September 2014

testing experience

The Magazine for Professional Testers

NEW: New Columns & Book Corner

printed in Germany
print version 8,00 €
free digital version
www.testingexperience.com
ISSN 1866-5705

By Rudolf de Schipper & Abdelkrim Boujraf

Test-Driven Developments are Inefficient; Behavior-Driven Developments are a Beacon of Hope?

The StratEx Experience (A Public-Private SaaS and On-Premises Application) – Part I

At StratEx (*www.stratexapp.com*), apart from developing the StratEx application, we obviously worry about quality and testing the app. After all, we are committed to providing quality to our users.

StratEx uses a concept of code generation to shorten the development cycle, hereby being able to quickly implement new features and functionality. It also provides for a nice uniform user interface, because we have kept the code generation as simple as we dared to. So it is not very easy to create a (generated) screen that looks very different from the others.

The code generation concept gives us reliable code (the part that is generated at least), leading also to reduced testing time. Still, next to coding efficiently, we wanted to find ways to efficiently test as well. The obvious (from our perspective at least) solution was to automate and to generate our tests. While this sounds easy, in reality it is not so obvious. We spent quite some time figuring out what the best testing strategy could be and how to implement this (using test automation)

At first we settled for user interface (UI) testing, using Selenium. We hand-recorded some testing scenarios and included them into our continuous integration build process (I will talk about this in another post). It helped in deploying builds that were smoke-tested, but it did not help at all when we made changes to the screens. And this was exactly what we were doing all the time, because it was easy to do with our generated code and we needed this flexibility for our users! And, of course, there is no way that we could compromise on this, not even to ensure our code was tested automatically. Yes, you are reading this right – we dare to deploy code that is not fully end-to-end tested! We prefer to deploy often and fast, with the risk that our users find a bug from time to time.

Still we were not happy about this, so we kept on looking for better solutions. A long investigation ensued, looking at better (faster) ways to test, ways to generate tests, ways to improve our hand-written code, reading many books and articles on testing (see the books and articles lists below). Even today the search is not over, and we still did not manage to generate our tests (fully). We do run automated tests these days, however, before deploying a new build.

Meanwhile, we would like to share some observations on the various test/development approaches we have encountered:

TDD (Test-Driven Design/Development)

The basic premise of TDD methodology is that your development is essentially driven by a number of (unit) tests. You first write the unit tests, and then run these against your (not yet existent) code. Obviously the tests fail, and your next efforts are directed to writing code that makes the tests pass. TDD and its strong focus on unit testing is, in our opinion, an overrated concept. It is easily explained and therefore quickly attracts enthusiastic followers. The big gap in its approach is that (unit) tests are also code. And this code must be written and maintained, and it will contain bugs. So if we end up on a project where developers spend x% of their time writing new (test) code and not working on writing production code, we have to ask what the point of this is. In our view, you are just producing more lines of code and, therefore, more bugs.

Another problem or weakness of TDD is that there are a lot of bad examples of TDD around. Many sites that advocate TDD give sample code that essentially goes like this: you have a unit test that needs to test a new class. The unit tests usually set a property on the class to be tested and try to read back this property. With any class of reasonable size, this will give quite quickly a nice set of unit tests. But, let's back up a bit, what exactly are we testing here? Well, all things being equal, in such scenarios we are testing whether the code is capable of accepting a value and the possibility of retrieving this value. This comes down to testing the compiler or the interpreter, because, as a developer, the amount of code you wrote (and its complexity) to achieve this behavior is close to zero. In other words, such unit tests only provide a false sense of security, as in the end you are not testing anything.

One argument against this is that over time such getter/setter code may evolve into something more complicated and then the unit test code becomes useful to avoid regression. Our experience shows, however, that a) this rarely happens at a scale large enough to make the initial effort worthwhile and b) if you are making such drastic change to your code, moving from elementary getter/setter pairs to more involved computations, it is very likely that you want your unit test to break.

What is the moral of this story for unit tests? Should we abandon them altogether? No, but we maintain that it requires some thought about what exactly you want to unit-test. 100% unit test coverage for us means we are wasting effort.

Let us take another point. TDD as a concept is not bad, in the sense that it forces you to think about what you actually need to build and how you can get it accepted (tested).

We believe, however, that its fundamental approach puts too much "power" in the hands of the developer. It gives the strong impression that the developer (under time pressure) is tempted to build "something" that passes the test, and that is all. So if the test is wrong, the developer is not to blame! We believe that this undervalues the strength of a good developer and gives anyone with a code editor the chance to position himself as a developer. This cannot be right.

Looking for software testability, we found another issue that we could not accept. It is the burden that the approach of producing "testable software" puts on your architecture. To make a system "testable" is one thing, but to enforce rock-hard principles on the architecture (Inversion of Control, Dependency Injection, or a very strict separation of layers)

in our opinion only dramatically increases the complexity of the code and, very importantly, it serves no purpose for the end objective of the system, namely to solve a business problem. For highly complex systems, such approaches are probably defendable and even very good. However, in the fast-paced, ever-changing world we live in, the least of our problems is whether a software architecture can be sustained for ten years, if we already know that in two years' time the entire business system will be obsolete. We may even argue that in two years' time the insights on what a good architecture should be will have dramatically changed. So why bother with this? We should concentrate on software that works, and preferably that ships in record time.

So we concluded that any testing methodology that requires extensive re-engineering of what is basically a workable and dependable architecture should be looked upon with a certain suspicion. As a consequence, we do not use unit tests that require our code to be able to work without a database. We do not use "mocking" with all its complexities and we do not spend time on making our classes and objects independent of each other. It does not make for the most "correct" code, we know. However, we do not mind. If tomorrow we find a better way to do it, we will change our code templates and simply re-generate our application code (well, most of it). So we are not overly worried about having the right architecture to begin with – in fact we have already changed it twice, but that is another blog.

So does this mean the end of test-driven development? Not at all. There are things you can test very well with unit tests. Plus, there is a new descendant that we have also investigated, and which shows promise for other areas we would want to test: behavior-driven development (BDD).

BDD has the same initial outset as TDD, in the sense that it starts the development process with the definition of the tests the future application will need to pass to be accepted. But BDD is more appropriate for this task because it seems to focus more on the functionality of a system than on how it should be built. So it is less prone to the criticism we have about TDD. For one thing, it provides for a way to bridge the gap between users and developers by using a specific language in which to specify tests (or acceptance criteria, if you wish). This language, Gherkin (*github.com/cucumber/cucumber/wiki/Gherkin*), is so simple that the learning curve is as flat is it comes, meaning that everyone can be taught to understand it in record time. Writing proper Gherkin requires a bit more time.

For us, its main advantage is that Gherkin provides for a way to communicate the functionality of a system at a level that is understandable to a developer. Its main downside is that you will end up with A LOT of Gherkin to fully describe a system of a reasonable size.

In the end, this is the main criticism we have about most of these "methodologies", (UML included). If you have a system that goes beyond a simple calculator (the usual example), no modeling language (as they all are, in a way) is powerful enough to describe a full and complete system in such a way that you can understand and describe it more quickly than by looking at the screens and the code that implements these screens.

So the search goes on ... ■

Read Part II of this article in the December issue (No. 28)!

Referenced books

- "The Cucumber Book" (Wynne and Hellesoy)
- "Application testing with Capybara" (Robbins)
- "Beautiful testing" (Robbins and Riley)
- "Experiences of Test Automation (Graham and Fewster)
- "How Google tests software" (Whittaker, Arbon et al.)
- "Selenium Testing Tools Cookbook" (Gundecha)

Referenced articles

- "Model Driven Software engineering" (Brambilla et al.)
- "Continuous Delivery" (Humble and Farley)
- "Domain Specific Languages" (Fowler)
- "Domain Specific Modeling" (Kelly et al)
- "Language Implementation Patterns" (Parr)

> about the authors

Rudolf de Schipper has extensive experience in project management, consulting, QA, and software development. He has experience in managing large multinational projects and likes working in a team. Rudolf has a strong analytical attitude, with interest in domains such as the public sector, finance and e-business.
He has used object-oriented techniques for design and development in an international context. Apart from the management aspects of IT-related projects, his interests span program management, quality management, and business consulting, as well as architecture and development. Keeping abreast with technical work, Rudolf has worked with the StratEx team, developing the StratEx application (*www.stratexapp.com*), its architecture, and the code generation tool that is used. In the process, he has learned and experienced many of the difficulties related to serious software design and development, including the challenges of testing.
LinkedIn: *be.linkedin.com/pub/rudolf-de-schipper/3/6a9/6a9*

Abdelkrim Boujraf owns companies developing software like StratEx (program and project management) and SIMOGGA (lean operations management). He has more than 15 years of experience in different IT positions, from software engineering to program management in management consulting and software manufacturing firms. His fields of expertise are operational excellence and collaborative consumption. Abdelkrim holds an MBA as well as a master's in IT & Human Sciences. He is a scientific advisor for the Université Libre de Bruxelles where he is expected to provide occasional assistance in teaching or scientific research for spin-offs.
LinkedIn: *be.linkedin.com/in/abdelkrimboujraf*