Adzic, G. (2011). *Specification by Example: How Successful Teams Deliver the Right Software* (1st ed.). Manning Publications.

Chapter 12. uSwitch Case Study

uSwitch.com is one of the busiest UK websites. The website compares prices and services for a variety of companies and products, including energy suppliers, credit cards, and insurance providers. The complexity of their software system is driven by high scalability as well as complex integrations with a large number of external partners.

uSwitch is an interesting case study because it illustrates how a company working in a Waterfall process with separate development and testing teams on a problematic legacy environment can still transition to a much better way of delivering quality software. uSwitch has completely overhauled their software delivery process over the course of three years.

At uSwitch, I interviewed Stephen Lloyd, Tony To, Damon Morgan, Jon Neale, and Hemal Kuntawala. When I asked them about their software process, their general answer was, "Someone suggests an idea in the morning and then it gets implemented and goes live." Early in my career, I worked for a company with a software process that could be described the same way—and that experienced fireworks on the production systems almost daily. But in the case of uSwitch, the quality of the product and the speed with which they deliver features is enviable.

Although uSwitch didn't set out to implement Specification by Example in particular, their current process contains the most important patterns described in this book, including deriving scope from goals, collaborating on specifications, and automating executable specifications. To improve the software development process, they focused on improving product quality by constantly looking for and addressing obstacles to quality.

When looking for a better way to align development and testing, they automated tests in human-readable form. After that, they discovered that tests can become specifications. Moving to executable specifications got them to collaborate better. In the course of refining the process, they had to make continuous validation more reliable, which led them to improve how they refine the specifications. When looking for better ways to engage business users, they started to derive scope from goals.

Starting to change the process

In 2007, uSwitch was using a Waterfall development process, working on long projects with big designs up front. A new CTO pushed the teams to "go agile" in 2008, and they introduced three week iterations with Scrum. In October 2008, the average time to market for a new feature at uSwitch was six to nine weeks. Although this was a huge improvement over Waterfall, the effort involved in each sprint consisted of roughly 40% of unplanned work. Scrum works best with cross- functional teams, but because of the way their development was organized, they never got to that point.

The QA team was separate from the development team. Because the testers used QTP—which the developers couldn't access—developers developed and testers tested without communicating with each other; as a result, developers found it hard to know when they were finished. Because the criteria for a release was that the QTP tests had to pass, testing was often a bottleneck in the process.

The deployment process at the end of a sprint took an average of three days, mostly because of testing, and they still had numerous quality problems. When the team moved to short iterations, QTP tests started to require a lot of maintenance. According to Hemal Kuntawala, "Nobody knew what they did, and they were a complete waste of time."

This led to a companywide effort to focus on quality. Everyone was asked to start thinking about quality. They recognized the problem with developers throwing things to testers without explanation and decided to merge the testing team and the development team. They removed the different job titles; testers became "developers" with a particular specialty. In the same way that tasks requiring specialist database knowledge could go to a more experienced database developer, testers were taking

on tasks that required specialist testing knowledge. But they were no longer solely responsible for testing. Programmers started to look into better ways to write unit tests and functional tests. The team decided to use Selenium instead of QTP to make tests more lightweight and accessible to everyone. This enabled developers and testers to collaborate better, but because Selenium is quite technical, this change didn't give them a better way to communicate with the business users.

Because uSwitch didn't have any reliable documentation about a system that had been built over a period of 10 years, legacy business rules often caused problems in understanding. Kuntawala says:

"One day we had a legacy business rule in Energy [a subsystem] that I didn't know about. It was frustrating. I wanted a way for us to know about business rules and how the application works without diving into the code and the unit tests. Not everything was unit tested anyway. Googling around, we found Cucumber to bridge the gap between tests and portraying what we wanted to achieve—the goal of the feature. We could write what we wanted in plain English, and it would fit in with the outside-in approach that developers were trying to achieve."

To get everyone familiar with the new tool, they started to convert Selenium tests to Cucumber. This was still test automation—checking for problems after the fact—but it sparked a move to test- first executable specifications. Jon Neale explained:

"The Given-When-Then format of Cucumber tests forced us to rewrite the stories and really nail down what we were building, showing us that we've forgotten stuff."

The team started taking business stakeholders through different Cucumber scenarios, not only to verify edge cases but also to identify which scenarios were important, reducing the scope and avoiding just-in-case code.

By the time they finished converting Selenium tests to Cucumber and reviewing them with the business users, they realized that testing at the end of the iteration didn't make sense. Neale said:

"We realized that we could gain quite a lot by sitting down and having specification workshops and drawing out exactly what we wanted to achieve and how we were going to achieve it."

The team then introduced specification workshops as a way of collaborating with the business users to nail down the acceptance criteria for future requirements. This significantly improved communication within the group. The developers (at this point testers were also called developers) learned about the domain. Business users learned about edge cases and more obscure user routes because developers were asking about them.

This change also affected the division of labor. Previously, work was mostly organized by technical tasks. With such technical chunks of work, they found it hard to work out a specific acceptance criteria for each task. The team moved the focus away from the implementation tasks to the value that a feature should deliver. They started describing stories from the user perspective, which made it easier to discuss and specify the acceptance criteria for chunks of work.

This new organization also allowed them to release software more often. Because technical tasks depended on each other, developers were reluctant to deploy a task until everything in a larger block of work was complete. By focusing on user stories, they worked on independent, smaller chunks that could be released more often.

Optimizing the process

As the number of executable specifications grew, the team noticed that the test results were unreliable. Problems with the environment often caused the tests to fail, even when the functionality in the system was correct. They had no stable environment to run tests against. They had a development environment, a testing environment, and a staging environment, but none of these was appropriate for running executable specifications frequently.

Because the developers used the development environment to try things out, it was often broken. The testing environment was used for manual testing and deployed on demand. Any number of changes could occur between two deployments, so when a test failed it wasn't clear what caused the problem. Business users were also manually testing on this environment, which could affect automated Cucumber test results. The staging environment was a mirror of production and was used for final deployment testing.

uSwitch created one more environment, to be used exclusively for continuous validation. This was a solution to stability problems: a dedicated environment that could be used for testing without interrupting other work. This environment was deployed automatically by their continuous build system. With this environment, the feedback from executable specifications was received quickly and became significantly more reliable.

Once they eliminated environment problems as a source of instability, they could see which tests or parts of software were unstable by design. Because all the tests were executing through the user interface, an increase in the number of executable specifications running the tests caused a bottleneck. Some of the tests were slow, and some were unreliable. They started removing unreliable tests and looking into the causes of instability in order to improve them. The tests were written at a technical level, which caused maintenance issues.

The team started rewriting tests, breaking them apart, and raising the level of abstraction. Kuntawala says that this was quite a big step for them:

"When we first started writing tests, they would rely on browser-specific things, for example, DOM identifiers on the page, which would change. Once we got used to the syntax and the power of Cucumber, we started writing tests in a real business language. Previously you would say stuff like, "User enters 100 in box _id." Now you would say, "The user enters a valid amount." A valid amount would be defined in a separate test. Once you have it written, you don't have to test that explicitly in every other test. A test for valid amounts would also try negative numbers, letters, and so on, but it was abstracting away from having that test in every other test. This was quite a big step."

In order to reduce long-term maintenance costs, the uSwitch team started refining the specifications, evolving a consistent language for specifications, and looking for missing concepts to raise the level of abstraction.

With a relatively good functional coverage in executable specifications and a stable continuous validation environment, the uSwitch team had a lot more confidence in their code. But their test suite was running slowly and didn't give them the quick feedback they expected. They decided that not every test was worth running for an automated regression check. Some tests were good to drive development but were unrelated to the functionality that increased profit.

One example was sending delayed emails. They were automatically running executable specifications while implementing the feature but disabled them once the feature was developed. Such low-risk tests wouldn't run as part of the continuous validation process. This gave them quicker feedback and reduced test maintenance costs. The next time someone picked up a development task related to that part of the system, they would reenable the test and clean it up if needed.

Running tests or validating that the system was ready to go live was no longer the bottleneck; now deployment to production was the slowest part of the process. The developers paired with the operations engineers to understand what was slowing them down. It turned out that pre-deployment test execution was causing delays. Some tests were timing out on the staging environment, which required the operations engineers to rerun the entire test pack. By identifying differences in the environments and rewriting the tests to make them more resilient, the developers reduced the execution time for the entire test pack from two hours to about 15 minutes.

Pairing up also helped to get operations engineers involved in the process. Previously, they could report that a test pack had failed, but their report lacked details. Once they understood how to interpret test results, they could provide the developers with a much more meaningful report if something went wrong.

The next change was getting the business users more involved with the development. Although the team was using user stories for planning at this point, they were writing the user stories themselves. The business users started writing stories with the development team, taking more ownership over specifications. They would generally define the benefit ("so that") and the developers would then define the solution ("I want"). The business users also became responsible for running specification workshops. This improved communication on the team. Damon Morgan explained:

"They were previously divorced from the process. They would ask, "Can we have this?" and we'd write it down in some odd language that they didn't necessarily get. They would see it move across the board, and it didn't really mean anything to them. Once we got into specifications [workshops] and talked to them a lot more about what should be actually delivered, having executable criteria for those stories and working with them to write the stories, they took much more ownership of the whole thing. We wouldn't get stories coming back from the business in terms of "you didn't do this right." It would be more in terms of "as a team, we didn't think about this scenario.""

With more involvement from the business users, the team at uSwitch built trust and confidence. This meant that there was no need for long-term prioritization and big chunks of work. It also meant that the business users would be more open to suggestions from the development team.

With closer collaboration and more trust, the business users were open to approaching the development scope differently. The team then started breaking down required functionality into minimal features that would be releasable and still give the business some value.

One example is the process of rewriting the energy directory, a four-level page hierarchy that contains an index of energy suppliers and plans. Instead of releasing it all at once, they were rewriting it one page at a time, hooking up that page to the rest of the services, and releasing it. Although this approach increased integration costs—because new pages had to be integrated with the old pages—they got a lot of value out of releasing earlier. One of the reasons for rewriting the directory was search engine optimization: Releasing one page at a time meant that Google could index some pages sooner. Also, the team found that smaller releases mitigated the risk of mistakes. If there was a problem, it could be attributed to a particular release. By having smaller releases, it was easier to pinpoint its cause.

Once the team started producing potential deliverables more frequently than the iterations themselves, the sign-off at the end of an iteration became a bottleneck. Instead of one big demonstration at the end, they started showing new features to the business users and getting sign- off as soon as a releasable piece of functionality was done.

The team noticed there was no more need for formal specification workshops, and they were replaced with informal chat sessions. Handling smaller pieces of work and receiving fast feedback allowed the team to proceed when they had enough information to start working, even though they didn't necessarily have enough information to complete the task. According to Damon Morgan:

"At the beginning the [specification workshop] meetings were a lot longer and bigger, and we were trying to spec out a lot more. Now it's really "We're going to start work on this feature now," and it's a relatively small feature so we'll speak with the parties involved. The whole team will get together to kind of do a mini-specification workshop—but it really is just a conversation; you don't even need to go into a room to have it. And you come up with the criteria for that, you start building it, and you show it a lot quicker. It's normally built and delivered in two days, and you move on to the next thing. We're much more iterative in the way we build stuff."

Because the process allowed developers to learn a lot more about the business domain than they used to, they didn't have as many problems caused by misunderstood business requirements, and they could get the right work done with less up-front information. Stephen Lloyd said:

"As a team we are much better integrated and we understand what the business wants a lot more than we used to. So the whole purpose of specifying out exactly what they require is less important because we understand the domain much better now than we did a year ago."

Finally, the team at uSwitch started deploying on demand and moved away from iterations altogether. To help with this process, they started regularly monitoring their production systems, tracking error rates and usage of new features. This additional visibility provided a safety net against unnoticed deployment problems.

The current process

After all those changes, the development process is much simpler. It's lightweight and based on flow, rather than iterations.

New ideas come into the backlog when someone suggests them during a daily stand-up meeting. Anyone can suggest a new idea—including business users or developers. A new idea is briefly discussed at the stand-up meeting and prioritized. The person who suggests it might draw some rough diagrams about it or prepare a business case for it before the meeting, in order to explain the idea better. Apart from that, unless contracts with external partners need to be signed, there isn't a lot of up-front preparation.

When the story becomes one of the top-priority items, the team thinks about what steps would lead to completion. Everyone with an interest in that story will meet to briefly discuss exactly what's needed and write down the acceptance criteria. In the past, the team had tried to produce Cucumber tests during these meetings, but they decided the syntax of Cucumber tests was getting in the way: One person would have to type and the others would be watching, causing an interruption in the

flow of discussion.

The development team and the marketing and email teams sit closely to each other, so they can work without a lot of detail up front. Developers will start working on the story and frequently talk to the business users, asking for more information or revisiting the acceptance criteria.

Acceptance criteria gets converted into Cucumber tests and automated during development. Developers will use exploratory testing to understand the existing parts of the system better before changing them. Sometimes they use customer session logs to understand how real users interact with a particular feature of the website. Based on that, they develop Cucumber tests and capture user journey paths that they need to consider while developing. They usually use browser automation toolkits to automate tests through the user interface. There's no manual scripted testing anymore, but they do a lot of exploratory testing, including trying out different paths through the system and trying to break it.

Once all the Cucumber scenarios pass, the change gets deployed to a release environment and then pushed to the production at some point that same day.

In general, the uSwitch team doesn't track many technical project metrics. Instead, they only look into lead time and throughput. They're much more focused on the business performance of the system and on the value added by a feature. To do so, they monitor user experience metrics such as conversion rates and feature usage rates on the production website.

At the time of my interview, the uSwitch team was moving away from estimations. Estimates are useful when the business users don't trust the development team or when they want to invest in larger pieces of work; now, neither scenario applies to uSwitch. The business users have a greater view of development and trust developers more than before. They also generally work on small increments. Estimating how long a piece of work is going to take isn't necessary.

The result

At uSwitch, the average turnaround time for a feature—from the time it gets accepted for development until it goes live—is currently four days. When I interviewed the team, they couldn't remember a single serious production issue over the previous six months. Boomerangs happen rarely—one every few months. During Hemal Kuntawala's presentation at the Agile Testing UK user group in 2009,[1]

one of the development managers from uSwitch said that “quality has increased substantially and conversion rates have grown.”

1

The entire development process is now driven by expected business values of features. Instead of big plans and large releases, they build small increments, release them often, and monitor whether the increment added value to the business. Because their business model depends on immediate web conversion rates, they can easily achieve this kind of evaluation. You can see some interesting metrics on how this process evolved in slides from Mark Durrand and

Key lessons

To me, one of the most important aspects of this story is that uSwitch decided to focus on improving quality instead of trying to implement any particular process (for more on this, see “Focus on improving quality” in chapter 4). Instead of a big bang approach, they constantly looked for the most important thing to improve and began work there. When they were comfortable with the resulting change, they inspected the process again and moved on to the next issue.

The realization that testing was a bottleneck and that QTP was too expensive and bulky for developers to work with led the team to adopt Specification by Example through functional test automation, an approach I suggested in “Start with functional test automation” in chapter 4. They first adopted Cucumber as a way to automate functional tests but then realized that they could get a lot more out of it because it enabled them to automate tests while keeping them in a human-readable form. This turned the specification process on its head.

Another big lesson from this story is that change, though initially driven by a tool, is mostly cultural. uSwitch removed the division between the testers and the developers and dropped the tester role, making all team members understand that an issue with quality is everyone’s problem. They started focusing on delivering business value instead of implementing technical tasks, which allowed them to increase the involvement of the business users during the development process. Without such close involvement of the business users, it would have been impossible to decide what to build, agree on it, implement it, and verify it within such a short turnaround.

More involvement from the business users meant that they started to understand and trust the development team a lot more, and the developers learned more about the domain. Formal specification workshops were an important step to building this knowledge. Once communication was improved and developers had learned a lot more about the domain, formal workshops became unnecessary. This is an example of how the process can be optimized once team knowledge has built up.

In my mind, the most controversial step taken by uSwitch was the decision to disable less important tests after the functionality was implemented. I’ve seen and heard most of the other ideas with other teams, but they’re the only ones who don’t run all the tests from their living documentation system frequently. Executable specifications for them are truly executable—there’s a potential to execute them but not an obligation. The team found that there’s a lot of value in executing them while they develop a feature but that slower feedback as the result of an ever-growing test suite costs more in the long term than protection against functional regression in less-risky areas. This is perhaps because they have other means to protect against problems in production, in particular the continuous user experience monitoring system.