-and-replay automation requires a user interface to exist, but Specification by Example starts before we develop a piece of software.

Some teams didn't understand this difference between traditional regression testing and Specification by Example at first and tried to use record-and-replay tools. Christian Hassa's story is a typical one to consider:

> The tests were still too brittle and had a significant overhead to maintain them. Selenium tests were recorded, so they were also coming in too late. First we tried to record what was there at the end of the sprint. Then

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

we tried to abstract the recording to make it more reusable and less brittle. At the end, it was still the tester who had to come up with his own ideas on how to test. We found very late how the tester interpreted the user expectations. Second, we were still late in becoming ready to test. Actually it made things worse because we had to maintain all this. Six months later the scripts we used were no longer maintainable.

We used the approach for a few months and tried to improve the practice, but it didn't really work, so we dropped it by the end of the project. The tests we wrote were not structured the way we do it now, but rather the way a classical tester would structure tests—a lot of preconditions, then some asserts, and the things to do were preconditions for the next test.

Three levels of user interface automation

To write executable specifications that are automated through a user interface, think about describing the specification and the automation at these three levels:

- Business rule level—What is this test demonstrating or exercising? For example: Free delivery is offered to customers who order two or more books.
- User workflow level—How can a user exercise the functionality through the UI, on a higher activity level? For example: Put two books in a shopping cart, enter address details, and verify that delivery options include free delivery.
- Technical activity level—What are the technical steps required to exercise individual workflow steps? For example: Open the shop home page, log in with "testuser" and "testpassword," go to the "/book" page, click the first image with the "book" CSS class, wait for the page to load, click the Buy Now link, and so on.

Specifications should be described at the business rule level. The automation layer should handle the workflow level by combining blocks composed at the technical activity level. Such tests will be easy to understand, efficient to write, and relatively inexpensive to maintain.

For more information on three levels of UI tests, see my article "How to implement UI testing without shooting yourself in the foot."[†]

[†] http://gojko.net/2010/04/13/how-to-implement-ui-testing-without-shooting-yourself-in-the-foot-2

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Set up context in a database

Even when the only way to automate executable specifications is through a user interface, many teams found that they can speed up test execution significantly by preparing the context directly in their database.

For example, when automating a specification that describes how editors can approve articles, we could pre-create articles using database calls. If you use the three layers (described in the previous sidebar), some parts of the workflow layer can be implemented through the user interface and some can be optimized to use domain APIs or database calls. The Global Talent Management Team at Ultimate Software uses this approach but splits the work so that testers can still participate efficiently. Scott Berger explains it:

> The developer would ideally write and automate the happy path with the layer of this database automation that sets up the data. A tester would then pick that up and extend with additional cases.

By automating the whole path early, developers use their knowledge of how to optimize tests. Once the first example is automated, testers and analysts can easily extend the specification by adding more examples at the business rule level.

Setting up the context in a database leads us to the second biggest challenge the teams from my research face when automating executable specifications: data management. Some teams included databases in their continuous validation processes to get more confidence from their systems or because their domains are data driven. This creates a new set of challenges for automation.

Test data management

To make executable specifications focused and self-explanatory, specifications need to contain all the data that's important to illustrate the functionality with examples but omit any additional information. But to fully automate the examples against a system that uses a database, we often need additional data because of referential integrity checks.

Another problem with automated tests relying on data stored in a database is that one test can change the data required by another test, making the test results unreliable. On the other hand, to get fast feedback, we can't drop and restore the entire database for every test.

Managing test data efficiently is crucial to gain confidence from data-driven systems and make the continuous validation process fast, repeatable, and reliable. In this section, I present some good practices that the teams I interviewed used to manage the test data for their executable specifications.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Avoid using prepopulated data

When: Specifying logic that's not data driven

Reusing existing data can make specifications harder to understand.

When executable specifications are automated to use a database, the data in the database becomes part of the automation context. Instead of automating how the contextual information is put into the database before a test, some teams reused existing data that suits the purpose. This makes it easier to automate the specifications but makes them harder to understand. Anyone who reads such specifications has to also understand the data in the database. Channing Walton advises against this:

> Setting up databases by prepopulating a standard baseline data set almost always causes a lot of pain. It becomes hard to understand what the data is, why it is there, and what it is being used for. When tests fail, it's hard to know why. As the data is shared, tests influence each other. People get confused very quickly. This is a premature optimization. Write tests to be data agnostic.

If the system is designed in a way not to require a lot of referential data setup, then specifications can be automated by defining only a minimal set of contextual information. Looking at this from the other side of the equation, Specification by Example guides teams to design focused components with low coupling, which is one of the most important object-oriented design principles. But this isn't easy to do with legacy data-driven systems.

Try using prepopulated reference data

When: Data-driven systems

Defining the full context for data-driven systems is difficult and error-prone. It might not be the best thing to do from the perspective of writing focused specifications. Gaspar Nagy's team tried to do that and found that specifications became hard to read and maintain:

> We had an acceptance test where we had to set up some data in the database to execute a step. When we did this setup description, it was looking like a database. We didn't say "table" in the text, but they were tables. Developers were able to understand it very well, but you couldn't show this to a businessperson.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

> For example, we had a table for the countries. We didn't want to hard-code any logic in test automation on what were the countries, so for each of the tests we defined the countries that were relevant for this test. This turned out to be completely stupid because we always used Hungary and France. We could have just loaded all the countries of the world into the database with a "given the default countries are in the system." Having a default data set would be helpful.

Marco Milone had a similar problem while working on a project in the new media industry:

> At the beginning, for the sake of getting the tests to run, we weren't doing things well. Setup and teardown were in the test, and they were so cluttered. We started centralizing the database setup and enforced change control on top of that. Tests just did checks; we didn't bother with entering data in the tests. This made the tests much faster and much easier to read and manage.

On data-driven systems, creating everything from scratch isn't a good idea. On the other hand, hiding information can cause a ton of problems as well. A possible solution for this is a strategy implemented by the teams at Iowa Student Loan. They prepopulate only referential data that doesn't change. Tim Andersen explains this approach:

> We "nuke and pave" the database during the build. We then populate it with configuration and domain test data. Each test is responsible for creating and cleaning up the transaction data.

Using prepopulated reference data is a good strategy to make test specifications shorter and easier to understand, while at the same time speeding up feedback and simplifying the automation layer.

If you decide to use prepopulated reference data, see the "Run quick checks for reference data" section in chapter 10 for information on how to make tests more reliable.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Pull prototypes from the database

When: Legacy data-driven systems

Some domains are so complex that even with prepopulated reference data, setting up a new object from scratch would be a complex and error-prone task. If you face this on a greenfield project, where the domain model is under your control, this might be a sign that the domain model is wrong (see the "Listen to your living documentation" section in chapter 11).

On legacy data-driven systems, changing the model might not be an option. In such cases, instead of creating a completely new object from scratch, the automation layer can clone an existing object and change the relevant properties. Børge Lotre and Mikael Vik used this approach for the Norwegian Dairy Herd Recording System. They said:

> Getting the correct background for the test so that it is as complete as possible was a challenge because of the complexity of the domain. If we were testing a behavior of a cow and we had forgotten to define a test case where she had three calves, we didn't see the code failing and didn't spot the error before we tested it manually on real data. So we created a background generator where you could identify a real cow and it pulls its properties from the database. These properties were then used as the basis for a new Cucumber test. This not only was useful when we wanted to re-create an error but also turned out be a real help when we start on new requirements.

When the Bekk team identifies a missing test case, they find a good representative example in the real database and use the "background generator" to set up an automated acceptance test using its properties. This ensures that complex objects have all the relevant details and references to related objects, which makes validation checks more relevant. To get faster feedback from their executable specifications, the background generator pulls the full context of an object, which enables the tests to run against an in-memory database.

Find a representative example in the database, and use those properties to set up tests.

When this approach is used to create objects on the fly instead of creating the context for a test (in combination with a real database), it can also simplify the setup required for relevant entities in executable specifications. Instead of specifying all the properties

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

for an object, we can specify only those that are important to locate a good prototype. This makes the specifications easier to understand.

Automating the validation of specifications without changing them is conceptually different from traditional test automation, which is why so many teams struggle with it when they get started with Specification by Example. We automate specifications to get fast feedback, but our primary goal should be to create executable specifications that are easily accessible and human readable, not just to automate a validation process. Once our specifications are executable, we can validate them frequently to build a living documentation system. We'll cover those ideas in the next two chapters.

Remember

- Refined specifications should be automated with as little change as possible.
- The automation layer should define how something is tested; specifications should define what is to be tested.
- Use the automation layer to translate between the business language and user interface concepts, APIs, and databases. Create higher-level reusable components for specifications.
- Automate below the user interface if possible.
- Don't rely too much on existing data if you don't have to.

Chapter 9. Automating validation without changing specifications. Specification by Example: How successful teams deliver the right software, ISBN
Prepared for yeongsheng.tan@baml.com, YEONG TAN
©2011 by Manning Publications Co. All rights reserved. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.