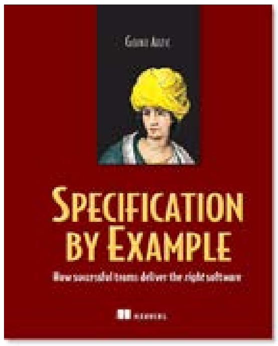
MANNING PUBLICATIONS



Specification by Example
By Gojko Adzic

Controlling the cost of maintenance for a living documentation system is one of the biggest challenges a team may face. In this article from chapter 9 of Specification by Example, author Gojko Adzic presents some good ideas that the teams he interviewed used to reduce the long-term maintenance cost of their automation layers and covers two specific areas that caused automation problems for many teams: user interfaces and data management.

You may also be interested in…

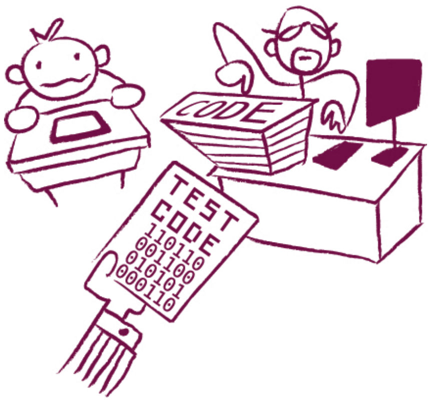
Managing the Automation Layer

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 article, 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.

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:

> 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.

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" further in this article 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.

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.

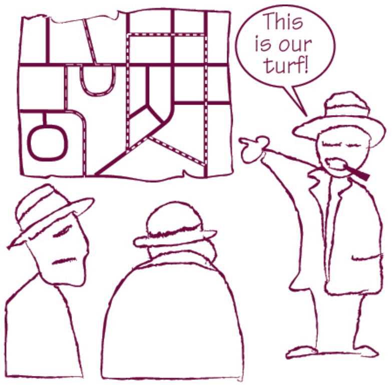
I rewrote those fixtures to initialize third-party window classes and access private variables using reflection to run through the real business workflow. I'd never do anything like that while developing production code, but this was the lesser of the two evils. We deleted 90% of the fixture code and 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.

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.

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.

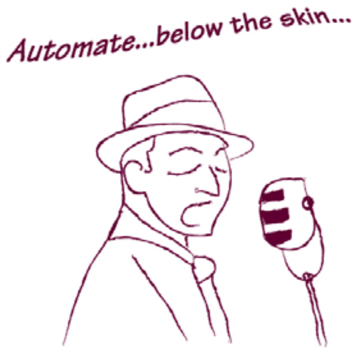
User interface automation is typically much slower and much more expensive to maintain than automation at the service or API level. With the exception of using visible user interface automation to gain trust (as described earlier in this article), going below the user interface is often a much better 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 direct HTTP automation, such as *WebRat*,[1] *Twill*,[2] and the *Selenium 2.0 HtmlUnit driver*.[3] 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.

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

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 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.

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. 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.

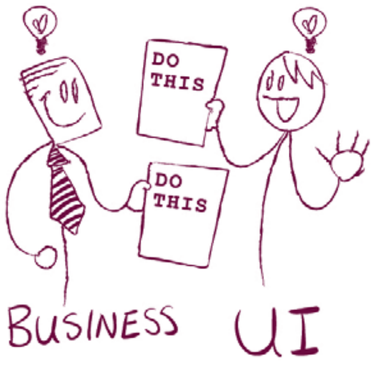
The investment they put into tests up to that point was wasted, because it was easier for them to throw away all those tests than to update them. The team decided to invest in restructuring the architecture of the application to enable easier testing.

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.

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:

> 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" further in this article 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 inter face elements, specify only user interface functionality.

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 rerecord 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 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."[4]

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.

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

For example, when automating a specification that describes how editors can approve articles, we could precreate 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.

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.
>
> 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.

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.

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 recreate 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.

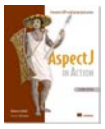
When this approach is used to create objects on the fly instead of creating the context for a test (in combination with a real database), it can also simplify the setup required for relevant entities in executable specifications. Instead of specifying all the properties for an object, we can specify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

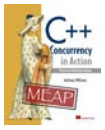
For source code, sample chapters, the Online Author Forum, and other resources, go to
http://www.manning.com/adzic/

Here are some other Manning titles you might be interested in:

Becoming Agile
...in an imperfect world
Greg Smith and Ahmed Sidky



AspectJ in Action, Second Edition
Enterprise AOP with Spring Applications
Ramnivas Laddad



C++ Concurrency in Action
Practical Multithreading
Anthony Williams

Last updated: August 23, 2011