Gojko Adzic

Specification by Example

How successful teams deliver the *right* software

MANNING

Table of Contents

14

Iowa Student Loan

Iowa Student Loan is a financial services company that pushes the ideas of Specification by Example to the limit. They're an interesting case study because their living documentation system gave the business a competitive advantage. It enabled them to efficiently deal with a major business model change.

The Iowa Student Loan development team builds and maintains a complex system, from the public website, which takes loan requests, to the back-office systems for underwriting and origination. Apart from that, the main driver of complexity on their projects is the data-driven nature of the domain.

I interviewed Tim Andersen, Suzanne Kidwell, Cindy Bartz, and Justin Davis, who worked on several different projects while the company was improving its software process. It was interesting to track how they rolled practices from a smaller project into rewriting the entire underwriting platform.

Changing the process

In 2004, the Iowa Student Loan development team implemented Extreme Programming by the book to improve the quality of their software. When their next project went to production, they prepared to handle bugs similarly to the way they did things in the past. Over the next 12 months, the new system only had half a dozen bugs. This proved to the management that agile development, especially writing tests first, was a good idea and that it significantly improves quality.

The tests were, however, very technical. The team used HTTPUnit (a unit-testing framework for websites). Programmers translated use cases to HTTPUnit tests, which weren't readable by anyone. When the system went live, they noticed that they were missing documentation. They hired a consultant, J. B. Rainsberger, to help them figure out what they were doing wrong and to give them ideas on improving with tools and practices. One of the tools he introduced was FitNesse.

The team was wrapping up the first project in which they used FitNesse as a way to capture the specifications in July–August 2006. This project allowed the team to learn how to use the tool and also led them to rethink how they write executable specifications. The business analysts had technical knowledge, so the specifications they wrote with developers turned out very technical. As a result, the business users couldn't understand them. Justin Davis explains that problem:

> I could look at tests and read them as a business analyst, and we were still writing them, but they were very disconnected from the other business members of the team.

This was just the start of a larger effort to rewrite the entire underwriting platform and automate a lot of the work that was previously done manually on paper. The next project would take three years with a team of six developers, two testers, a business analyst, and an on-site business user. They brought in a consultant to help them communicate better with the business users. Tim Andersen says:

> David Hussman said that we should work harder on developing tests that make sense, so that when businesspeople read them, we don't have to explain the tests to them. That was pretty hard to do. It took a shift in our thought process, and we had to be a lot more business savvy. It required a lot more understanding and conversation about how the system should work instead of just technical requirements.

They started to describe the system with user personas, which allowed them to consider how different groups of users were interacting with the system.

Instead of using generic users, they started thinking a lot more about why different groups of people use the system, what they want to get out of it, and how they use it. This allowed the business stakeholders to engage better and provide more meaningful information to the team. You can see some nice examples of the personas they used in Tim Andersen's presentation from the Code Freeze 2010 conference.[1]

Optimizing the process

Because their executable specifications were previously very technical, the automation layer was complicated and hard to maintain. Tests described technical components as parts of larger flows, so they had to be automated by faking portions of the user workflows. Tim Andersen says that the test results also weren't reliable:

[1] http://timandersen.net/presentations/Persona_Driven_Development.pdf

> We were able to show the test working, and then we weren't able to show working software. Our tests were lying (false green bar). For example, a borrower can borrow money if he is less than 18 years old. We'd have a test that if they are less than 18 on that day, it would say, "You're not allowed to borrow without a co-signer." If you change the date of birth to be over 18 years, it would say "OK, you can borrow without a co-signer." Our test was green, but when we actually opened a browser and tried it in development, it didn't work. Even though we coded the validation rule, it wasn't hooked up in the right place. Our test code was setting up a loan within a fantasy state.

The business users didn't trust the test results from executable specifications, so they didn't consider them important, which was another barrier to get them more engaged in the process. Andersen says:

> There was a lot of frustration on both sides. We asked, "How come they aren't reviewing the tests; how come they aren't valuing the tests?" At the same time the business team was frustrated: "How come the developers have a passing test but it doesn't work?" They did not believe in those tests.

The team restructured the automation layer for executable specifications to go through production flows, not trying to fake state. The new way of automating specifications fit in nicely with the ways they were describing the system with personas. Andersen says:

> "Fantasy state" is the term I kept using to let other developers know that I didn't trust a test that wasn't using the correct entry point. Other symptoms of fantasy state are "thick fixtures"; fixtures shouldn't have much logic in them and should be pretty lightweight. Using personas helped us find the right level of abstraction to identify the appropriate entry point in our application. Before we used personas, we often picked an inappropriate entry point, which led us to heavy fixtures that were prone to fantasy state.

The team organized the automation along the activities that would be available to a persona. Each persona was implemented as a fixture in the automation layer, talking to the server using HTTP calls, essentially going in the same way a browser would but without launching a browser. This enabled them to significantly simplify the automation layer,

but it also made test results much more reliable. Some tests started failing after that, and the team discovered bugs that had previously passed unnoticed. Around May 2007, the test results became a lot more reliable and the automation layer was easier to maintain. Andersen adds:

> Changing our test code to leverage the application to set up the state of a loan exposed these bugs so we could fix them, and our false green bar symptoms vanished. It also had an impact by dramatically reducing the cost of test maintenance.

Once the executable specifications were talking about the business functionality on a level that the business users could understand, the automation layer became a lot simpler—it was connecting to the business domain code. It also made the specifications a lot more relevant, because they no longer had misleading false positives from tests that looked at only a part of the flow.

The feedback started to slow down as the number of tests grew. Many slow technical tests were executed through a browser. Andersen said that looking at the system from the perspective of personas helped reduce those problems:

> We used FitNesse as a tool to make WatiJ [a UI automation library] configurable. Before using personas, we kind of fell back to the browser tests as a last resort because "we have to test this somehow to make sure that it really works." Those browser tests multiplied like rabbits.

The team rewrote the browser tests to use personas, which significantly improved the feedback time. Instead of launching a browser every time, the new automation layer issued HTTP requests directly. They also looked into running tests with in-memory databases instead of SQL Server, but they decided that they should improve the performance of the real SQL database using indexes instead. The team broke down the continuous validation process into several modules to get better visibility on what was slowing down the tests.

Instead of always creating new specifications, the team started thinking about integrating change requests with existing specifications. This reduced the number of tests and helped avoid unnecessary setup tasks. Andersen explains:

> We started thinking about scenarios. A new feature might not be a feature by itself; it might be a change to a set of scenarios. Instead of writing a new test for each requirement, we were thinking about that in the context of our current system and what tests we need to change versus what new tests we need to write. That helped keep our build time constant.

This led them to start reorganizing the specifications to reduce the number of tests. They would look for smaller partial specifications and consolidate them into bigger ones. They would break apart large specifications into smaller, more focused ones. "You basically have to refactor your tests and your test code as much as you would refactor your old code," says Andersen.

Iowa Student Loan was an early adopter of Specification by Example, so they had to deal with immature tools, which got in the way of collaboration several times. Because the team was using open source tools, they were able to modify the tools to suit their development process.

Once they started putting executable specifications into a version control system, the business analysts could no longer change them on their own without access to development tools. The developers wrote a plug-in for FitNesse that handled version control system integration, allowing them to still run a wiki where business analysts can change specifications.

As the number of tests grew, the team started to have problems with functional regression. Bugs that should have been caught by existing tests slipped through, because the relevant tests were disabled. Some tests were disabled because developers were unsure of how they fit into the new functionality; some were disabled when the team was waiting on a decision from business stakeholders. People then forgot to reenable these tests or follow up on the discussions. Developers at Iowa Student Loan wrote an automated check for disabled tests (see the "Automatically check which tests are turned off" section in chapter 10), which reminded them at the end of every iteration what they had to follow up on.

They used JIRA to manage requirements and FitNesse to manage executable specifications, so rearranging FitNesse pages broke links in JIRA. They extended FitNesse to support keywords and used keywords to link executable specifications and JIRA web pages. On another project, they took a different approach and created a business framework. The business framework is a set of pages in FitNesse designed to be a stable documentation entry point, which then has internal links to tests. This was a start of a good living documentation system. Justin Davis explains:

> "One of the goals of the business framework was to create a front to FitNesse that the business team could use while also allowing developers to understand the order of the things in the current system. In effect, it provides a map to how the system behaves. So if you have a context there in terms of knowing how the system works, you can go to this framework to find what you want. The system flow would be there, and you can choose which steps you'd like to view tests and requirements for."

Introducing the business framework and making sure that the executable specifications actually get validated frequently and stay relevant enabled them to create a useful living documentation system. They had a relevant source of information on what the system does, which anyone could access.

Living documentation as competitive advantage

With such a good living documentation system, they were able to handle very big changes efficiently. Three months before the end of the project, the business model of the company suddenly had to change. They normally fund loans through a bond sale. Because of the credit crisis in 2008, the bond sale failed. The business is technology driven, so this business model change had to be reflected in their software. Andersen says that the living documentation system helped them understand what was required to support this business change:

> Typically, we use bond proceeds to fund private student loans. However, we changed our business model and made all of the funding portion of the system configurable so that we could use lenders to provide funds and continue to provide loans to the students. It was a dramatic overhaul of a core piece of the system. Before this new funding requirement, our system didn't even have the concept of a lender because we were able to assume Iowa Student Loan was the lender.
>
> We were able to use our existing acceptance tests and repurpose them to say, "OK, here's our funding requirement." For all of the tests we had, we discussed the impact and provided funding so they would still work. We had some interesting discussions based on scenarios where there is no more funding available, or funding is available but not for this school or this lender, so we had some edge cases for these requirements, but it was really making the new funding model more flexible and configurable.

Once they understood the impacts of this new business model on the software, they were able to implement the solution efficiently. According to Andersen, such change would be impossible to implement quickly without a living documentation:

> Because we had good acceptance tests, we were able to implement a solution within a month. Any other system that didn't have the tests would halt the development and it would have been a rewrite.

This is when the investment in the living documentation system paid off. It supported them in analysis, implementation, and testing of the impact of a business model change, at the same time enabling them to quickly verify that the rest of the system is unaffected.

Key lessons

They started out by focusing on a tool and quickly realized that it doesn't help them achieve the goal of bringing business users into the process. So they started approaching the specifications from the perspective of a user. This enabled them to communicate with their business users better and reduce the costs of maintenance for tests. When the tool prevented them from collaborating effectively, they modified it. This is another argument for using open source tools.

Implementing Specification by Example at Iowa Student Loan was driven not by the need to improve quality or automate tests but by the need to build a relevant documentation system in order to be more effective and engage business users better. They invested heavily into building a good living documentation system, which paid off well. It helped them implement a business model change, which was very powerful.