

MASTER'S THESIS | LUND UNIVERSITY 2016

Showstoppers for Continuous Delivery in Small Scale Projects

Jakob Svemar

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-01



Showstoppers for Continuous Delivery in Small Scale Projects

Jakob Svemar

December 22, 2015

Master's thesis work carried out at Softhouse Consulting Öresund AB.

Supervisors: Lars Bendix, lars.bendix@cs.lth.se
Fredrik Stål, fredrik.stal@softhouse.se

Examiner: Ulf Askelund, ulf.askelund@cs.lth.se

Abstract

Small scale projects outsourced to consultants provide their own difficulties when compared to more standard software development. Some of these problems are a lack of infrastructure and customers inexperienced with software development.

This thesis is looking at the possibility of implementing continuous delivery in such an environment. The concrete problems are small projects with very little room for experimentation. But also the inexperience in automated testing which is essential for efficient regression testing. This led this thesis in two directions.

The first one is how can you create a situation where continuous delivery could be beneficial, where developers prefer writing automated test cases instead of performing *Ad Hoc* manual testing during development and relying on a larger testing phase towards the end, much like what is done in waterfall development. The solution is to perform more deliveries to the customer throughout the project, with the customer having the responsibility of providing feedback on these deliveries. For the developers to embrace automated testing, a shift in focus is needed, from functional testing through the GUI to smaller unit and integration tests that will be easier to write and maintain.

The other direction is addressing the fact that there is very little to continuously deliver during early stages of development, which could essentially make up half the project length. But also that there are several small projects each year. Making configuration management a support function for projects allows for standardisation and sharing the cost between all the projects.

Keywords: Continuous Delivery, Continuous Integration, Agile development, manual testing, automated testing, quick feedback

Acknowledgements

First of all, I would like to thank both the academic supervisor, Lars Bendix, and the company supervisor, Fredrik Stål, for ongoing support throughout this process. Without their feedback and assistance it would not have been possible to complete this thesis.

Secondly I would also like to thank everyone at Softhouse for providing the necessary information and making me feel welcome at their company. It is their participation that has allowed this thesis to be completed.

Thirdly, I thank the examiner, Ulf Askelund, for showing interest and taking the time to review this thesis.

Finally, I would like to thank my family for ongoing support throughout my education.

Contents

1	Introduction	9
2	Background	11
2.1	Context	11
2.2	Research Method	12
2.3	Relevant Information	13
2.3.1	Target Audience	13
2.3.2	Continuous Integration	13
2.3.3	Continuous Delivery	13
3	Analysis	15
3.1	Interviews	16
3.2	Perceived Problems	18
3.3	Finding the Real Problems	19
3.3.1	P.P1 We do not know when we are done testing	20
3.3.2	P.P2 We find bugs too late in the development cycle	20
3.3.3	P.P3 None or little regression testing	20
3.3.4	P.P4 Acceptance testing (by customer) finds bugs which should have been caught by the team earlier	21
3.3.5	P.P5 Unstructured testing/inefficient testing, reliance on manual exploratory testing	21
3.3.6	P.P6 Loss of knowledge and lack of time hinders automation of different parts of the process	22
3.3.7	P.P7 Lack of documentation of test cases makes it very unclear of what is actually tested for each version	22
3.3.8	P.P8 Long feedback cycles	22
3.3.9	P.P9 Using Continuous Delivery antipatterns	23
3.4	Digging Deeper	24
3.4.1	Root causes	24
3.5	Gathering Facts	26

3.5.1	Bug report frequency	26
3.5.2	Bug durations	27
3.5.3	Automating Acceptance Testing	28
3.6	Survey	30
3.7	Solution Requirements	30
4	Design	33
4.1	Improving Development	33
4.1.1	More and smaller releases	34
4.1.2	Test ranking	34
4.1.3	Code coverage	35
4.1.4	Refocus project test flow	36
4.1.5	Redesign acceptance tests	37
4.1.6	Standardised commit comment	38
4.1.7	Summary	38
4.2	Making Continuous Delivery Viable	39
4.2.1	Every team manage their own Continuous Delivery	39
4.2.2	Dedicated Continuous Delivery team	40
4.2.3	Dedicated Continuous Delivery support	40
4.2.4	Summary	41
5	Discussion	43
5.1	Evaluating the results	43
5.2	Lessons learned	44
5.2.1	The root cause analysis	44
5.2.2	Designing solutions	45
5.3	Related Work	45
5.3.1	Continuous Delivery: Reliable software releases through build, test and deployment automation[3]	46
5.3.2	Agile Testing: A practical guide for testers and agile teams[5]	46
5.3.3	Container-based Continuous Delivery for Clusters[13]	47
5.3.4	Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy)[14]	48
5.3.5	Enabling Agile Testing Through Continuous Integration[15]	48
5.4	Future Work	49
5.4.1	Dedicated testers	49
5.5	Difficulties with Mobile Testing	50
6	Conclusion	51
	Bibliography	53
A	Interview Questions	55
B	Survey questions	57

Popular Science Article

59

Chapter 1

Introduction

The way software is developed can vary greatly, and different settings create different conditions. Softhouse is a consultant firm that provides a range of services. One of which is in-house development. Their largest team consisting of nine developers, a scrum master and a project manager, is working on ticket solutions for public transportation. This spans several different customers, generally communal or governmental companies, and it has been the main focus during this thesis. Each application is unique but uses similar solutions. This has created a situation where projects run in very short time frames with limited budgets. Projects are won based on tender offers with three main components, a suggestion for the solution, a price and a duration the project will take. This means projects are set up in a traditional manner and shares a lot of similarities with the waterfall model. As trust builds between customers and consultants, more agile approaches have been possible, especially during development.

The purpose of this thesis has been to look at the current development practice, identify bottlenecks for continuous delivery and figure out changes that could help remove them. Solutions are focused on automating testing to reduce the duration of feedback cycles during development. This is a corner stone for continuous delivery to function.

As a side note to the question of what can be done to improve the development, an important question throughout the thesis have been if automated testing and continuous delivery is viable in this limited scale, or if it simply costs too much to be worth it. This thesis explores ways that can help distribute both costs and benefits to all projects.

To do this the following research questions were formulated in the very beginning:

R.Q1 How could increased traceability help highlight if a story is tested sufficiently?

R.Q2 Could a story focused test selection method be useful during development?

R.Q3 How do we link stories and tasks with test cases?

R.Q4 What CM support is needed for this to be a feasible solution?

R.Q5 What level of automation is needed for developers to embrace this functionality?

From discussions in the beginning of the work it became clear that the research questions were not representative for the real problem area and would be difficult to investigate. Two perceived problems were defined and together with my desire to explore the benefits and possibilities of implementing continuous delivery formed the basis for this master thesis. The problem statements were:

- How do we know when we are done testing?
- We find bugs too late in the development cycle.

The first question may seem to have a simple answer, when all bugs are fixed. Or rather, when the software is stable enough as software rarely is completely bug free. For every change in the code, software could have become unstable creating the need to reverify that the code works as expected.

Finding the point where the software is stable enough becomes difficult when relying on manual exploratory testing. It takes a lot of time and is prone to human error. The end results is that it is difficult to measure not only if enough testing has been done, but rather what testing has been done.

Another side effect of manual testing is that regression testing is reduced to a bare minimum. A previously implemented feature may remain untested through several iterations of software. But the team operates on an assumption that the software is in a constant "expected to work" state, only when a bug is found is this assumption changed.

The second statement is closely related to the first one. Bugs are found when testing is done. If the majority of testing is done at the end of a project, the majority of bugs will be found at the end of the project.

The scope is limited to the in-house development at Softhouse and specifically the team creating ticket solutions have been the focus. The project analysed comprises of two mobile applications, one for iOS and one for android. It also includes a back-end server with a browser interface. Other teams and developers have also influenced the findings, but on a much smaller scale. Target audience is developers in similar situation where small projects are common and automated testing is scarce.

Chapter 2

Background

As described in the Introduction, the goal with the thesis was to identify bottlenecks that slow down development and to investigate how and if continuous delivery would be suitable for a specific context.

This chapter will start out describing this context more clearly. Next it will describe the research method that was used to obtain the results and finally will give some relevant information for the reader, like who is the target audience for this thesis and defining what continuous delivery is.

2.1 Context

The context for this thesis is slightly different than for more common development. In this case, common development is defined as companies developing their own product which they then provide for their customers and maintain continuously over a longer period of time.

In contrast, these projects are developed by a consultant firm on behalf of a customer who will then provide it for their customers. They are generally short projects with around five scrum sprints, around ten weeks of development, and around ten developers. Adding to this, development has to combine agile methodology and more traditional waterfall development due to the fact that to even get the project, large parts of the project is planned before even winning the bid over other consultant firms.

As a personal observation, this appears to be a more and more common way of developing small applications, especially for companies that are not experienced in software development. The challenges with this is to maintain an agile practise when everything around the project very much remains rigid with fixed deadlines and budgets.

2.2 Research Method

This thesis has been mainly a theoretical one with the main focus on the root cause analysis. It has been important to gather relevant information from both developers and project leaders and to gather facts that corroborate the claims made.

At the start of the thesis interviews were carried out with developers to find out how the current development was done, and what they found was lacking. This was done to get a basis and find what the perceived problems were as a first step in the root cause analysis.

As the first round in the analysis interviews were chosen to give a qualitative perspective on development as opposed to a quantitative which could have been achieved by sending out a survey. This was done to allow for tailoring the interview to the interviewee, expanding on topics that were found to be particularly interesting by the developer. A vocal answer was also thought to give more elaborate answers in general and thus provide a better basis for the thesis. An added bonus was the fact that it allowed me, the student, to introduce the thesis work and myself to the employees. Something that would not have happened if a survey was answered via a web formulary. A survey could also have been largely ignored, resulting in the need for interviews anyway.

As many questions arose from the initial interviews, a lot of informal discussion were intended to be used throughout the thesis to follow up on questions that might rise. This posed a problem later on as these discussions were not really documented to the extent they should, which made referencing some claims difficult.

After the interviews the results were analysed and the perceived problems found. Then an iterative process was started to find the real problems and ultimately the root causes. This iterative process was supported by comparing findings to other research and literature, to find common ground.

Once the problems were defined a quantitative process to gather evidence of their existence was selected. A quantitative process was chosen here because the intention was to gather statistics. The greater the sample size the more likely the results are accurate. This was done by processing a lot of data gathered from the target project.

First of all the acceptance tests were analysed, to determine if they were possible to automate, or if their design could be changed to make it easier to automate.

Bug reports were analysed to prove that bugs indeed were found towards the end of the project. Which confirmed that development relied on a waterfall approach to testing.

Bug reports that were submitted during development were also analysed to determine how long a bug existed. This was done to find out if the current development suffered from long lead times and whether or not bug fixing was a priority compared to new feature development.

To make sure the solutions stay on topic a set of requirements were created. Finally solutions were designed with a focus on improving iterative release capabilities. The solution suggestions were not evaluated due to a lack of time and are intended to highlight areas that are most important to address if the team decides to move towards continuous delivery.

2.3 Relevant Information

To aid the reader of this thesis, this section will describe the target audience for this thesis, and describe the definition of continuous integration and delivery that is used. The reason for these definitions are simple, to avoid readers making their own interpretation of what it might entail.

2.3.1 Target Audience

The intended target audience for this thesis are people that can relate to the title. They are working in small scale projects and are looking to make a transition from traditional development methods like waterfall towards continuous delivery.

The thesis aims to keep the discussion at a level where people who has studied computer science or has worked with software development.

2.3.2 Continuous Integration

The first one to define continuous integration was Martin Fowler, the definition used in this thesis will be the one he uses[1].

In a short explanation, continuous integration means that developers update and merge their code with the main branch as often as possible, at least ones a day. This helps developers to stay up to date with everyone's changes making the process of merging easier. Continuous integration also include that the code is compiled and tested to verify that it is actually working. By keeping the software in constant working order ensures a stable product and makes releases go more smoothly.

The main thought behind continuous integration is that by merging code often you reduce the risk of breaking the code, and if the code is broken, will make finding the bug easier. It also helps reduce issues like double maintenance which is caused by having developers work on their own branches[2]. By merging often differences between main branches and developer branches are kept at a minimum. A huge part of continuous integration is automating the repetitive steps, in this case building and testing the software. Without automation steps will be skipped or even the entire process of integration will be postponed due to the time it takes to make them.

2.3.3 Continuous Delivery

Continuous delivery is a continuation of continuous integration. Where integration aims to keep the code in working order, and to make development easier, continuous delivery aims to make releases easier by making them smaller and more frequent. The purpose of this is to be able to get more end user feedback but also to make the product more commercially competitive.

Again, just as with continuous integration, automation is the key. By making steps automatic you save time but can also ensure that they are done. Automation also increases the reliability that they are done correctly, provided they were done correctly the first time.

The definition for continuous delivery is taken from the book Continuous Delivery[3] written by Jez Humble and David Farley. In short it means that every commit should be a potential release candidate. This is achieved by creating a deployment pipeline and automating as many of the steps as possible. In an ideal world a change would go through an automatic build process, integration testing, and acceptance testing and finally be ready to be deployed with a push of a button.

Chapter 3

Analysis

As the main part of this thesis, the root cause analysis took a large part of the time and effort. The analysis was done in three steps, defined as perceived problems, real problems and root causes. Several methods to investigate were used as mentioned in 2.2. The goal with the analysis was to create an extensive picture of the development practices used, what problems existed that would prevent continuous delivery and finally figure out potential root causes. During the initial discussions, two perceived problems (P.Px) were already located.

P.P1 We do not know when we are done testing

P.P2 We find bugs too late in the development cycle

The first step was performing interviews with developers and figure out how they worked and what issues they thought might exist in addition to the first two perceived problems. This resulted in a list of perceived problems which would be the stepping stone to uncover the real problems (R.Px).

The second step was digging into the perceived problems, figuring out if there were underlying issues and essentially asking "Why is this a problem?" or "Where does this problem come from?". As these real problems were mainly derived from theoretical reasoning, based on gut feelings, and follow up discussions with project members, it became essential to find some hard facts to confirm the statements.

Once facts were gathered and problems had reasonable proof that they exist, the investigation to find underlying root causes began.

Finally, to be able to create relevant solution proposals, a set of requirements were created. The purpose with these were to keep focus and to some extent narrow the scope of the thesis.

3.1 Interviews

Apart from the initial discussions to formulate of the problem statement, the interviews were the first step in gathering information regarding the current development practises. They were conducted on a face to face basis with a questionnaire prepared to guide the discussion. The intention with the questionnaire was to keep the discussion on topic, but allowing the interviewee to elaborate on topics he or she found interesting. The goal with the interviews were to provide the basis for the root cause analysis in the form of a collection of perceived problems.

In appendix A the questions for the interviews are listed. The people interviewed were working on a few different projects. The main focus was the project dubbed as "Ticket solution", the second target project were working on "Charging stations". Aside from this, one interview was conducted outside of specific projects and one with someone working as a consultant on a customer led project. The idea behind interviewing these other people was to see if the situation was similar on all projects located in-house, and how they compared to projects lead by the customer, essentially acting as some sort of baseline.

The main difference between the in-house development and the consultant working on external projects were the formal rules for development. For the external project the developer had the responsibility to develop unit tests, while integration, gui and acceptance testing was performed by dedicated testers. In contrast, in-house testing was carried out by developers in a format of their choosing. It is however worth mentioning that the scale of projects compared vary, in-house development is focused on small projects with defined end dates while the external projects run for a much larger time span.

The first question, "What is the state of the delivery process in your current project" were designed to get a picture for how a story moves through the development process, where potential bottlenecks may occur. The biggest obstacle was the testing. Both in-house projects relied a lot on manual exploratory testing. In the case of the ticket application, some automated testing was done on the server side while the mobile application was only tested manually. The charging station application was a little better with the use of some unit testing. But again, most testing was done manually. To make time for testing, testing sessions have been introduced. Ideally done twice a week, the team gathered and used the product with the goal to test new features. Developers did however admit that it was not always enforced, and that a reluctance to be "the bad guy" that forces the rest of the team to start testing existed. This lead to the third perceived problem, that no or little regression testing existed.

P.P3 None or little regression testing

At the end of a project there is an acceptance test phase. The second question, "How are acceptance testing performed?" addressed this part of the development cycle. In the charging station project this was done by the customer, while in the ticket solution project both developers and customer were involved. It was estimated that acceptance testing took 80 man hours, e.g. two people one week to perform. As the testing is manual, very little regression testing is done at this point. Acceptance tests were run by developers and any bugs found were fixed. After this the customer took over acceptance testing, and used the same acceptance tests to verify the product. Their acceptance testing was performed in

an production environment, or as close as possible. Customers still found bugs or had remarks regarding design and basic texts that were shown in the application. As far as real bugs go, this creates two possibilities, either the production environment is different, or the customers testing is more thorough. The fourth perceived problem was formalised.

P.P4 Acceptance testing (by customer) finds bugs which should have been caught by the team earlier

Questions three to five were intended to show what traceability that existed between stories, tasks and tests. These questions were directly related to the research questions mentioned in the introduction. Stories and tasks were linked, but it did not result in any particular insight. Due to the focus on manual exploratory testing, very few test cases were created at all during development and hence, no traceability was possible. The heavy focus on manual testing makes regression testing increasingly tedious as the project grows and more and more features are added. There was also not a clear picture of what should be tested and it was up to the developer to define the parameters for the testing, this means that it is easy to forget to test some part of the implementation, and especially miss regression related testing. The fifth perceived problem was located.

P.P5 Unstructured testing/inefficient testing, reliance on manual exploratory testing

As the lack of automation became obvious, the question "Why is the testing not automated?" was added more officially as interviews tended to go in that direction. It was used to try and find the reason to why manual testing was preferred. There were a few reasons for it. The first one was rooted in the fact that automated testing lacks the human touch, it can only evaluate a specific variable, and loses the full picture. This is closely related to the second reason, if you still have to perform manual testing to be sure everything is working as intended, why waste time implementing automated testing too?

The third reason is related to the difficulty of automating testing. As the developers perceived themselves as inexperienced with it, it simply took too much time to get started. This originates from the desire to continually produce something for the customer, creating test suites is something the customer gets no direct value from, hence, spending time on it results in slower progress, at least initially.

Previous attempts to automate testing, especially GUI testing, had been done by individuals, but never really caught on, and when the specific developer left the team the knowledge was lost. This creates the sixth perceived problem.

P.P6 Loss of knowledge and lack of time hinders automation of different parts of the process

The seventh question, "Have you been in projects with very long testing cycles?" did not yield any particular results, developers in the ticket solution project felt that testing during development was quick, which could be an indication on lack of regression testing. It also mentioned that the use of testing hours to verify a feature or bug fix could add up to two or three days in lead times.

The last question, "Are there any changes you would like to see?" was designed to

give the interviewee a chance to bring up anything that might have been missed during the interview or emphasise something they felt should change. It resulted in developers mentioning some simple tools that could help and recognitions of problems in the current development. A summary of the results can be presented in four short bullets.

- Introduce code coverage.
- Less *Ad Hoc* Testing.
- Build servers, to encourage automated tests.
- Automated gui testing requires too much work that the customer does not see.

These suggestions from the developers were taken into consideration when different solutions were considered. These suggestions show that there is an interest from the developers to improve, and could further indicate that they feel pressured by time constraints to actively improve their practises.

3.2 Perceived Problems

During the initial discussions the first two perceived problems were stated and when interviews were completed, six perceived problems were discovered. Additionally, some hypothesis of my own had been created, which were based on observations and discussions with different employees and the thesis supervisor at the company.

P.P7 Lack of documentation of test cases makes it very unclear of what is actually tested for each version

P.P8 Long feedback cycles

P.P7 came from the interviews where it became clear that testing was done in an *Ad Hoc* way. It also ties in with the original problem statement and P.P1. With *Ad Hoc* testing the only thing that remains after a test run is the fact that the developer think he or she is done. In contrast to an automated test suite where there is a log, for instance Jenkins which clearly states that a build passed and what test cases were run. Further discussions revealed that once a feature is added to the main repository, it becomes very unclear whether or not that feature is covered during regression testing. Agile development talks a lot about reducing the job of documentation, but not the need for it. Instead writing automated test cases provides the necessary documentation that otherwise has to be handled manually.

P.P8 came from discussions with my supervisor at the company. Feedback is essential in agile development, it is even mentioned as a fundamental value in the Extreme Programming methodology[4]. The sooner feedback can be received the better. As mentioned in the Continuous Delivery book[3][p. 61], initial feedback on a commit should come in less than five, or at most ten minutes. But also feedback between customer and developer is important, even if it is difficult to reduce these lead times to mere minutes. Early approval from the customer can reduce the impact of what a change request could have on development as other stories should not yet be based on what the customer wants to change.

With many of the perceived problems relating to testing, it started to become obvious that one of the bigger issues involved the way quality assurance was done. But looking at the suggestions from the interviews and referencing the continuous delivery book, a number of antipatterns was noticed[3][p. 7]. Especially the use of manual releases was discovered. With manual releases, work is not only slowed down, but it increases the risk for making mistakes due to repetitive tasks. It is very unlikely that a computer makes a mistake during the release process if an automated script exists and have been tested. These antipatterns creates the final perceived problem.

P.P9 Using Continuous Delivery antipatterns

The total result were nine perceived problems that should be further investigated to try and identify the real problems that caused these symptoms.

- P.P1** We do not know when we are done testing
- P.P2** We find bugs too late in the development cycle
- P.P3** None or little regression testing
- P.P4** Acceptance testing (by customer) finds bugs which should have been caught by the team earlier
- P.P5** Unstructured testing/inefficient testing, reliance on manual exploratory testing
- P.P6** Loss of knowledge and lack of time hinders automation of different parts of the process
- P.P7** Lack of documentation of test cases makes it very unclear of what is actually tested for each version
- P.P8** Long feedback cycles
- P.P9** Using Continuous Delivery antipatterns

3.3 Finding the Real Problems

Once the interviews and the analysis of the interviews was done and had been summarised to nine perceived problems it was time to start the root cause analysis. The first step was to go over each perceived problem and figure out why it was seen as a problem, and if there were any underlying problems that might have caused it. Once the real problems had been discovered it became necessary to narrow down the problems to their specific root causes. The solutions in chapter 4 will focus on addressing the underlying root causes and not the symptoms noticed as perceived problems.

As this part of the analysis is done mainly from a theoretical stand point, the need to back up the statements with facts becomes relevant. Section 3.5 presents data gathered from the ticket solution project with the intention to prove that problems exist.

3.3.1 P.P1 We do not know when we are done testing

A very simple way of knowing when you are done testing is when you know there are no more bugs in the code. However being 100 percent sure is not a realistic goal. Instead, compromises have to be done and some kind of definition of working has to be done. This is where testing comes in, by designing test cases, either on unit, integration or acceptance level, these test cases define what the code is expected to do. Looking at the **P.P1** and in the context of the ticket solution project. It is not possible to know when testing is done until two things occur. First, test cases designed to test the implemented feature need to exist, and secondly, these test cases need to pass. The perceived problem of not knowing when testing is done can be traced to a real problem (R.Px) of not defining what should and what should not work for a story to be completed.

R.P1 Acceptance tests not developed in conjunction with features/stories

3.3.2 P.P2 We find bugs too late in the development cycle

Bugs are found when the software is tested. If bugs are mainly found at the end of the project, the testing during development is most likely not good enough. It could have to do with issues like development environment being different from runtime environment, but since the interviews revealed that acceptance test cases are not written until late in the project, the biggest issue is most likely that testing is not prioritised until later stages of development.

R.P2 Testing is not a priority until closer to release

3.3.3 P.P3 None or little regression testing

When asking the question "Why is there no regression testing?", the only apparent answer is because it is too time consuming or boring. It is not a feasible solution to spend hours to test an application each commit. As the application grows, so does the amount of regression testing needed. When relying on manual exploratory testing, some regression is indeed covered, but since it leaves no trace that the application is actually tested it can be considered unreliable, especially if it is the only type of testing.

R.P3 Manual test selection and execution is the only quality assurance

3.3.4 P.P4 Acceptance testing (by customer) finds bugs which should have been caught by the team earlier

Having customers find bugs involves two problems. The first one is that the quality assurance process is not good enough and the second is that the customers expectations were not the same as the developers. The first one is directly related to how the testing process is performed, both **R.P1** and **R.P3** are related to this.

The second one, that they expected something else, their idea of a feature differs from what the developers thought the customer wanted becomes a problem if this is discovered late in the development process. But if they are found early, they become prime examples of why agile development is advantageous in comparison to traditional waterfall development. When the customers verification process is mainly located in the end of the project, changes, whether they are bugs or design choices, could become more costly. More specifically the question "is a story actually done if the customer has not approved of it yet?" can be asked. Again this ties in with **R.P1** but more specifically it raises the problem that stories are not actually done until the customers acceptance testing is completed. Instead they are left in some kind of "Hopefully done" state.

R.P4 Stories are not completely done until end of development

3.3.5 P.P5 Unstructured testing/inefficient testing, reliance on manual exploratory testing

It is common for a human to forget something, especially if it is a task that is to be performed several times over and over. When the testing is performed *Ad Hoc*, mistakes are bound to happen. Combining this with continuous delivery, or even continuous integration raises a problem, "Do not check in on a broken build" [3][p.66]. But how do you know if the build is broken or not? The answer is simple, you test the build! Every change should be tested to verify that previous functionality have not been broken, and that the new functionality is working as intended. Without testing with clear results, continuous integration becomes unreliable. And consequently, the software is not always kept in a working order, ready for delivery.

R.P5 Lacking continuous integration environment to implement continuous delivery

3.3.6 P.P6 Loss of knowledge and lack of time hinders automation of different parts of the process

Setting up automated testing, build servers, or even to learn how to write good test cases takes time. The main advantage of automated test cases are that once they are done they can run often and cheaply. When automated testing takes a full projects time to be set up, the question "Will this bring any benefit" rises. The most apparent answer is no, the current project most likely will not benefit. At least not enough to compensate the cost to implement it. The next project however may.

When isolated attempts to improve testing fail before their effects can be properly analysed, improvement will never happen. There need to be time to explore new options and it needs to be embraced on a larger scale. The real problem is similar to the perceived problem, lack of time, and/or resources hinders the implementation of continuous delivery.

R.P6 No prioritisation to implement continuous delivery within a project

3.3.7 P.P7 Lack of documentation of test cases makes it very unclear of what is actually tested for each version

Lack of documentation can be perceived to save time in the short term. But not knowing for a fact that features work as intended makes future testing more difficult, resulting in more work towards the end of the project. The problem with documentation is that it has to be done in one way or an other. Performing manual acceptance testing for each commit is simply not an acceptable solution, this has lead to a reliance on exploratory testing.

In the end, the problem of not knowing what is tested culminates in the fact that acceptance testing in the end of the project takes around 80 hours to perform, which makes continuous delivery virtually impossible.

R.P7 Acceptance test is 80 hours of manual labour and manual testing forms the foundation for all testing

3.3.8 P.P8 Long feedback cycles

There are several different stages of feedback, the initial feedback on a commit, feedback that the complete story is working as intended and finally that the customer agrees with the developers interpretation of the story. The real problems that cover these points have already been listed, manual testing delays feedback (**R.P3**). It is not possible to perform a sufficient amount of testing in five to ten minutes.

Not having the customer validate stories until the end of the project (**R.P4**) results in changes that should have happened early may not be requested until very late in the project. Finally, by not designing acceptance tests in conjunction with development (**R.P1**) the customers job to validate the implementation before the final release is made more difficult.

3.3.9 P.P9 Using Continuous Delivery antipatterns

Releasing often means repeating tasks. And repeatable tasks should be automated [3][p.25]. Automating processes takes time and knowledge, and commitment to achieve it (**R.P6**).

3.4 Digging Deeper

Looking back at the real problems discovered, some of them look quite similar. Several problems are focused on testing happening in the end of the project. These are **R.P1**, **R.P2**, **R.P4**. Instead of treating each problem as a separated issue it might be better to look at them as all connected to the same problem. Seeing testing as the last step in development is commonly referred to as the waterfall model [5][p.13]. Having the ambition to have a working product in the early iterations of a product, and adding new functionality as time goes seem to conflict with this. As such these real problems could be combined to one new and bigger problem.

R.P8 Mixing agile code development with waterfall testing

Another thing noticed during the analysis of the perceived problems and what the underlying real problems could be, were the focus on manual testing (**R.P3**).

When looking at how manual testing could be removed the answer is automate it. And to do that **R.P6**, lack of prioritisation of improving development becomes relevant. Another way of looking at it is the fact that automated GUI testing is generally perceived as expensive and difficult to maintain. This was even brought up during the interviews and raised the question, "Why automate the most difficult type of testing?". The final real problem is discovered.

R.P9 Testing is focused on functional gui testing

Gathering all the real problems gives the following list. As they have been gathered through mainly theoretical reasoning and some follow up discussions one more step before finding root causes is necessary. The problems existence need to be confirmed by looking at actual data from the projects.

R.P3 Manual test selection and execution is the only quality assurance

R.P5 Lacking continuous integration environment to implement continuous delivery

R.P6 No prioritisation to implement continuous delivery within a project

R.P7 Acceptance test is 80 hours of manual labour and manual testing forms the foundation for all testing

R.P8 Mixing agile code development with waterfall testing

R.P9 Testing is focused on functional gui testing

3.4.1 Root causes

Once the real problems had been theorised from the perceived problems it was time to find the root causes (R.Cx). These causes define the underlying problem and the idea behind identifying them is to, if using a medical metaphor, to treat the cause of the illness and not the symptom.

The root cause analysis resulted in three causes, two were related to the way quality assurance was done and one that was related to the commitment of improving development.

Looking at the agile testing triangle in figure 3.1 shows a basis in unit and integration testing with a top of acceptance and finally manual testing. In contrast, testing in the ticket solution project is centred around manual functional GUI testing (**R.P9** and **R.P3**), and attempts to automate have focused on automating those tests instead of building a foundation with unit and integration tests which are generally considered cheaper to run and maintain.

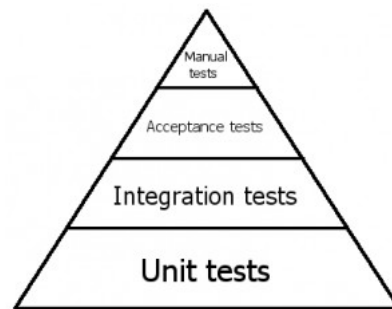


Figure 3.1: The testing triangle illustrates a recommended division of focus for different types of tests.

Additionally the majority of testing have been postponed to the end of projects (**R.P8**) This indicates that the developers do not see the advantage of continuously evaluate the software through automated unit and integration testing, something that could be considered a corner stone in agile development[5][p.25] as well as continuous integration and delivery [3][p.11].

Finally, not seeing the advantage of automated testing could provide an explanation why they are not focusing on implementing continuous integration (**R.P5**).

R.C1 Developers do not see an advantage with automating unit and integration testing

With the heavy focus on manual testing, acceptance testing has become a procedure that takes a lot of time, and is often pushed to the end of the project (**R.P7**), as the process of performing acceptance testing is too time consuming to be done over and over. This results in the waterfall based testing approach mentioned as **R.P8**.

It is noticeable that testing occurs when it is time for a release, the importance to provide working software is clearly understood by the developers. However the current release format and the general dislike for testing has led to testing being used more as a way of confirming that the software works and not as a way to receive feedback. Verification where for instance automated testing gives a clear answer, and validation, where the customer confirms that expectations have been met are valuable throughout development and not just in the end.

R.C2 Quality assurance is tied to releases

The final root cause is directly related to **R.P6**. Developers feel that there is no room

to focus on improving the development practise because the value is often not seen until later projects, which makes it difficult to add the cost to a project that will not benefit from it. This is directly related to the specific context of developing at a consultant firm where every project has a different customer.

This results in three root causes that will be the target when designing a solution.

- R.C1** Developers do not see an advantage with automating unit and integration testing
- R.C2** Quality assurance is tied to releases
- R.C3** Improving development practices are not prioritised

3.5 Gathering Facts

As a lot of the root cause analysis was based on interviews with developers there was a lot of theorising. To make sure that the problems actually exist some proof is necessary. To support the claims in this report about perceived and real problems as well as root causes, bug reports from developers, acceptance test reports from the customer and project plans were analysed. The actual acceptance test cases were also analysed to look at the possibility of automating them.

3.5.1 Bug report frequency

To confirm the assertion that bugs are found late in development an analysis of all bug reports in the ticket solution project was done. Apart from a surge in bug reports in the middle of the project, over 40% of all bugs were reported after the product went to customer acceptance testing on June 1st. The histogram of all bugs can be seen in figure 3.2. The surge in the middle of the figure represent the first time the user interface was integrated with the other services and manual testing became simpler. It is worth to note that not all of these reports are bugs, it includes misunderstandings, duplicates and spelling mistakes. Another possible reason for the the uneven distribution is that small bugs discovered by developers may be fixed without ever being reported.

Out of the data from the customer acceptance testing, the data in table 3.1 was gathered. Out of all the reports from the customers, 64% appeared to be bugs with clearly faulty behaviour. Out of those actual bugs, 64% were reproducible and as they did not appear to be related to differences in development and production environment, should have been caught by the developers before the customer began their acceptance testing. These bug reports reinforce the conclusion that bugs are found late in development and the focus on manual testing has led to a waterfall approach to testing (**R.P8**).

Not testing early could lead to a couple of different problems. The first one is the fact that the product may be faulty. This in turn may cause future functionalities to be based on faulty code, but it may also make developers less inclined to release software to customers. The high amount of superficial changes (23%) like rephrasing texts indicate that there is a lack of customer feedback during the sprints before release.

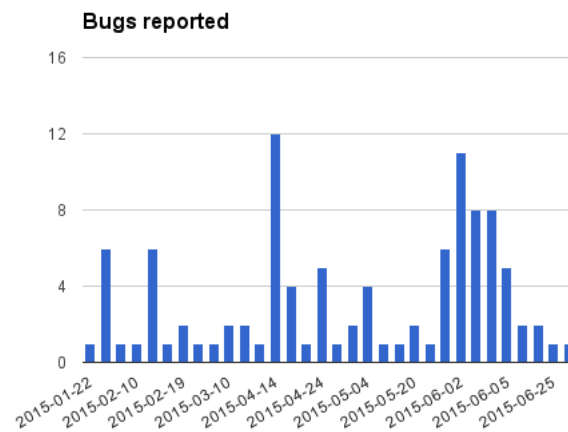


Figure 3.2: All bugs reported during the ticket solution project

Actual bugs	64%
Percentage of actual bugs the developers should have detected	64%
Spelling mistakes or rephrasing sentences	23%

Table 3.1: This table summarises the analysis of the bugs submitted by the customer in the ticket solution project.

The second problem could be with the actual testing. Designing acceptance test cases at the end of the project could decrease the quality of the test cases, and make testers rely more on exploratory testing to actually find bugs. This is an inefficient way of testing, but also adds a stress element to developers by creating the uncertainty of not knowing when there is enough testing[6].

3.5.2 Bug durations

The last data gathered about bug reports were the time it took to fix them after they were discovered. In figure 3.3 all bugs reported are displayed. They show that most bugs are fixed in less than 12 days, but also that there are bugs that can live for upwards of 60 work days. In figure 3.4 the three extreme bugs are removed to provide a better view of the general life of bugs. It shows that few bugs are fixed within three days and most are fixed within ten days.

Average days for a bugfix	10
Average days without outliers	8

Table 3.2: This table shows the average days for a bug to be fixed in the ticket solution project.

It is very difficult to gauge the difficulty of fixing a bug, but with averages of close to two working weeks indicates that feedback cycles are rather slow or that bug fixing is not prioritised. As a comparison, the lead times for a story in the current project (not the same as the bug data was collected from) are very similar, they are added during a sprint and are not really considered done until the end of the sprint which gives a rough estimate of eight to ten working days. It is important to note that any real definition of done can not be achieved until a story is acceptance tested which happen at the end of the project.

When gathering this data, attempts to track bugs through the commit comments were made to make some kind of measurement of the complexity of bugs and confirm if the long lead times were related to the difficulty of fixing bugs or rather the priority of fixing and approving them. Due to rather bad commit comments these efforts yielded no particular results. One possible reason for the long durations could be the use of testing hours. A fixed bug could be considered broken for an additional two days, more if a testing hour is skipped or postponed, before it gets crossed off. Further more, the time a bug may live could be increased by the fact that bugs are discovered in chunks, as seen in figure 3.2.

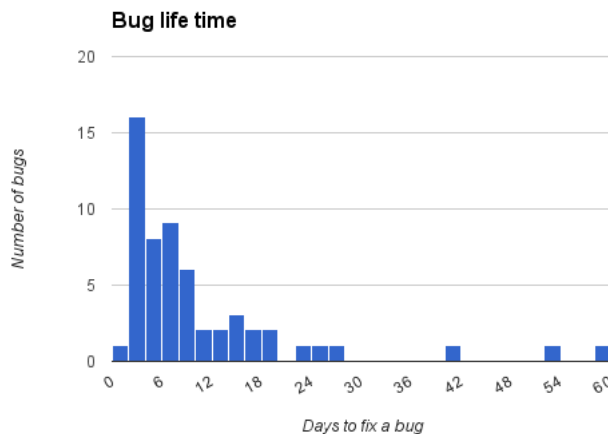


Figure 3.3: Display the frequency of bugs that live for X days.

3.5.3 Automating Acceptance Testing

As acceptance testing in the ticket solution project were a rather time consuming endeavour that could take around 80 hours of manual work (**R.P7**), it becomes one of the big bottlenecks for implementing continuous delivery.

An analysis of what acceptance tests that can be automated and what they actually test were intended to give a picture of whether or not it is useful to implement automated testing at different levels like unit, integration or GUI. The results of this analysis can be seen in table 3.3. It quickly became apparent that unit test cases generally could be seen as too small to be able to cover acceptance tests. This is not saying that they do not provide usefulness, only that they are less useful at the acceptance testing level.

A large chunk of the acceptance test cases are centred around testing a specific feature (33,6%). For instance "Buy a ticket". While the acceptance test starts with navigating through different views and ends with a visual verification that the ticket is received and

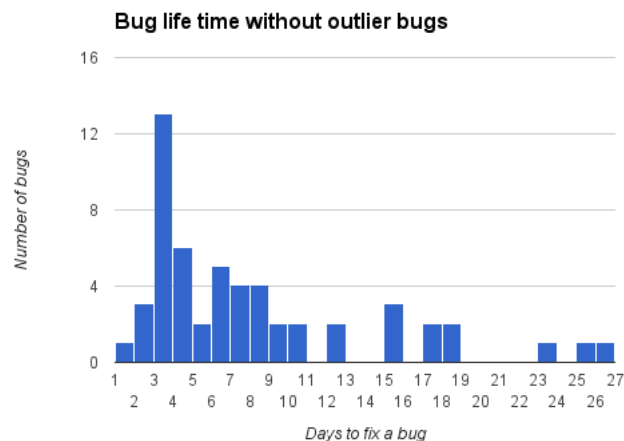


Figure 3.4: Display the frequency of bugs that live for X days without three bugs that had abnormal long life.

that it remains after the application is rebooted. The core functionality of actually receiving a ticket could be tested without the graphical user interface. Verifying that a ticket is displayed correctly on the client phone would then have to be a separate acceptance test that needs to be run manually or automatically, but less often.

The evaluation of automating acceptance tests through the GUI is based on discussions with developers regarding what can and what can not be done within reasonable expectations. One issue that was noticed during this evaluation were that the application needs to perform in a specific way when it is offline and it needs to be protected against manipulations of the internal clock. While these features can be checked automatically, they still add another layer of complexity to already difficult automation tasks, especially the clock manipulation appears to be rather difficult to automate. A preliminary analysis indicate that 50,4% the acceptance tests could be accomplished through automated testing. If offline mode and clock manipulation proves to be simpler then expected, this number could go up an additional 12%.

Finally, just below 40% of all acceptance tests seems more or less impossible to automate, or simply to unreliable for them to give any beneficial results.

Functional tests without GUI	33,6%
Automated through GUI	62,4%
Automated through GUI without clock/flight mode manipulation	50,4%
Purely manual tests	37,6%

Table 3.3: This table contains estimates on the amount of acceptance tests that can be automated.

3.6 Survey

Towards the end of the thesis project, a short survey was sent out to both in-house and consultants working on customer sites to gauge their perceived knowledge about testing and SCM. Questions can be seen in appendix B. The goal with the survey was to confirm a hunch that development in-house is less organised and does not set aside time to experiment with, for instance automated testing.

Participation in the survey was quite limited, only six in-house developers and six developers on customer sites as well as three scrum masters or agile coaches. This meant that the results are very unreliable and could not be used for much more than a basis for further investigation. The one outstanding result from the survey was that in-house developers in general felt less secure with dealing with SCM and test related tasks. This could indicate that project schedules do not include enough time to learn, or setting up working continuous delivery flows but teams are still expected to solve it themselves, which is summarised in **R.P6**.

3.7 Solution Requirements

With root causes formulated and facts gathered to strengthen the claims it was time to create some suggestions for solutions. To better do that a set of requirements were created to help focus the solution into the desired direction. As the root cause analysis revealed problems with testing, and indicates that there are issues with continuous integration focus will be put on improving that aspect of the delivery pipeline. Another aspect of the delivery pipeline, which in this report will be considered out of scope is automated deployment. There simply were not time to do both. That said, having an automated deployment pipeline would probably be beneficial, especially down the line if more frequent releases gets implemented.

The first requirement is focused on increasing the communication between developers and customers. The customer should be getting frequent updates of the software and the developers should get frequent feedback on their implementations. Feedback is a corner stone in agile development and continuous delivery is an ideal setting where frequent communication is intended.

Req: Increase feedback between developers and customers.

To have the confidence to release often to customer the quality needs to be ensured. This means testing is required for every release at the least, ideally it should be for every commit. This gives the second requirement, that evaluation of the software should be possible continuously.

Req: Allow for a continuous evaluation of the software

To be able to further improve development, analysing data is an important part in discovering where bottlenecks exist. As mentioned in 3.5.2, it was difficult to track bugfixes due to bad commit comments. This made the analysis of why bug lives had high mean

values close to impossible. The third requirement is that any solution suggestion that is implemented should be easy to evaluate for future reference.

Req: The solution should be easy to evaluate

As the context for development is a bit different then from more conventional production companies, the need for preserving knowledge between different projects is increased. Further more the scale of projects can create situations where teams frequently change. Most importantly is probably that changes may not benefit the project they are first implemented in. Changes that directly benefit the current project are much easier to "sell" to a customer. This creates the final requirement that the solution need to focus on preserving knowledge from one project to another.

Req: Focus on improvements that are not unique for one project

This results in four requirements that are meant to guide the solutions towards the desired goal of a more iterative release process and start taking steps towards continuous delivery.

Chapter 4

Design

With the root causes defined in 3.4.1 and the scope limited with the help of requirements in 3.7 the work to address the problems began. As the root causes hint at two different problems, quality assurance and the ongoing process of improvement, so will the solutions be divided into two main parts. Epics if you will.

The first one is related to **R.C1** and **R.C2** and will focus on a more iterative process with smaller and more frequent releases. What solutions work the best in combination and whether or not it suits the current situation. In short it provides some suggestions to start moving towards continuous delivery.

The second part deals with **R.C3** and the costs of implementing these delivery pipelines. Providing reasoning for questions like "What setup is the most beneficial?" and "Could it be worth it?" are the goal.

4.1 Improving Development

As mentioned in the introduction to this chapter, the first part of the solutions focused on solving **R.C1** and **R.C2**. As the design process started it became apparent that some further analysis was needed. The question "Why do the developers not see an advantage with automated testing?" was asked.

The always present problem of lack of experience is recognised, not being confident to write good tests would definitely reduce willingness to write these tests. When looking at waterfall development, there are several stages where specialists perform different tasks, requirements engineering, development, testing. With agile development these lines all blur and developers need to adopt these other roles as well.

The main part of the problem most likely stems from that the current development process does not take advantage of being able to release often. If a release does not give good feedback to developers the reason for a release, from a developer perspective, becomes pointless. This was confirmed by looking at project plans for different ticket solution

projects. Even though each sprint ends with a demonstration, the customer was not required to actively give feedback on implementations. In short, developers only made one real release, and testing is used to ensure that the release was working.

To address this issue, this part of the solution is focused on increasing the frequency of releases but also make a release more valuable in terms of feedback to the developers.

4.1.1 More and smaller releases

The first step towards continuous delivery is to increase the frequency of releases. At the very least each sprint should be regarded as a release which is delivered to the customer, but not necessarily pushed to production. To achieve continuous delivery every story should be a potential candidate for release. This is probably not a sensible solution today due to the heavy load of manual acceptance testing, but every story should be stable enough to be considered a release candidate ready for manual acceptance testing, the final stage before a product goes into production. To achieve this, and to be able to take advantage of smaller releases, several of the other suggestions to improve development are focused around testing, and mainly automating parts of the testing to reduce the load of manual testing to make releases quicker, and benefit more from more releases.

As stated before, a release candidate does not have to go into production, but it should be sent to a customer and stories that are considered done need to get final approval from the customer. For this to work there needs to be a customer on the other end ready to receive a release, something that needs to be detailed in the project plans and agreed between customer and developer.

A good starting point would be to end every sprint with a release, where the customer is obligated to sign off on the specific stories implemented that sprint. This is relevant for both agile development but also fixed price projects, if the customer changes his or her mind later on they need to perform a change request.

To feel confident enough to release something to the customer, testing is needed. With more frequent releases, test runs needs to increase too. This is where automation comes in, it does not matter how many releases exist if steps are automated. With a foundation of automated tests, manual exploratory testing can be focused on areas that are affected by the new changes. Leaving other areas reasonably tested by automatic tests. A sensible place to start would be to automate unit and integration testing, and then acceptance testing. Down the line automating the release pipeline would help a great deal too, but that is something that is considered out of scope for this thesis.

4.1.2 Test ranking

With manual testing being as prevalent as it is, some way to reduce amount of testing required at different stages of development is necessary. Less testing means lower quality, and it becomes more important to choose relevant test cases that could catch defects in the new changes. As mentioned before, an estimation for acceptance testing is around 80 hours of manual work. This quickly becomes unsustainable if any form of increase in releases is attempted.

One solution is to not run all acceptance tests all the time, but rather to perform a basic test selection to try to identify relevant test cases. This is of course a risky approach, not

testing brings uncertainty. It relies on the assumption that recent changes have not broken past functionalities, preferably this assumption can be strengthened by having some lower level tests automated.

A simple way of categorise test cases is shown in table 4.1, the next step is to define what constitutes a rank one and what constitutes a rank four.

There is always the ever present issue of security, for the ticket solution project making sure that payments are done correctly would be considered extremely important.

Another very important acceptance test is to make sure that normal user behaviour works. Again, buying a ticket, but not just confirming that the payment goes through, but that the user can actually buy the ticket, receive and activate it, and that it is displayed correctly.

A less important function could be related to real time updates to traffic delays. Definitely a nice feature, but in the grand scheme of things less important.

Finally, test cases that are the least important to run could be test cases for analytical functions, a bug in displaying mean values of tickets sold is not a show stopper.

run very often (1)
run often (2)
run every now and then (3)
run only at big releases (4)

Table 4.1: A simple way of categorising test cases

Having a list of the most important test cases has an additional benefit, they are prime candidates to automate, after all, doing a repetitive task is one of the best reasons for automating something. It could even be worth it to automate GUI testing for those.

These selection examples are focused on an constant importance of test cases. This is not always true, relevance of test cases increase and decrease depending on where a change occurred. This relates to one of the research questions posed in the beginning of this thesis, but due to an extended root cause analysis was not fully investigated. It remains clear that some adaptation of the test suite based on the actual changes is advantageous, and a good starting point would be to run test cases directly related to the current story and stories directly related to it.

As a final conclusion for test ranking, if manual testing remains as an important part some sort of selection criteria is necessary to reduce the test load but still make sure the application is working. It is also necessary to have the testing structured and documented to know what is working and how to test a specific feature. This is also necessary if test ranking is to be implemented, and test ranking gives the added benefit of indicating what test cases should be automated.

Test selection is one way to reduce test load, a better solution is to automate it and keep the complete coverage. In the real world some sort of combination would most likely be the most effective one.

4.1.3 Code coverage

A requested feature to help with test design has been code coverage. During the process for investigating the value with code coverage two research papers were reviewed. There

are some advantages, as mentioned in a research paper by Stefan Berner et al[7].

One of these advantages were detecting new defects, mainly those not related to "blue skies", or as this thesis calls it, normal flow. Code coverage helps with this by highlighting the fact that error handling code is not accessed during testing.

The second interesting advantage was the complete opposite, by measuring code coverage a reduction in test cases testing the same code occurred. This is great when dealing with a changing code base, and keeps changes to the test code base to a minimum.

The paper goes on to list lessons learned. There were a lot of positive results, but one thing that stood out as very relevant for this thesis and the situation in the ticket solution project. It was their word of caution, "Do not introduce coverage analysis and visualization tools in projects without a reasonably usable automated test suite."

During the interviews it was discovered that the server part of the system had some automated testing. This was mainly to test the normal flow of the program, but it means that the server could be ready to take advantage of code coverage, but the mobile applications are not.

The second paper[8] appears a bit more negative, it raises the issue of developers trusting the coverage too much. This is something that is addressed in the first paper as well, it is important to be clear with what the coverage is supposed to help with, just using it to achieve code coverage is not an effective testing method.

As a final recommendation, do not implement test coverage until an automated test suite is in place. Use it to find areas of the code that are untested and finally, remember that 100% test coverage does not equal a well tested product.

4.1.4 Refocus project test flow

As shown in the testing triangle in figure 3.1, agile development relies on a foundation of easy to execute automated test cases to give developers quick feedback that their code is working as expected. While these test cases does not guarantee that the product as a whole will work, it increases the likelihood that two working components will work together as opposed to two untested components will work together. This contradicts the way testing is done in the ticket solution project where focus on automated testing is very low, especially on the mobile application side of the implementation.

Advantages with unit tests is that it gives confirmation that the core functionality work before manual testing begin. Simply put, it is pointless to manually test a product that is broken on a basic level. As the design pattern model-view-controller is being used, adopting automated unit and integration testing on the model should be fairly easy and a great starting point for automated testing. A logical next step would be to automate tests for the controller while leaving the viewer to last due to the inherent difficulty in automatically testing graphical user interfaces.

It could also be good to adopt test driven development (TDD), both for unit and integration testing, but maybe even when creating acceptance tests, even if it is on an abstract level. In a research paper based on development at IBM[9] the effects of TDD are summarised with generally positive results. One of the most interesting points was the point about defect rate. "With TDD, unit testing actually happens" coupled with a 50% improvement rate in functional defects detected. One of the most claimed positive effects from TDD is usually better design. This is only addressed in passing, no conclusive evi-

dence that TDD has had a positive effect. The actual design advantages with TDD seem to be rather difficult to quantify, mostly because design covers a rather large spectrum of measurable factors. One thing that both the IBM paper and a paper by David Janzen and Hossein Saiedian[10] mention that complexity is one factor that is improved by TDD.

The paper mention that there is initially a slight drop in productivity, but that in the end this is negligible. It is important to remember that once an automated test case is written it can be used to confirm functionality over and over again without any (almost) additional cost, and it should be used over and over to get the full benefit.

By utilising TDD even for acceptance tests both platforms (android and IOS) will know what is required from the start, in stead of designing the test cases afterwards and adapting either one of the solutions so that they work in the same way. It will also allow the customer to validate user flows before development start which is advantageous since changing an acceptance test is a lot easier then changing the whole application.

To refocus the testing flow is the best change that can be made to make development more agile, improve continuous integration which is a necessary part of achieving continuous delivery.

4.1.5 Redesign acceptance tests

Testing can be divided into two segments, verification and validation. Simply put, verification is used to make sure that functionality works as specified. Validation is used to make sure that the functionality is actually the right functionality for the current situation.

Current acceptance testing in the ticket solution project is oriented around the user and it is designed to verify that functionality works. Since they are designed to follow a users input, they rely heavily on the graphical user interface which makes automation difficult.

To reduce the amount of manual tests required for a release, acceptance tests should be divided into two types of tests, functional tests to verify functionality and user acceptance tests to validate the functionality is what was actually requested by the customer.

By designing acceptance tests on different levels the manual test load during releases could be reduced. For instance, an acceptance test like "Buy a ticket without a registered card" include the whole process from navigating the GUI and buying a ticket, inputting information for a payment card, and finally making sure that the ticket is received and looks correct.

By dividing the test into different part, each part would be easier to automate. Making it easier to automate more of the test suite will allow for continuous evaluation with acceptance tests during development, leaving only visual verifications as something to validate manually. Automated testing also allows for more variants to be tested. Buying a ticket or two, or even more, all basically have the same human interaction, making sure that the system can handle different amounts of purchases is ideal for automation.

From the evaluation of the acceptance test in 3.5.3 it seems clear that it is not possible to fully automate the whole acceptance test, at least not with a reasonable amount of resources. But by reducing the amount of acceptance tests that have not been run throughout development acceptance testing should be a shorter process with less defects still present.

4.1.6 Standardised commit comment

When performing the analysis for this thesis it was remarked that it was difficult to track development by looking at commit comments. In fact even when knowing when a bug was fixed, it was difficult to find the exact commit that was related to that particular issue. A suggestion for what elements should be present in a standardised commit comment can be seen in table 4.2.

- | |
|--|
| <ol style="list-style-type: none">1. ID that connects to an active task2. What actually changed with the commit3. How the commit was tested, or why it was not |
|--|

Table 4.2: Three points that could be useful in a commit comment

The intentions with standardised commit comments are to create more traceability which could be useful for future analysis and improvements. It could also help developers locate bugs that are found late in development, though it has to be mention that developers have not really expressed having any problems with this task at present time.

Another intention with these commit comments is to reflect over testing, which this thesis has found are one of the bigger obstacles to implement continuous delivery. By asking developers to describe how a commit is tested, or if it is not, why, could help developers improve their testing by finding what they all find is difficult to test or more detailed data to why testing is not performed. The idea of adding the last step, a comment about testing came from a blog post by Steve Berczuk[11].

It is probably rather difficult to get developers to always follow a standardised commit comment protocol without implementing automated checks that make sure the rules are followed. Especially since the perceived value for developers is rather low, but adding a reflection stage regarding testing could be really useful for a team that is starting to learning how to test. It is also a rather cheap cost for "implementing it" and if developers like the feature steps could be taken to automate parts of it, like adding what feature that is currently being worked on. This is something that is explored in a master thesis by Richard Simko[12].

4.1.7 Summary

These solutions all focus on improving development by enforcing quality assurance. The most important change to achieve continuous delivery would be to refocus the project test flow to allow for smaller, faster and automated test cases. This includes breaking down long acceptance tests into more manageable components.

To take advantage of these more frequent quality gauges but also encourage the developers to always keep the software in a stable condition, the customer should be involved more and at the very least be able to give feedback on a biweekly basis. Their part during development should be to decide what should be implemented, and to sign off that what the developers implemented is actually correct.

Test ranking could have some usefulness, but it does not change the fact that there is still a need for more testing during development. Test coverage has some advantages, but most of them are not noticeable until there is a test suite to get coverage from. Finally,

standardised commit comments had one major component for development, get developers to reflect over how the testing can be done. Even just adding how a change has been tested, or why it has not, could be enough. But this is something the developers needs to be committed to do, as there is no real way of making sure that they actually do it.

Chapter 4.1 deals with what changes could be useful to improve development, but speak very little about the costs of continuous delivery and how they affect the particular context this thesis is done in. Chapter 4.2 aims to address these issues.

4.2 Making Continuous Delivery Viable

As mentioned before, the context for this thesis involves a consultant firm which undertakes small and short projects. This has resulted in very little prioritisation on release pipelines since every project basically only one big release eventually followed by bug fixes. With only one major release it is impossible to recover start up costs for continuous delivery.

This means that with the current deployment method, continuous delivery have not been a priority, or even useful. Going forward, the question regarding the usefulness of continuous delivery needs to be asked. Simply put, true continuous delivery where every change gets put through the whole deployment pipeline and ready to be deployed with the push of a button is most likely not an optimal solution. It will simply cost too much to either have a full quality assurance department or spend a large part of development automating every type of testing to support a fully automated deployment pipeline. Looking back to 3.5.3, a simple estimation shows that it could be possible to reduce testing by 30% without GUI testing, or between 50 and 60 % with it. Raising the quality by practising continuous integration however would offer benefits like more stable software, which would encourage more involvement with the customer. Even if full acceptance testing is not done continuously it would be good to evaluate new features continuously and get the customers feedback as early as possible. Practising continuous delivery to the customer and being ready to start acceptance testing when ever the customer wants a new release would be an ideal starting point.

The question remains how this should be done, how should a deployment pipeline be managed to gain the most benefit but with the least cost. A simple deployment pipeline would include an automated build server that runs all available test cases in the project. If they all pass it would be time to perform manual acceptance testing and finally there is a deployment phase. This would also benefit from automation to both reduce risk of human error but also reduce the effort needed to perform a deployment.

The rest of this chapter will discuss three different options and their advantages and disadvantages in order to address **R.C3** Improving development practices are not prioritised.

4.2.1 Every team manage their own Continuous Delivery

This is the way work is organised today. Due to time constraints and developers own interest and competences this has often lead to very little implementation of automation for different stages. This could be changed by setting up goals for the teams.

The positive point is that knowledge is kept in the team, which would make it easier to communicate if problems arise. Also by having every team perform their own configuration management, teams can act as each others redundancies and if one team loses their CM expertise they can receive help from other teams. This is however outweighed by the fact that very little is actually implemented.

But this raises the issue of different teams creating similar solutions which can be considered inefficient and relates to Wayne Babich double maintenance problem.

4.2.2 Dedicated Continuous Delivery team

The second option is to have continuous delivery as a service to all other projects.

Advantages of this would be less double maintenance. By having an organised CM team the risk for losing knowledge is also reduced. The main advantage is that there are dedicated personnel which constantly work to improve the way development is done, it does not become a side project which can be worked on when there is time. If cost was not a factor, this would be the best solution.

The negative side of this is the increase in cost and the difficulty in measuring the long term effects of having such a team. But one thing is clear. Every time a task is automated, and that task is performed, some gain is achieved. For this to be a viable solution there needs to be a need to run these scripts often, for instance there needs to be frequent releases, and someone to receive them. But it also requires a lot of test automation to continuously verify changes for deployment.

It has previously been estimated that acceptance testing takes 80 hours, in contrast, a quick estimation would put a deployment at just a few minutes. This makes the biggest downside with deployment the risk for human error, not the time it takes to perform it.

If there existed a team that dealt with SCM related tasks for continuous delivery there would most likely need to be a team that dealt with acceptance testing as well, as the current biggest bottleneck in development is quality assurance. The solution with dedicated testers have not been explored in depth but is mentioned in section 5.4.1.

4.2.3 Dedicated Continuous Delivery support

One could consider a combination of the two previous alternatives. The biggest advantage from 4.2.2, having a small team dedicated to working on improving development would be kept while reducing the over head cost the previous alternative required.

A smaller team would focus on setting up development pipelines with build, test and deployment automation, providing the infrastructure for the teams to use while leaving the day to day operations in the hands of the teams. Teams would still be responsible for creating the automated test cases and performing acceptance testing and deployment.

It would however introduce the risk of loss of knowledge, but this could be mitigated by documenting the work and more importantly making sure that all teams know how to use the infrastructure.

4.2.4 Summary

So which solution would be the best for the context of this thesis. The benefits from implementing true continuous delivery where every change goes through full acceptance testing would probably not outweigh the added costs of having both SCM and Acceptance testing teams, even if deployment would become cheap, there would still be the issue of manual testing and the difficulty of automating it completely. Partly implementing continuous delivery where deliveries are made to the team (continuous integration) and to the customer but not to production would allow for quicker feedback between developers and customers and hopefully a better product.

To achieve this there needs to be a focus on improving the development process, but also a realisation that it probably is not worth it for just one small project. For continuous delivery to work there needs to be multiple deliveries, perhaps more deliveries than one project can manage. This means that the benefits and cost of continuous delivery needs to be distributed to all projects. So the best solution would most likely be to have dedicated continuous delivery support which will provide a platform for setting up infrastructure, and help reduced duplicated work while not being as expensive as the second alternative.

Chapter 5

Discussion

The first section will discuss the goals of the thesis, and to what degree they were achieved. Then it will go on with lessons learned and what would have changed if the thesis would be done today instead.

The related works section will bring up literature that has provided the main references, similar cases and a master thesis devoted to the practical difficulties of implementing continuous delivery with focus on the tools necessary to create such a pipeline. These books and papers have acted both as inspiration when designing solutions and as reference points for that the conclusions this thesis have reached are similar to that of what other research have found. And as such, are relevant not only for the specific project that this thesis is based on but could contribute to a greater audience.

The chapter will end with a future work section, which will describe where this thesis ends and what remains to be done as well as bring up problem statements not really addressed in this thesis.

5.1 Evaluating the results

To evaluate the results the purpose of the thesis needs to be revisited. The problem statements and research questions in chapter 1 formed the basis for the perceived problems and focus of the initial interviews.

The biggest part of the thesis was the root cause analysis. Figuring out what were the actual bottlenecks and finding evidence of it. The root cause analysis revealed a heavy reliance of undocumented manual testing. Only in the end of a project was testing organised which lead to a combination of agile and waterfall based development process. While plenty of tools can be used to automate the delivery pipeline, the biggest issue with development were the lead times, from a story was designed until it was finished. The greatest obstacle for this was the manual testing that not only made testing slow, but pushed it towards the end of the project due to the cost of reproducing manual testing. It was after the

initial analysis that the focus of the thesis shifted from exploring if traceability between test cases and stories could provide a tool for test selection to instead focus more on highlighting the issues created by manual testing and encourage more organised testing, largely aided by automated tests.

Once the root cause analysis was done several changes were created that could help with reducing *Ad Hoc* testing. As the proposed changes have not been implemented it is difficult to ascertain their effectiveness, some of the solutions are recommended by other research, others by core values in agile development and some are purely theoretical. Especially the second part of the solution, making continuous delivery viable, is based on theoretical reasoning. It is very difficult to quantify the advantages gained, which leaves a relatively simple cost analysis which culminates in more people means more costs. Due to this and the fact that this thesis determine that testing is the biggest issue to achieve continuous delivery, the option where there is a dedicated SCM team is probably overkill. They should work in a support capacity, helping by creating improvements and setting up new delivery pipelines, but daily operations are left for the teams. This means an increase in cost but helps with creating better solutions that can save time in long term while not spending time creating similar solutions from scratch.

5.2 Lessons learned

It is always easier to make a decision with more information, this section is dedicated to reflect on how the thesis was carried out and what would have changed if the thesis was started today.

In the beginning of the thesis there was an opportunity to attend a configuration management conference and discuss the idea for the thesis. Looking back it feels like that opportunity was not utilised as well as it could have. Mainly because it was in the beginning of the thesis and the understanding that the focus would be on moving from manual to automated testing had not yet been realised.

5.2.1 The root cause analysis

First of all, as the thesis focused a lot on quality assurance and the lack of acceptance testing during development, the customers view point would have been interesting. Will they be willing to spend more time to validate functions throughout development? Looking back at possible reasons as to why this question was not raised, only one real issue comes up. As the thesis is criticising the current way of working meant that there was some hesitation from me to bring the customer in on this. At the very least the team could have been asked if they had any objections to me contacting the customer.

The root cause analysis took up the majority of time for the thesis, looking back it begs the question if that was necessary or not. Gathering information, whether it is facts or interviews take time, as a new person in the office the social aspects become very important. Initiating contact is something that became easier and easier throughout the process.

A lot of time was also spent analysing the results. Performing this analysis was quite a bit different from previous work done during the education, and one reason for taking the

time it did is most likely due to inexperience. If it was done today it would hopefully be done quicker.

During the analysis the problem of waterfall testing was realised. To further investigate this, interviews to see if other parts of development might work according to the waterfall methodology. Reviews of bids for contracts were also carried out. While this did yield some discoveries, like the customers lack of involvement during development, the time spent was probably too much.

This thesis was carried out by a single student and even while mentors both at the university and the company were available for the most part, as well as employees on site were friendly, it was difficult to quickly discuss a thought or idea. If the thesis would have been carried out with a partner, having those short and quick discussions would have been a lot easier.

5.2.2 Designing solutions

Just creating theoretical solutions are not as interesting as providing actual data regarding their success or not. As such, some implementation would have been beneficial. Especially refocus project test flow could have been introduced by having developers create acceptance tests before development started. It would lay the ground work for including the customer more in the development process. On the other hand, if implementation was the goal, implement would have needed to start a lot earlier to allow time to measure the effects. Combining this with the analysis would be very difficult given the fixed time frame of the thesis.

5.3 Related Work

This section will detail work that has had some influence or relation to this thesis.

The biggest reference for the concept of continuous delivery is the book "Continuous Delivery: Reliable software releases through build, test and deployment automation"[3] by Jez Humble and David Farley and have been a huge inspiration.

Due to the focus on quality assurance and how that should be performed the book "Agile Testing: A practical guide for testers and agile teams"[5] by Lisa Crispin and Janet Gregory was referenced.

It becomes very relevant as continuous delivery and agile development share two fundamental concepts, continuous integration and constantly evaluating the quality of software instead of having large test phases in the end of the project.

Additionally from these books, research papers have provided helpful insights during the thesis. Three of them will be presented here.

5.3.1 Continuous Delivery: Reliable software releases through build, test and deployment automation[3]

This book explains the whys and hows of continuous delivery. Why it is advantageous to be able to quickly release software and how it can be done. It starts out by identifying problem areas that hinder the adoption of continuous delivery and then moves on to how these issues can be fixed.

The main problem that continuous delivery aims to fix is reducing lead times. This is relevant from a commercial stand point in that it allows a company to quickly respond to customers needs and decrease the time to market for a product. This is also beneficial for developers. Whether it is by providing feedback from quality assurance or even from customers directly, receiving feedback as fast as possible is the main goal. Problems with development should be solved as fast as possible, before they have had time to affect other components.

For the solution part there is a lot of focus on automating processes, everything from builds and tests to creating a delivery pipeline where software is moved through stages automatically and finally released.

The main goal with the book is to emphasise the fact that reducing lead times are beneficial, and automation is a tool that is invaluable when speeding up frequent and menial tasks.

An underlying tone for all this is consistency. To always know the state of everything in the project is what allows this rapid pace that traditional development does not. This is where software configuration management comes in. Version control, making sure everyone is using the right, usually the same, libraries are key components for this.

The findings in the book are supported by experiences of seasoned developers knowledge. This has made it difficult to challenge a lot of the claims made as a student without the experience that comes from working with software development for years. Many of the claims however appear to be reasonable and the findings in this thesis seem to confirm the problem areas stated in the book. Regarding the solutions it becomes a bit more difficult to evaluate. With little personal experience and without solid facts to back up the claims in the book the results are left to be taken at face value.

As the goal with this thesis have been to analyse the possibility of adopting continuous delivery in small scale projects this book has been a huge inspiration. Many of the ideas are based around the fact that automation brings stability to processes and the ability to repeat tasks endlessly.

5.3.2 Agile Testing: A practical guide for testers and agile teams[5]

As explained by the title, this is a book about software testing in an agile environment. The main focus of agile testing is bringing it closer to development. To make sure that testing does not become some future step that is done before the product can be released. Agile testing is done iteratively as often as possible to test small changes of the code.

This means that testing needs to be streamlined. It is not possible to test every change the same way as a end of project test cycle is executed.

Agile testing is not just about using a specific type of test cases or tools, it is a fundamental change in process for testers and, as with many agile teams, developers who often carry at least a semi role as testers as well.

The two main problems with implementing agile testing are the following. Getting developer to change the way they develop, and how they use testing in this process. The second problem is to reach a point where the software can be tested frequently and quickly with the purpose of giving developer quick feedback on their changes.

Changing to agile testing takes a leap of faith, it is therefor important to have courage and see these changes through. They do mean extra work in the beginning, especially before the advantages are seen.

To address the problem of being able to test frequently automation becomes an important part. Especially as a first check in the process of verifying the quality of software. The more things that can be tested automatically, before the process requires manual testing, the better.

Testing in the ticket solution project was identified as one of the major bottlenecks for continuous delivery, more specifically the use of manual exploratory testing. This conflicted with agile testing because it failed to be easily reproducible and resulted in a long testing period towards the end of the project.

The solution suggestions in this thesis are inspired by the concepts mentioned in this book. They are intended to shift focus from slow to quick feedback by adopting automated testing as a first quality assurance gate, and to involve the customer to provide more feedback early on.

5.3.3 Container-based Continuous Delivery for Clusters[13]

While not directly influencing this thesis, another master thesis carried out simultaneously in the field of continuous delivery is "Container-based Continuous Delivery for Clusters" by Per-Gustaf Stenberg. His thesis evaluates the use of containers as a means to implement a continuous delivery pipeline. It provides a complimenting analysis on what is necessary from an infrastructural viewpoint as compared to this thesis which has focused on changing the way developers work to benefit from continuous delivery. Looking at part of the conclusion of Stenbergs thesis.

"It is important to note that continuous delivery is not just about implementing a new workflow and system for delivery. It has more to do with a cultural change. The introduction of continuous delivery will not magically make a development team more agile. The agile mindset needs to be in place beforehand in order to get the full potential of continuous delivery."

This raises the same point as this thesis, continuous delivery needs an agile mindset from the developers and the first step to reach that is to have the developers embrace the concept of iterative small releases instead of one big towards the end of a project. Once that is in place creating an infrastructure that helps with that is invaluable.

5.3.4 Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy)[14]

To summarise why continuous integration would be a great tool in development one need only to listen to this quote, "That afternoon we received an email from a customer saying that they had noticed the defect fix and wanted to say a huge thank you for resolving a pain point in the application. The new hire came over to me and with a big smile said "this continuous delivery is hot stuff!". This article provides an encouraging reading while still pointing out the issues that came up when Rally Software moved to continuous delivery. As a recurring theme within continuous anything, automation plays a huge part. In this paper the focus lies on emphasising trusting the automation. It is an important step when adopting agile development, to learn to trust the computer to do what it is told and to have the confidence that when something goes wrong, due to the automated delivery pipeline will allow for a quick fix. This is directly related to this thesis, as it was discovered during the interviews that developers did not fully trust automated testing, and felt that they still needed to check it themselves. This paper helps proving that it is possible to trust automated testing, and even if it is not possible to completely automate everything, a lot of it can be.

The impact on support structures is also brought up. To be able to deliver a product continuously, not only the developers need to be included in the work progress, in this case, sales and marketing had to adapt their processes as well.

This article differs from others that have been reviewed in that it brings up the importance of embracing change. Adopting continuous delivery means changes for everyone, developers, marketing, sales and quality assurance, but the changes are good.

The paper claims that it is expensive to implement automated acceptance testing, but that the results are beneficial, little proof is provided however. One impression from the paper is that it is based on a product with a long life span, this is a big contrast to a short lived project where heavily investing in acceptance testing may not pay off. Finding the right balance of automation based on the projects life span is most certainly a challenge.

5.3.5 Enabling Agile Testing Through Continuous Integration[15]

This paper is presented as a case study for implementing continuous integration. It explains the connection between continuous integration and agile development and how the quick feedback helps developers.

Underlying research is presented as a check-list of what is needed to implement continuous integration and agile testing. The paper goes on to list the ten practises of continuous integration as defined by Martin Fowler and discussing if the team were fulfilling them before and after their continuous integration was implemented.

The relation between the paper and this thesis is the focus on automated testing. This is an integral part both for agile development and continuous integration. Both unit testing and acceptance testing is brought up, and as with this thesis, the difficulties of automating acceptance testing are the concern. The paper is however on a more practical scale, when this thesis has been on a theoretical level and focused on why there have been issues with

implementing automated testing.

The paper concludes with going through the check-list and discusses what goals were achieved, and why, if they were not. The conclusion is in a positive tone and describes the overall success. While the check-list is mostly fulfilled the paper did not address if the changes gave any benefits to development process.

The greatest value of the paper is the fact that it provides a real world example of a generally successful attempt to implement continuous integration and the fact that the results were positive. This gives motivation to others to try implementing similar solutions.

5.4 Future Work

This chapter will address what was not encompassed by this thesis. What remains to be done but also what other research questions that have been raised during this thesis and could provide a basis for other thesis's. As a separate section, dedicated testers is introduced. This is something that is often used in large organisations, and much like dedicated SCM teams is something that increases overall costs.

There was not enough time to implement any of the solutions, thus that is a good starting point, to actually apply the solutions. To begin with, refocusing the project test flow by building up an automated test suite according to the testing triangle in figure 3.1, to encourage continuous integration. Redesigning acceptance tests to allow for more frequent releases and actually starting to release more often.

Simultaneously creating a delivery pipeline could be beneficial to relieve the developers from repetitive tasks and further encourage frequent deliveries, both to the team, the customer and ultimately to production.

From a more academic stand point, the original research question "Could a story focused test selection method be useful during development?" is left largely unanswered. The reason for this is that it was discovered that testing during development was mostly undocumented. This result in no data to analyse to discover if bugs discovered during development were more often then not related to stories the current one had a dependency towards. This could provide grounds for a thesis but, as found during this thesis, needs a more documented testing approach.

5.4.1 Dedicated testers

Much like configuration management can be outsourced from the project and applied as a service to projects, quality assurance can be handled the same way.

Dedicated testers could have multiple functions, performing manual regression and acceptance testing, but also provide testing frameworks and write automated test cases while developers focus on development and basic testing like unit testing.

Finally, if all they do is performing manual testing, should this be handled in-house or could it be outsourced to external companies?

5.5 Difficulties with Mobile Testing

Automating testing have been a big part of this thesis. Especially the difficulties of automating GUI testing on mobile applications have been a big reason for why there have been very little automation and why the focus of this thesis have been to move testing away from the GUI and focus more on unit and integration.

While automating GUI testing is difficult in their own sense, there a few factors that make it even more difficult. Accessing system features, or rather changing them, like the system clock or flight mode is prohibited for applications. While the reasoning behind it is unclear, a reasonable guess would be security reasons. This raises the question if this is something that should, and could be unlocked for developer modes. Running test code with predefined settings on the phone, like flight mode, or actively change the clock during test execution, could provide a lot of aid in helping with test automation.

Chapter 6

Conclusion

Continuous delivery really shines when used in a project with a live product. When developing a product from scratch it becomes a bit more complicated. But even if the product is not ready to be released to market, releases for the customer can be used to evaluate single features to get early feedback before it becomes a base for future features.

To improve the continuous delivery process, or even continuous integration, the quality assurance focus needs to change. Relying on manual testing keeps regression testing to the minimal, and more importantly it makes testing slow and unreliable. This puts the quality level of the software in question after every commit. It could also be the answers to the initial problem statement of not knowing when testing is done and the research questions in the introduction:

R.Q1 How could increased traceability help highlight if a story is tested sufficiently?

R.Q3 How do we link stories and tasks with test cases?

By creating test cases during development and not as an after thought, stories will have a clearer definition of when they are done while providing a link between test cases and stories.

R.Q2 Could a story focused test selection method be useful during development, was left largely unanswered due to insufficient data to evaluate. The question remains and could be grounds for another thesis.

To achieve continuous integration, testing needs to shift from manual testing towards a solid foundation of cheap unit tests and integration tests. Thus providing a platform for early feedback for developers during development. Automated GUI acceptance tests are a great tool, but with slow execution times and known to be unstable, should not provide the basis for quality assurance.

The second problem statement of finding bugs late in the development cycle should also be mitigated by more formal testing early on. With acceptance test created on a feature

basis, functionality can be verified, but also the validity of the function and the test case by involving the customer.

Finally, to answer the last two research questions:

R.Q4 What CM support is needed for this to be a feasible solution?

R.Q5 What level of automation is needed for developers to embrace this functionality?

More focus needs to be put on saving and distributing knowledge between different projects. Setting up a continuous delivery pipeline is not beneficial when applied to only one project. They need to be reused for future projects and shared with other in-house teams.

The needs for automation should be focused around testing. For developers to embrace automated testing the commit process should run test cases automatically. The need for automating the release process increases as continuous integration is applied.

Bibliography

- [1] Continuous integration. http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf. Accessed: 2015-11-24.
- [2] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley, 1986.
- [3] David Farley Jez Humble. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.
- [4] Ward Cunningham Tatiana Apandi Diaz. *Extreme programming : pocket guide*. O'Reilly, 2003.
- [5] Janet Gregory Lisa Crispin. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley, 2009.
- [6] Kent Beck. *Test-Driven Development by Example*. Addison-Wesley, 2003.
- [7] Rudolf K. Keller Stefan Berner, Roland Weber. Enhancing software testing by judicious use of code coverage information. In *Software Engineering, 2007. ICSE 2007. 29th International Conference, 2007*.
- [8] Margaret Burnett Gregg Roethermel Joseph Lawrence, Steven Clarke. How well do professional developers test with code coverage visualizations? an empirical study. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium, 2005*.
- [9] Williams L. Maximilien E.M. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference, 2003*.
- [10] Hossein Saiedian David Janzen. Does test-driven development really improve software design quality? *Software, IEEE*, 25(2), 2008.
- [11] Motivation visibility, and unit testing. <http://blog.berczuk.com/2010/05/motivation-visibility-and-unit-testing.html>. Accessed: 2015-11-05.

- [12] Richard Simko. Automating traceability in agile software development. Master's thesis, Faculty of Engineering LTH at Lund University, 2015.
- [13] Per-Gustaf Stenberg. Container-based continuous delivery for clusters. Master's thesis, Faculty of Engineering LTH at Lund University, 2015.
- [14] Steve Stolt Steve Neely. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Conference (AGILE), 2013*, 2013.
- [15] Sean Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE '09.*, 2009.

Appendix A

Interview Questions

Interviews were performed in swedish and this is a translated representation of the questions.

1. What is the state of the delivery process in your current project?
 - (a) What is the current strategy for testing?
2. How are acceptance testing performed?
 - (a) How did you arrive at this solution?
3. How are stories and tasks linked?
 - (a) Do you perceive any advantage with this?
4. When a test case is written, how do you show what it is testing?
 - (a) Is it easy to find out what other developers test cases test?
5. How do you select a test suite for regression test when a story is completed?
 - (a) Are all test cases run?
 - (b) What is the verification process for a story?
 - (c) are tasks tested individually or are they tested when the full story is done?
6. Why is the testing not automated? (added after the first few interviews)
 - (a) Is the CM structure necessary for it missing?
 - (b) Are the acceptance test cases too difficult to automate?
 - (c) Is it the maintenance cost that becomes too high?

7. Have you been in projects with very long testing cycles?
8. Are there any changes you would like to see?

Appendix B

Survey questions

Questions were created and distributed using Google Forms and sent out to developers, coaches and scrum masters at the company. Answers varied from multiple choice to graded scales. This is a translated representation of the questions.

1. Do you work in the office or at a client location?
 2. What is your normal role in the team?
 3. What tools are used in your current project?
 4. On a scale of 1 to 4, how good is your knowledge of the tools you use?
 5. On a scale of 1 to 4, how capable do you feel you are to set up a source code repository?
 6. On a scale of 1 to 4, how capable do you feel you are to set up an automated build server that for instance builds a project and or run automated test cases?
 7. On a scale of 1 to 4, how high faith do you have that you can write good unit tests?
 8. On a scale of 1 to 4, how high faith do you have that you can write and maintain automated GUI tests?
 9. On a scale of 1 to 4, how much do you trust that automated testing give enough feedback?
 10. Which of the following task do you not want to perform in your daily routine?
 - (a) Deal with SCM related issues (Git, jenkins, jira, etc)
 - (b) Write unit tests
 - (c) write integration tests
-

- (d) write GUI tests
- (e) perform manual testing
- (f) something other

Showstoppers for Continuous Delivery in Small Scale Projects

Jakob Svemar
Faculty of Engineering, Lund University

Abstract—Small scale software development projects provide their own difficulties.

This thesis is looking at the possibility of introducing continuous delivery in such an environment.

The solutions are focused on creating an iterative process and including the customer in the early stages.

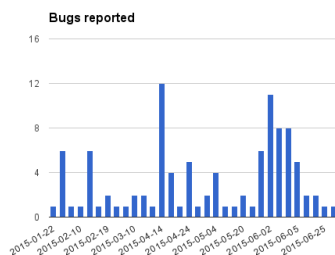
The way software is developed can vary greatly, and different settings create different conditions. A consultant firm can provide a range of services. One of which is in-house development. Their largest team consisting of nine developers, a scrum master and a project manager, is working on ticket solutions for public transportation. This spans several different customers, generally the public sector, and it has been the main focus during this thesis. This has created a situation where projects run in very short time frames with limited budgets. This means projects are set up in a traditional manner and shares a lot of similarities with the waterfall model.

From initial discussions in the beginning of this thesis, two perceived problems were defined. Together with my desire to explore the benefits and possibilities of implementing continuous delivery formed the basis for this master thesis. The problem statements were:

- How do we know when we are done testing?
- We find bugs too late in the development cycle.

Finding the point where the software is stable enough becomes difficult when relying on manual exploratory testing. It takes a lot of time and is prone to human error. Another side effect is that regression testing is reduced to a bare minimum. A previously implemented feature may remain untested through several iterations of software.

If the majority of testing is done at the end of a project, the majority of bugs will be found at the end of the project as seen in the figure.



Bugs reported for the analysed project.

The purpose of this thesis has been to look at the current development practice, identify bottlenecks and figure out changes that could help remove them.

The scope is limited to the in-house development and specifically the team creating ticket solutions. Target audience is developers in similar situation where small projects are common and automated testing is scarce.

Continuous delivery really shines when used in a project with a live product. When developing a product from scratch it becomes a bit more complicated. Still, releases to the customer can be used to evaluate single features to get early feedback before it becomes a base for future features.

To improve the continuous delivery process, or even continuous integration, quality assurance focus needs to change. Relying on manual testing keeps regression testing to the minimal, it also make testing slow and unreliable.

By creating acceptance tests during development, stories will have a clearer definition of when they are done while providing a link between test cases and stories and allow the customer to validate features early on.

Setting up a continuous delivery pipeline is not beneficial enough when applied to only one small project. They need to be reused for future projects and shared with other in-house teams.

The needs for automation should be focused around testing. As the process matures and releases are made more frequent an automated release process will be a logical step.

Author: Jakob Svemar

Contact information: jakob.svemar@gmail.com