Agile Testing on an Online Betting Application

Nuno Gouveia(✉)

Blip (Betfair), Rua Heróis e Mártires de Angola, n°59 4°, Porto, Portugal
nuno.estrada@blip.pt

Abstract. Agile development with continuous integration and constant releases is only sustainable followed by a rock solid quality process. At blip/betfair we work very hard to build and continuously improve our quality process to provide at the same time a unique reliability experience to our customers and new features fast. Three major components of this process are: **Mind maps** to help us learn more about our product and represent our knowledge about it in a structure way, **Exploratory Testing** that must be free and creative and happen as soon as possible in the process to allow fast feedback cycles and **CI pipelines** with high levels of automation testing to avoid regression. Agile development with continuous integration and constant releases is only possible with a rock solid quality process.

Keywords: Agile · Testing · Mind maps · Exploratory tests · Continuous integration · Quality

1 Introduction

Before joining Blip in July 2013, all I knew about Agile was based on articles or conversations with other people. After two and a half years of experience, where I have had the freedom to decide the best way to do my job, I feel that my perception about quality in agile environments has evolved a lot. This report shares my journey from learning the existing quality process for the *Exchange Desktop* project, to helping improve it to its current state.

1.1 Background

Blip is part of the *Betfair Group plc*, one of the biggest and most successful online betting companies in the world, being the largest internet betting exchange, and also has a strong position in the traditional online bookmaking business. The growth of the online gambling market in the recent years has led to an increase in the number of online betting websites. This generates a fierce competition for market share which in turn increases the pressure for the companies to be more innovative and to release new features fast in order to draw the attention of as many users as possible. At Blip, the way we deal with this pressure is to use agile methodologies (mostly Scrum) and continuous integration.

H. Sharp and T. Hall (Eds.): XP 2016, LNBIP 251, pp. 193–200, 2016.
DOI: 10.1007/978-3-319-33515-5_16

Although we need to deliver new features fast, our main concern when it comes to the continuous integration of our products is **quality**. Despite being considered an "entertainment" product, our applications deal with an extremely sensitive subject for our customers: their money. With this in mind we dedicate a lot of time to the development of a strong quality process that gives us the confidence to release our products frequently. This process is in constant evolution with the input from Delivery Managers and Developers but primarily from Quality Analysts.

We work mainly with the most popular agile methodology Scrum, in multidisciplinary teams of around 5–7 people, which includes the quality analysts (usually one for each team/we don't have a separate quality department). This integration of quality analysts into the teams means that different projects have different realities and in most cases, the project teams have the freedom to tailor their quality process according to their needs. For this reason, I will only refer to the quality process for the project that I have worked on, for the past 2,5 years, the *Exchange desktop* product that has 2 on-going projects: the *Sports Site Web* (SSW), a Java-based (backend) desktop app, and the *Exchange Desktop Site* (EDS), a newer (1,5 years), Angular-based desktop app.

I will focus on three different subjects within our quality process - *Mind mapping*, *Exploratory Testing* and *Continuous integration pipeline* - and try to describe their evolution over the time I have been in the company.

2 Mind Mapping

2.1 Early Days and Old Process

Back in 2013 when I joined Blip, mind maps were not a part of our quality process. As this was my first full time job as a Quality Analyst (in my previous job, testing was just a small part of my tasks), I started by learning the current process from other QA's from the project. As we work in Scrum with two weeks sprints, the QA process is tightly connected with our Scrum process.

As in most Scrum teams, the first interaction that a QA had with any user story was in the **grooming** ceremony. In this ceremony the Product owner explained the user stories in the backlog to the team. The QA, like any team member, tried to understand any possible flaws with the specification and make sure it was ready to be played. Before the team estimated the user story, it was common for the QA to discuss with the team a general idea of the strategy to test the user story, with special focus on the kind of automated regression tests that could be done. This helped the team estimate the user story with a bit more detail. This part of the process is still in use nowadays, as we feel it works well.

In the beginning of a **sprint** the first goal of the QA was to define test cases (this would happen before any real testing with the application). So for every user story the QA would do the following tasks:

1. Study the user story in more detail with special focus on its acceptance criteria,

2. Create test cases in our agile management tool (*AMT*) for all the scenarios that we would test **manually** as well as different input combinations for the scenarios to validate the acceptance criteria, and
3. Create test cases for all the scenarios that would be automated as regression tests.

Applying this process to all the user stories would usually take at least the first couple of days of the sprint. After a user story was implemented, the QA would test it in a development environment following the test cases that had previously been defined, trying to find bugs. We understood later that doing so was quite restrictive, limiting our creativity and exploration. Our test activities mostly followed the "scripts".

2.2 The Introduction of Mind Maps

In late 2013, one of the more senior QA's shared a new technique he had learned to help him explore the user stories and create a visual representation of our knowledge about a product feature, **mind maps**. This idea blew my mind completely and I immediately started thinking how we could use this in our process. On the following sprint I started experimenting with this idea and created mind maps to help me in my testing strategy for every user story. The use of this technique had a huge and instantaneous impact in the way I worked. Since then, the way we use mind maps has gone through different stages. I will now explain what happened and what we learned along the way.

2.3 Mind Maps for Test Scenarios

The very first thing I tried to do was to use the mind maps to replace step 2 of our process, which was the one I thought was the most inefficient. So instead of documenting every scenario that I was going to test with in different test cases in our *AMT*, I would create a mind map with all that information and attach that single mindmap to the user story. This was helpful at first because it was a much more visual way to represent that information than in separate test cases and it was more practical to use when I was testing the user story, executing those scenarios. Besides using the mind maps to guide me as I was testing the user story, I also started ticking off as complete the different scenarios and steps as I tested them in the app. I would later insert this updated mind map to the respective user story as a test result. This could be used as historical data to be consulted in the future if necessary. After some sprints following this method I realized that all this test result information was useless because nobody really needed it. Not me, not my delivery manager, not the other QA's. So as a good Agile practitioner I stopped doing it because it wasn't adding any value. I also stopped creating mind maps for test scenarios as I started observing that having all those predetermined test scenarios wasn't really helping me find many bugs and was having the same impact in testing that step 2 previously had.

2.4 Mind Maps to Represent User Stories

Around the time I was struggling with the route to pursue next with the mind maps, Michael Bolton came to Blip for the *Rapid Software Testing* workshop. We talked a lot about exploration, experimentation and learning. One technique exhaustively used during the workshop was precisely using mind maps to learn and explore the features and characteristics of a product. This inspired me to use mind maps in the same way, to learn more about the product instead of representing testing scenarios as I was doing before.

I decided from that point on to try to represent the information on the user stories in a more structured way. Instead of mapping scenarios, I started mapping the different components of the module (or small section) that was being implemented and think about the different states that each component could have. Instead of replacing step 2 of the process, this new method helped me with the step 1 - learning as much as possible about the user story. Later, after the team had implemented the user story, I used the mind map that I had created to help me with my exploratory tests. At this point, and with the new concepts about exploration we had learned in the workshop, we (the QA's on the project) decided to drop step 2 of our process, as we understood that it wasn't adding value and didn't really help us discover many bugs when we were testing. This meant that the only test cases that we defined in our *AMT* were those defined in step 3, the ones that would be automated as regression tests.

2.5 Repurposing Mind Maps - The Oracles Breakthrough

Usually when implementing a new module, the work is divided into several user stories. Instead of creating one mind map for each user story, we started creating a single mind map for the entire module and updating it in every user story. These mind maps also included things common to the entire module like the visual specs or google analytics events that would be fired on specific actions. Figure 1 shows one our mind maps.

This process was working fine, but I realized we had a problem: we were creating all these mind maps for every user story and modules and they were very useful while we were testing. But after that, they were stored in a folder and they were never used again. We understood that this was quite wasteful because we had so much valuable information being basically thrown away after the first utilization.

Linking Everything. What I started doing was linking everything. At this point we had a mind map for each module but they were disconnected. So we created a "root" mind map that would reference all the different modules and pages we have in our application. Through this "root" mind map, anyone could access any mind map of any module in a few clicks. For some modules there were more levels of maps. In some cases we can have a module that has different implementations according to the sport we are in (e.g., market header for football, tennis, volleyball, etc.) and in those cases there will be a mind map

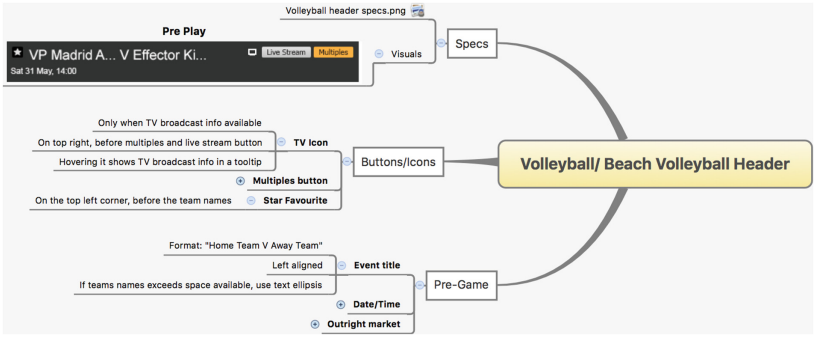

Fig. 1. A simplified version of an actual mind map for the volleyball implementation of the *Sports Header* module

for each of those implementations and one "super" mind map that links all of them together. For more complex modules, we may have the mind map of the module which has "sub" mind maps for different areas or sections of the module.

This concept of linking mind maps really revolutionized the way we used this tool. Most importantly it helped us create something that was missing in our project: a high level vision of our product. Using this powerful tool, we centralized all the relevant information about our product that would otherwise be scattered all around our *AMT*. Mind maps started being used not just by testers but also by product owners, developers or any other person who wants to understand the expected behaviour of any module in our application. We would often see someone asking a question like: "Is this module supposed to appear in this page?" or "Do you remember the rules for this module we implemented a few months ago?" being answered two or three clicks after opening the root mind map. Now we can confidently say that this tool is in fact an **Oracle** for the behaviour of our application.

3 Exploratory Testing

Exploratory testing in its essence has been used in Blip for many years, but it wasn't always called that. Two and a half years ago, the term we used for the tests that were performed by the QA's when a user story was implemented, was *Manual Testing*. Only later, around the same time we started using mind maps, we started using more consistently the term Exploratory testing, which is now completely rooted in our testing vocabulary.

Exploratory testing at Blip has evolved a lot over the years. We made many improvements like introducing new testing techniques we learned or stopping specifying the test cases (for "manual" testing) beforehand, to allow us to explore and experiment with more freedom, without a predetermined script. Those were improvements but the aspect of exploratory testing that I want to talk about is not so much **how** we do it, but **when** we do it.

3.1 Testing After CI Pipeline

Back in 2013, the "exploratory" testing performed by the QA's only happened when the code reached the *alpha* environment (with production data), which was the last step of our CI pipeline. From the moment the developer submitted the code to the pipeline, it would take over one hour to reach this last stage, assuming everything went well (no errors on deployment or broken tests). This wouldn't be a problem... except if the QA found one or more bugs! In those cases the cycle would go back to the beginning, and in the best scenario the QA would be able to test again, more than one hour later (assuming the developer would fix the bug immediately once it was communicated). This was an extremely large feedback cycle and had a huge impact in the amount of time it took to get a user story in front of the Product Owner, and so we decided to change that.

3.2 Setting the Local Environment

Our strategy was to make sure that we as QA's would have the same local environment that developers use to test their code. This way, when development is ready, we simply have to checkout the code from the remote repository and run it on our machines. We always have to make sure that this code is also updated with the latest version of the *master* branch, so that we don't test the new user story on an outdated version of the site.

This process allows us to have the user story in our hands faster as we don't have to wait that it reaches the *alpha* environment. It also allows us to reduce drastically the time for the feedback cycle of any bug that is found - once we find it we can communicate it to developer and after he/she fixes it, we have it again in our hands seconds later.

After the exploratory testing is finished, the code is then submitted to our CI pipeline and once it reaches the *alpha* environment, we perform some sanity tests on the feature just to make sure everything is ok, and then we send it to our PO.

This change we implemented may seem small, but it had a huge impact on the lead time of our user stories and allowed us to have them much earlier in front of our PO for approval. Feedback is an essential part of agile development, especially from the "client" (our PO in this case). The sooner it happens, the better.

4 Continuous Integration Pipeline

Continuous Integration has been present at Blip for a long time now.

Our CI pipelines are a central piece of our quality process because they are the ones that allow us to continuously integrate our code while making sure that no regression has happened in our applications.

Each project at Blip has its own CI pipeline and as in other components of the quality process, it is normal that each project adopts its own strategy for that too.

For the Exchange desktop product we have two ongoing projects, and therefore two pipelines, one for the SSW project and other for the EDS project. In the SSW project we are currently only doing maintenance work, so the pipeline structure hasn't changed in a long time, as well as the test suites that run in it. On the contrary, the EDS project is our currently most active project, which means that has a continuously improving pipeline, both from the regression testing suites and from the pipeline structure itself. For those reasons I will focus on describing the EDS pipeline.

4.1 EDS Pipeline Structure

Figure 2 shows an example of a build in our pipeline.

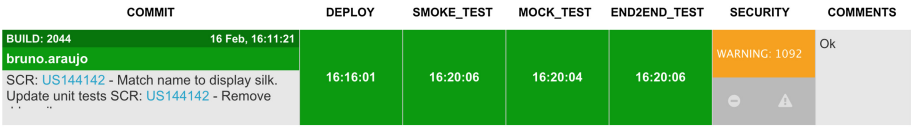

Fig. 2. Example of a build in our EDS pipeline. It shows all the steps: commit, deploy, smoke test, mock test and end2end test which were all successful

When a commit is made to the *master* branch, a new build is generated, triggering the first step of the pipeline.

1. **The first step** of our pipeline basically generates an *RPM* for that version of the code and runs our *unit test* suite against it. The unit tests suite is our largest test suite (over 2000 tests) for obvious reasons: it allows us to evaluate the correctness of each function in the code and with a very low execution time (under 16 s). Coverage can be a misleading metric, but we try to use it as a rule to have at least 80 % branch coverage in our *unit test* suite. After this step the build is ready to be installed in any environment;
2. **The second step** is to deploy our build to our *alpha* test environment. This environment uses production data and can be used both for exploratory testing and automated regression testing. Once this step is completed, the following three steps start simultaneously. They are our UI test automation suites and run against the *alpha* test environment. They used to run sequentially, but now they run in parallel saving a lot of time, specially when all the tests pass.
3. **The third step**, executes our **Smoke test** suite. It runs tests with barely any interaction with the page aside from the assertions themselves (or expectations as we call it) which makes the tests quite fast. This suite is quite small representing roughly **10 %** of our regression strategy.
4. **The fourth step**, runs the **Mock test** suite. These are simulation tests, where we control all the responses from the services, testing mainly two different situations: the modules in isolation (testing all their components and their possible states) and more complex situations that generated or could

generate critical defects. These are the most stable tests once they are not subject to possible service failures and we have total control over the data that is shown on the page. This suite represents around **60 %** of our regression strategy.

5. **The fifth step** executes the **End2End test** suite. These tests represent more "real" scenarios that usually interact with several different modules of the application, not making use of any mocked data. These tests are great to test the integration between modules but are always held hostages of the data that is available in production. This suite represents the remaining **20 %** of our regression strategy.

4.2 Current Challenges

Our current pipeline is far from perfect, and we continuously work to try to improve it. Some of our biggest challenges are:

- **Stability** - This is our number one priority. Our tests aren't as stable as we wished they were, and we have some flakiness at times. Our approach to improve this issue has been to tackle the most unstable tests first, disabling and fixing them. We are making some progress but we still have some tests (the ones in worst shape) that show a failure rate of around 13 % which is not that good.
- **Speed** - The entire pipeline, from the first to the last step takes about 19 min. We are always concerned about this, because we don't want our pipeline to become a delivery bottleneck for our User stories, and as we grow our codebase, the number of regression tests keeps increasing. Our latest improvements were to use as much parallelization as possible: our functional test suites (steps 3 to 5) run simultaneously as well as the suites themselves also parallelize their tests, using a *selenium grid*.

Acknowledgements. First and foremost I would like to thank Blip for being an amazing company that provides a fail-safe environment and for always encouraging us to find better ways to do things. I'd also like to thank my colleagues in my project in particular to the *Avengers* team for always welcoming change and personally to Pedro Tavares who mentored me in my early days at Blip and introduced me to the mind maps concept. Finally I would like to thank my shepherd Rebecca Wirfs-Brock for her thoughtful guidance that made this report possible.