2012 45th Hawaii International Conference on System Sciences

Automated Acceptance Testing as an Agile Requirements Engineering Practice

Børge Haugset
NTNU
borge.haugset@sintef.no

Tor Stålhane
NTNU
tor.stalhane@idi.ntnu.no

Abstract

This article describes how the use of automated acceptance test-driven development (ATDD) impacts requirements engineering in software development. We extend an existing framework of inherent risks in RE by adding knowledge from literature and a case study. We show how ATDD can be seen as a mix of the traditional RE focus on documentation and the agile focus on iterative communication. ATDD can mitigate some of the inherent risks in RE. It also carries with it the need for a proper domain and a very rigorous development method that requires disciplined developers and dedicated customers, preferably on-site.

1. Introduction

The creation of high quality software is about making software that meets the customers' both explicit and implicit needs (ISO 9001). This requires that the customers and developers reach a common understanding of what to make as well as understanding the possibilities of what can be made. There are many methods on how to reach this understanding. During the last decade *acceptance test-driven development* (ATDD) has gained a lot of attention as a development method within the agile community. It has been proposed as a way to provide a common understanding of the needs of a system and to repeatedly and automatically test software at the business level. The first and most used tool made for this is Fit, which in [1] was proposed as a way to improve the communication between customers and developers. Fit and FitNesse (based on Fit) is described by Ricca et al. as *"Fit (Framework for Integrated Test) is an open source framework used to express acceptance test cases and a methodological tool for improving the communication between analysts and developers. Fit lets analysts write acceptance tests in the form of simple tables (named Fit tables) using HTML or even spreadsheets. A well-known implementation of Fit is FitNesse, substantially a Wiki where analysts/customers can upload requirements and related Fit tables. A Fit table specifies the inputs and expected outputs for the test."* [2].

Some studies have focused on ATDD since its introduction nearly a decade ago. Student experiments have confirmed that Fit tables enhances communication of requirements [ibid.] The main author of this paper has studied a set of projects with successful use of ATDD [3]. There has, however, been no research on describing how this method impacts requirements engineering in a project from a theoretical perspective. We believe this would be an important study, placing ATDD beside the other methods in a developers' arsenal. We want to contribute to this, and ask the following research question:

Where can you place ATDD as a requirements engineering practice?

To answer this question it is necessary to look at how ATDD differs from other RE approaches. We have in this work chosen to focus on RE and agile RE. There has been a lot of work in these fields already, providing us with some shoulders to stand on. Ramesh et al. has identified a set of risks inherent in RE, and compare this to agile RE [4], [5]. It felt natural to extend this framework to encompass ATDD. Our knowledge of ATDD is based on previous literature, plus our own research. Special attention is paid to the previously reported case of a set of developers who felt they had cracked the code of ATDD [3]. We show how ATDD can be seen as a mixture of the documentation-centric RE and the communication-focused iterative agile RE, carrying its own set of benefits and challenges. The benefits stem both from the inherent strengths that lie in discussing requirements using the acceptance tests as a mediator and its abilities for automation of the same tests.

The article is composed by the following sections: In chapter 2 the research method and following limitations are described. Chapter 3 contains a literature walk-through of RE, agile RE and ATDD. Chapter 4 describes the results from a case study, while Chapter 5 discusses the previous findings. The conclusion can be found in Chapter 6.

978-0-7695-4525-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2012.127

2. Method

This article is based on four in-depth interviews with developers from a consultancy company (Konsult) in Norway. An analysis of their use of ATDD has already been published [3], for a richer description of the case we refer to this article. The previous article focused on describing the case in detail, and understand the developers intentions and use of ATDD. We have also performed a literature study and another case study, aiming at describing both current knowledge within the research community and actual use of ATDD [6], [7]. Now we approach the data from these three studies from another angle, and describe the requirements engineering aspects of the development method. The last case (Konsult) was chosen because they seemed to have a matured understanding and use of ATDD. We have not performed any further interviews or observations previous to this article. Here we wish to place our previous findings in a theoretical framework.

Limitations of the study

The study has some limitations, and an obvious question regards its generalizability. The amount of interviews in the case study is low. We feel that there are important insights to be made from even a low amount of interviews. Our interviews were long (55-90 minutes), and gave us a deep understanding of how they worked. We also draw on our previous research on the topic when reaching our conclusions. The amount of studies on ATDD is low, and we have found no previous research linking ATDD to an RE framework. Our findings and conclusions here will provide the first such study, something we believe will benefit both other researchers dealing with the same topic and developers that consider using ATTD. We have chosen to include a lot of citations from our interviews in order to give the reader a rich understanding of the case, and thereby be able to draw their own conclusions.

3. Literature

In this section we will describe the current status on research. We describe aspects of RE, agile RE, existing comparisons of the two, and ATDD.

3.1 Requirements Engineering

Requirements engineering (RE) is a large domain. We will focus on the methods and techniques used for requirements elicitation. In addition, we will focus on functional requirements and leave non-functional requirements (NFRs) out of scope. Interested readers should see P. Saripalli's study of this area [8].

All RE methods have activities related to requirements elicitation, analysis, specification and validation. What varies is the way these activities are organized and documented and how much of the available resources each of them get.

Whatever the method used, the RE process will always start with identifying the stakeholders – no stakeholder, no requirements. The stakeholder may be a real stakeholder – a customer – or a proxy, such as a product manager or the sales department. Once this is done, there are three ways to proceed. We start with the stakeholders':

- Goal – what do they want to achieve?
- Main activities – what do they do when they "do their job"?
- Main goals – what do they want to achieve when they "do their job"?

Some authors – e.g. [9] – claim that starting by identifying the high level goals is crucial. Ideally, the goal approach as defined by e.g. the KAOS method, using the goal patterns Achieve, Cease, Maintain, Avoid and Optimize is a documented, efficient and stringent method – see for instance [10]. Although not explicitly stated, the goal-oriented methods requires a close cooperation between stakeholders, domain experts, systems analysts and software developers.

Experience, however, shows that in many cases, the stakeholders, and especially users, start by identifying the functions that they need, either directly or through one or more scenarios. Each function is an activity – e.g. "I need to be able to print the ledger of each account". This is for instance the case when using UML with its use case diagrams – high level – and textual use cases – low level [10]. The use cases can also be constructed using observations and social analysis. There is a large variation in how functional requirements are collected. At one end of the scale we have the case where the customer delivers a set of functions that shall be implemented to the situation where the developers and the customers get together using a lightweight method to discuss and agree upon requirements – see for instance [11] or the requirements engineer runs a set of interviews with the stakeholders [12].

The third alternative – stating the main goals – is what is done when writing user stories, e.g.: "As a <role> I want to <perform some tasks> so that I can <reach some goal>" – see for instance [11]. The user stories are usually written by the requirements engineer and one or more stakeholders. The requirements documented as user stories are just goals. The detailed requirements – how this will be realized using functions – is done later in the process.

There exists also a set requirement elicitation methods called collaborative requirements engineering with a stronger user involvement. Ang et al. present a good overview of the existing methods in this domain and introduce a new method that they have developed [13].

3.2 Agile RE

When considering agile RE, we need to remember two important facts [14], namely that agile methods are: *adaptive* rather than predictive and *people oriented* rather than process oriented. In addition, agile development highlights the difference between formal models, which are part of the system's documentation and informal models, whose main purpose is to support face-to-face communication (customer involvement). It is important to be aware that different agile development methods have different RE processes. There are large and important differences between XP and some of the methods from the Crystal family.

The main concept in agile RE is not the technique used but the importance of customer involvement. Several non-agile RE methods also focus on customer involvement but agile development is unique in the way it focuses on customer involvement throughout the whole development process. Thus, any method or techniques is suitable as long as it helps to provide customer – developer communication. An observation from a project manager in [15] is revealing here: *"We value conversation more than the actual tool itself [Fit]"*.

The most commonly used technique for documenting requirements in agile development is user stories although both use case diagrams and textual use cases are also quite popular. Both use stories and use cases are kept as high-level descriptions until implementation, where they are described in sufficient details for implementations. In this way, all experience from previous activities in the project can be used when the final, implementation-related decisions are made. Models in agile RE are mostly used to enhance communication, and they are thus mostly throw-away models [14]. If they are kept at all, they are often kept as snap-shots of white board drawings.

Agile RE is mostly weak when it comes to non-functional requirements (NFR). One of the reasons for this is that the customers are also mostly weak in defining NFRs irrespective of whether they use an agile method or not. An exception noted in [4] is user friendliness, where they observe that most changes done in the agile development based on customer feedback, is related to GUI. There are suggested ways to circumvent the lack of focus on NFR [16].

Some statements made by people working in an agile environment or doing research in that area indicate that agile development is mostly or exclusively used for simple problems and domains – e.g. *"…the customer sits down with the development team to provide detailed information on a set of features…"* [4]. Based on this we can be pretty sure that they are not going to control an Airbus. Thus, instead of discussing all possible shortcomings of agile RE, focus should lie on identifying the domains where agile RE can be used (e.g. simple logistic systems) and where something else should be used (e.g. a process controller that is compliant with IEC 61508). If one alternatively tries to enhance agile RE with all kinds of add-ons to meet all kinds of problems – real or theoretical – one may throw away the good things with agile RE where it is already functions well in order to also apply it to areas where it is not really needed. Focus should be on continuous improvement, collaboration, and delaying decisions until the last responsible moment. Neither do we here intend to enter the discussion of which kinds of projects agile development is suited for.

3.3 Existing comparisons of RE and agile RE

Ramesh et al. provide a framework for how the agile approaches mitigate or exacerbate the risks found in 'waterfall RE' [4]. As we intend to use this framework directly in our discussion we go through them in detail:

Lack of requirements existence and stability: Agile mitigates this risk, and this is driven by the focus on face-to-face communication, iterative RE and constant planning.

Issues with users' ability and concurrent among users: Agile RE has a mixed impact on this issue, and is driven by access to the right customer representatives. This can be problematic when the knowledge that the developers need is fragmented across several customer segments, each of them having different priorities. The short development cycles in agile exacerbates this problem. At the same time agile developers communicate more with customers. Characterized as an intractable risk, meaning it is difficult to cope with.

Inadequate user-developer interaction: This risk has mixed impact from agile RE. The improved communication helps, but the customer needs to trust the agile process. There's a concern that developers end up only talking to dedicated on-site customer representatives, thereby not seeing the whole picture.

This can be helped by a development strategy, leading to a tractable risk.

Overlooking crucial requirements: This risk is mitigated by agile RE as it leads to a constant focus, both from developers and customers, at identifying the most important requirements throughout the entire development lifespan.

Modelling only functional requirements: This risk is exacerbated by agile RE, as the method tends to focus on functional requirements. Issues like scalability and security are overlooked in the early phases and has no detailed design. This makes it more difficult to implement it later in the process. Customers also fail to recognize the importance of non-functional requirements. Ramesh et al. define this as an intractable risk, but this is mostly a developer experience and discipline issue, and suggestions on how to handle this can be found in [16].

Not inspecting requirements: Exacerbated by agile RE, as the lack of documentation makes it difficult to verify the system by inspections or walk-throughs. The tractability of this risk depends on the complexity of the system and the amount of attention that is paid to design. Paetsch et al. note how the lack of documentation might lead to long-term problems such as sharing knowledge [14]. On the other hand they also describe how this problem is lessened by the agile focus on documenting only the core aspects of the software, thereby increasing the chance of keeping it up-to-date.

Presenting requirements in the form of designs: Agile RE has minimal focus on documentation, exacerbating this risk. Little documentation makes for quick implementation, but if the same requirements need to be changed the lack of requirements documentation could impede the evolution of the software. However, agile development is all about embracing change, and the developers Ramesh et al. talked to did not see this as a serious issue. Therefore they deem this a tractable risk.

Attempting to perfect requirements before beginning construction: This risk is mitigated by agile RE as it has a focus on perfecting requirements through iterative RE, and is tractable.

Intrinsic schedule flaws: Agile RE has a mixed impact on this risk. The method is unable to address cost and schedule estimation. Requirements emerge throughout the development period, leading to updating the schedules and cost estimations. This leads to a more precise schedule over time. Ramesh et al. define this as a tractable risk.

Agile RE is, to quote Ramesh et al., neither panacea nor poison to the intrinsic risks to RE. It proves helpful in some ways, and harmful in others. Which development style that is most useful varies according to the development setting.

3.4 Acceptance test-driven development

The first author and a colleague has described the status of research on automated acceptance tests (AAT) through a literature review, as well as providing insights from two case studies [7], [6], [3]. We summarize some findings here, and extend on this with further detail in the discussion. The findings most closely related to requirements suggest that it is relatively easy to learn Fit [17], and students preferred this to writing requirements in prose [18]. While one of the intentions of Fit was to make the customer engaged in writing the acceptance tests, this was reported to be problematic. The business analysts in a real project were not interested in describing more than plain text [19]. Customers have in projects brought other kinds of documentation to meetings, such as diagrams, mock screen shots or embedded commentaries [20], [15]. This suggests something more than the mere Fit tests may be needed to convey understanding of a requirement. In the case of FitNesse, which is a wiki that contains the requirements (mostly as Fit tables), there is the possibility to also include other content such as a textual description of the requirement, and links to other artifacts, so the tools support this need. The process of writing requirements as Fit tests has also been described as difficult because it requires discipline, and this can lead to skipping them [20]. The process of simply writing the Fit documents could, however, have a positive effect as it led customer and team to discovering inconsistencies [ibid.]. Later student experiments have verified that the process of creating Fit tables (developers and customer sitting together discussing the requirements using Fit tables as the mediator) improves the understanding of requirements [21] over using prose, and they have also been connected to reducing noise, over-specification and ambiguity [17]. Ricca et al. however conclude that you should refrain from using Fit if the tables would become too complex (Web Fit tables are mentioned as an example) [2].

4. Results from the case study

For a detailed description of the case discussed below we refer to [3], and we will only briefly describe the context here. We interviewed four developers from a consultancy company (Konsult). The project group, now a total of ten developers and two interaction designers are working on-site at a customer office (Kunde). Kunde had little experience in agile (and ATDD in particular) prior to the developers Konsult.

All projects are run in a test-driven fashion, describing unit tests in a BDD (Behavior-Driven Development, for a description see http://behaviour-driven.org) format using Cucumber (see http://cukes.info/). The project with the most extensive use of FitNesse currently has 1100 lines of tests, the average test being 15-20 lines long.

Business requirements are discussed in bi-weekly meetings with the customer. The requirements that are considered most important are selected for the next iteration, and then discussed using a white-board. This discussion ends with an agreement in a table format, and the developers create FitNesse tests based on this agreement. While the developers start coding according to these tests, they also send the tests as Excel sheets to the customer for filling in data. The good thing about FitNesse, one subject explained, is that *"In FitNesse you have data separated from code which means that you actually can show this to a business person, unlike Java code which you just can't show"*.

The communication of requirements has been focused around the FitNesse tests, instead of talking loosely about the domain: *"This forces both us and them to think in the same way, we establish a good common understanding by sitting down talking about the tests."* The customer was not actively writing Fit tests, this was mostly the developers' job since it would require too much technical knowledge. That did not mean customers had not heard of it: "*They are really very conscious about it, as we have worked hard to make them so. We bring printouts, showing it do them, and such, but for them FitNesse is what we use, their link to the FitNesse tests are the Excel sheets. They are very well aware of what the Excel sheets are used for."*

Not all requirements ended up as Fit tables. The decision was partly based on reducing uncertainty: *"If there is no room for misunderstanding, we don't bother [defining FitNesse tests]. [However,] If we are not one hundred percent certain we understand the problem, we do it in FitNesse."* The most simple requirements, such as shifting data from A to B or just presenting something from a database was not described in FitNesse. Neither was it used if it was difficult to describe the requirement in the tabular fashion that is used in FitNesse. If possible, FitNesse proved to be a good tool for complex and difficult requirements: *"...But FitNesse is gold because you have really documented what business probably would have done non-documented. They would have sat there, punched a little back and forth...to verify something. Or they would probably not even have tested that much."* The developers all explained how they felt that the domain was very well suited, dealing with logistics. One project was so well suited that they ended up with full coverage of FitNesse tests, and could deploy the system with almost no manual testing.

The process of programming often resulted in having to discuss the requirements with a customer several times, even throughout the iteration. The proximity to the customers was important: *"It is a lot of back and forth, and if I had to write an email and wait a couple of days for a reply or take a taxi to the customer, it would never work. It's clearly a success factor."* Requirements also varied according to which customer group the developers talked to (managers, IT, documentation): *"But it makes it hard for us to stick with like: This is the requirement, we have done this. We always have to think about switching requirements and that's why automated parts of test are very good. Because what the test says is actually what we've done. And if the next meeting refines the requirements from the previous meeting, then the tests must [change]."* This developer thought that making changes visible was the main benefit of using automated acceptance testing.

At the end of every iteration a demonstration was held. Here the customer could provide immediate feedback with respect to their requirements. In addition, these meetings opened for more detailed feedback. One of the developers explained how the testing was done: *"...what we regard as tests, as good programmers, are things that can be automated, and it is difficult to test a goal or a vision automated, so then it's more about agile, continuous feedback from the customer, continuously showing what you make etc... they [the customer] at all times can correct the goals. So in that regard I'd say that the demo and everything we have of customer meetings, follow- up we do, helps to direct us towards the goal..."* Another, manual test was performing activities that would be too difficult to automate, such as checking cross-browser compatibility. In the freight guide project this particular task could take as little as one hour per iteration, and the customer had never performed any manual testing.

Our main insight from this study was that development was done using ATDD two phases that took place every iteration of the project: In the *specification phase* ATDD was used to help communicate requirements (externalization). This process had nothing to do with automation but with uncovering, understanding and describing requirements. In the *verification phase* ATDD was used to automatically verify that the code adhered to the customer requirement on an acceptance level, accompanied by manual testing for certain issues.

5. Discussion

In our discussion we will go back to Ramesh et al.'s characterization of risks in RE, and describe how ATDD impacts these. We will look for both mitigating and exacerbating factors.

5.1. Lack of requirements existence and stability

Agile RE mitgated this risk. ATDD has the same iterative interactivity, and should give the same benefits. At the same time ATDD has a focus on the customer and developer agreeing on the existing requirements through the use of highly specified written tests, and as such ATDD would reduce the uncertainty and increase the stability. In a project setting with shifting or unknown requirements ATDD has both benefits and challenges. By using acceptance tests to mediate requirements (specification) you can gain a better understanding of them, and make more correct choices. Maurer et al. confirm the communicational importance of automated acceptance tests for mediating domain knowledge to the developers, and how it gives feedback that the understanding was correct [22].

The use of ATDD has a definite cost in that it requires a lot of customer and developer effort, but the result can be used directly in aiding development of code (verification). If the requirements change the tests must reflect this, adding a larger cost than regular agile development. In a development project with full coverage of automated acceptance tests you would not have any requirement developed without first describing it as an automated acceptance test, and agreeing with the customer that this is what is desired. If the requirements turn out to be stable, the acceptance tests add little cost but large benefits. We do not claim that ATDD works for all kinds of projects, only that it has been successfully used. There is also the perceived benefit of ATDD having a higher upfront cost but saving effort in the long run, since the tests act as a safety harness for making changes and acts as documentation of code. It is difficult to judge how the same projects would succeed without these tests available, neither have we seen any research on the matter.

5.2. Issues with users' ability and concurrent among users

Agile has a mixed impact on this risk. Agile and ATDD both depend on communication with customers, often with differing views on requirements and prioritization. One developer stated *"I find it very interesting in many ways. But it makes it hard for us to stick with like: This is the requirement, we have done this. We always have to think about switching requirements and that's why the automated parts of testing are very good. Because what the test says is actually what we've done. And if the next meeting refines the requirements from the previous meeting, then the tests must change."*

One problem is that different customers specialize in different topics, and the developers need to communicate with different key personnel when they have questions about the various sorts of requirements in the project. The developers we spoke with, who had successfully implemented ATDD, stressed the importance of being on-site. They had also held workshops in the beginning of the project, where they gathered ideas in the form of one-liners, which the business people then graded in terms of business value. These one-liners where then used as starting pointers for further discussion.

The automated acceptance tests were used as a way of easing the cost of constantly changing their code and navigating the views of the customers. One benefit of ATDD is the ability to show all customers the tests, thereby both communicating the existing requirements. The customer has to understand both the domain and the notation for the tests. Fit tests are, however, easy to teach, and people will quite easily understand them. This has also been shown in student experiments [ref]. Making the customer write the tests was, as stated by a developer, *"wishful thinking"*, and they adopted a different approach where the developers wrote the tests and the customer filled in the appropriate data.

5.3. Inadequate user-developer interaction

Agile mitigates this because of frequent user interaction. At the same time this risk is exacerbated if the customer does not trust the process, or if the interaction goes through a few selected customer representatives. To reap the full benefits of ATDD we believe that it requires even more interaction with customers than agile development does, and with the right people. We have previously reported that some developers saw Fit as beneficial even if the customer was not involved in the practice [6], [7]. They did however not gain the benefits automated acceptance tests can give for communicating requirements, only the benefits from regression testing.

Literature suggests that using Fit initially can hinder communication, by focusing too much on preparing 'syntactically correct Fit documents' for the development team [19]. Melnik reports of one project manager saying: "We value conversation more than the actual tool itself [Fit]" [15]. This resembles one of our developers, who states that *"If this tool had not existed we would probably just have done it in another way, and not had any problems with that. But still, this tool did not introduce just the tool, but also the way of thinking [separating test data from test code, thereby making it possible for customers to understand]."* The method leads to sitting down with the customer(s) and discuss requirements (which are the acceptance tests) in a format they can both be comfortable with.

According to the developers, this particular customer [Kunde] had little experience with agile, and little trust in the development process when the project started. Through early focus on delivering value and by repeatedly showing through the automated tests that their functionality delivered as specified, they gained trust. In one project the trust was so high that the developers could deploy every iteration with minimal manual testing, and little customer verification outside the iterative demonstrations that were held. This process requires a lot of commitment from the customer. One developer told us how he thought the customer must have been tired of having developers around, asking questions. This indicates that the development process required a huge effort from both parties, and not all customers are willing or able to participate in such a process. Problematic is also the physical proximity we believe is required to sustain such a process. If successful, on the other hand, the Fit tests can act as *boundary objects*,

We do not state that requirements are only supposed to emerge using Fit tables. Studies show how customers in projects using ATDD bring info sheets like diagrams, callouts and mock screen shots to meetings [20], or that the tests are accompanied by embedded commentaries and occasional diagrams [23]. Emerging requirements, we believe, is about communication. Proximity makes it easier to gain this common understanding of what is actually needed and possible.

5.4. Overlooking crucial requirements

Agile RE has a positive impact on this risk and we believe ATDD has an even larger positive effect here. By directing the discussion around the tests, you get the benefits explained by Ricca et al. [2]. Our developers also preferred to use the tests to improve their communication with the customer, guiding the conversation. This leads to a detailed and carefully written-down set of runnable tests that also function as documentation at an acceptance level, and this can be used by the customer and developer team alike. One study reports a customer that successfully wrote Fit tests [20]. This specific customer, however, had an information systems background. Further, business grade students after having a rough start ended up producing good specifications in the end [24]. Customers have been less interested in writing such tests [19], and they have been seen as complex beasts that were easy to put off writing [20].

Melnik et al. find no evidence that good quality acceptance tests specifications made by a customer team resulted in better implementation by a developer team [24]. This is interesting because it is a student experiment where the tests were specified by one team and developed by another. They suggest that developing using Fit reduces *noise* by making it more difficult to describe irrelevant information, but that *silence* (the tests are lacking important requirements) was not addressed properly in Fit. From our study of a successful case, and by looking at literature, we suggest that *silence* can be reduced by the close and constant communication they had.

We do not have empirical data to show that it is impossible for non-technical customers to write tests that do not overlook crucial requirements, but that some may seem reluctant to put in that extra effort. The developers we interviewed had solved this by writing the tests themselves, after going through them with the customer. Uncertainty and complexity were triggers for using FitNesse as a mediator, and using Fit tables for complex requirements is indeed suggested a good practice also in literature [2]. Used this way, we think Fit tables may help to not overlook crucial requirements.

According to one developer, it was important to note that Fit tables were not well suited for all kinds of requirements: "*It's very good for many aspects of the project, which are like the integration with the other vendors. How the business side wants this data to be displayed for the end-user. It's a useful tool. But for us in planning and structuring how we have to plan tasks, we don't want to just pick one task and just do that and be completely blind about the other tasks. We need some sort of oversight and this tool isn't very... well the tool itself is just one tool, but it's sort of limiting."* We believe that this emphasizes the importance of being aware of the larger structures (i.e. software architecture) while looking at the details, and that this can be eased by continuous interaction with the customer.

5.5. Modeling only functional requirements

This risk is exacerbated by an agile process. We have no findings that show that ATDD mitigates this risk, at least not directly. What ATDD will do is to automate the requirement tests that are possible to automate, and thereby free time to focus on other tasks that are in more need of manual work. The developers we have spoken to state that they had not gotten many complaints about non-functional requirements, something Ramesh et al. also suggest customers tend to do. The developers said that when they considered these requirements, they did not implement them in ATDD. This is as expected. When needing to test non-functional requirements this has to be done in other ways.

5.6. Not inspecting requirements

Agile RE worsens this risk, as there is a lack of detailed specifications. This is perhaps a risk where the practice of ATDD will give considerable benefits, outperforming both RE and agile RE. An important point with ATDD is that it not only functions as automated tests – in the requirement specification phase it also acts as a jointly understood definition of 'done' at an acceptance level. This may ensure the correct requirements are found and thus handled, but so far this has not been shown empirically. Secondly, in ATDD each and every functional requirement is an up-to-date executable test. As one developer claimed, when he thought about tests he really meant automated tests. The developers gradually build a runnable set of tests that envelop the complete system, which in turn can be executed during the entire project life. These tests are, however, only as good as the data used in the tests. One developer claimed that he thought the customer would not be as detailed in their testing if they had to do it manually, and as such the tests provided better requirements coverage than manual tests would.

One must not be led to believe that all requirements should be written as automated acceptance tests. As described in [3], the developers we talked to based their choice of testing strategy on several factors: they used Fit tables for requirements that were uncertain, complex, and were possible to describe in a tabular format. Out of the four projects they worked on, one of them had full ATDD coverage of the requirements; the other three were partly covered by Fit tests. The rest of the requirements were implemented in a regular unit-level TDD fashion, using Cucumber. We should also note that that using ATDD is a demanding practice, requiring discipline. Literature and our own previous studies have shown that developers tend to neglect updating the automated acceptance tests when deadlines were approaching. By doing so the whole harness of tests becomes a burden.

5.7. Presenting requirements in the form of designs

While exacerbating this risk since lack of documentation can create risks for changing existing code, agile development handles this by "embracing change". In ATDD, requirements are described as tables that act as both documentation (the acceptance tests functions as the agreed upon documentation of the requirement at an acceptance level) and runnable tests. We believe this leads to mitigating the risk that the agile process exacerbates. Seen as documentation, the tests make transition for new personnel easier. Seen as executable tests, they make it easier to change exiting code without the fear of breaking code in hard-to-find places. It thus reduces the risk of making changes, allowing for simpler evolution of the code.

5.8. Attempting to perfect requirements before beginning construction

RE tries to document all requirements in the beginning, but these requirements may turn out to be wrong or incomplete. Agile RE focuses on gradually building a set of requirements through an iterative communicative process, but not documenting them in detail. ATDD combines the documentation-focused RE with iterative communication-focused agile RE. This is done by developers meeting the customer (or representatives) and detailing requirements (using acceptance tests to create examples of what 'done' means) and then specifying a subset of these as runnable tests. Further, a changing requirement is handled in the same way; you cannot change the code unless you also change the tests that check it. One should remember that the process of describing requirements as automated acceptance tests means there is a higher up-front cost. This can be balanced by simpler regression phases later on, but one should decide where such a method is applicable.

5.9. Intrinsic schedule flaws

This risk is exacerbated in agile RE. We can so far not see that ATDD directly influences problems of cost estimation and schedule estimation. We do believe that having a coding safety harness such as automated acceptance tests indirectly influences this risk by making the developers trust that they can alter existing code in a safer way. If the developer introduces a bug in some part of the code, the tests make it easier to spot it, and this reduces the time spent debugging. Since the developers are able to more quickly track down bugs it should make it easier to estimate the time (and hence the cost) to make changes. We have however not performed any measures in this regard, nor found any references dealing with this.

6. Conclusion

Acceptance-test driven development (ATDD) has during the last decade gained a lot of attention, but few studies have been performed to describe how this method impacts the requirements engineering. In this article we have tried to answer the research question:

Where can you place ATDD as a requirements engineering practice?

This has led us to look at RE risks identified by Ramesh et al. and see how the use of ATDD influences these risks.

In our opinion ATDD should be placed in the middle ground between RE and agile RE, borrowing aspects from both. It carries its own set of mitigations and exacerbations of the risks identified, as we have shown in our discussion.

ATDD gives detailed documentation of all requirements but it does so so in an iterative fashion. This documentation, built as runnable tests, is founded on close *communication* with the customer. These emerging tests enable automated verification of the business requirements throughout the project life span, and can at the same time act as an up-to-date documentation for both developers and customers. By reducing time spent to test the automated parts, developers can spend more time covering aspects that still require manual labor. Our study has shown that by using ATDD the developers have built a great deal of *trust*, to the point of deploying without much customer testing. The customers have seen that the tests convey their business requirements in the correct way.

It is not necessarily easy to use ATDD if you want to gain all its advantages. Firstly we believe it requires more customer cooperation than regular agile development, suggesting that on-site development should be obligatory. Secondly, not all requirements can be described in this fashion. Careful consideration should be given to each requirement before choosing whether to use automation at acceptance or only at the unit level (in which case acceptance testing needs to be performed in another way). Regardless of the use of automated acceptance tests, unit tests still have to be used. The developers we have interviewed used regular TDD once the acceptance tests were written. Other application domains may not be as good for using this strategy as the example we have studied. Logistics may be particularly good for describing requirements in a table format.

Lastly we see that ATDD is a rigorous method that demands discipline from the developers. If developers neglect to update the tests when deadlines emerge, the framework of acceptance tests quickly become cumbersome to maintain and provides few benefits.

ATDD has a place in the ever-increasing arsenal of development methods. The community itself is questioning its value, some believing that it costs more than it saves (http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html). We think the benefits of ATDD come mainly from its ability to help with two different topics:

Firstly it can help with uncovering the correct requirements. The tests act as *boundary objects* [25], [26], leading to a joint understanding of what 'done' means. Understanding the role of acceptance tests as boundary objects is an uncovered area in research, and we would like to pursue this issue.

Secondly ATDD focuses on detailing requirements in an automatically testable way. Also here little research has been done on this field, particularly on case studies. This work is a first attempt at describing ATDD from a requirements engineering perspective, and we would like to further explore this topic. We need more research on which kinds of domains this method is most useful for. Nonetheless we think the method carries aspects that can be of use no matter what domain is in question.

10. References

[1] R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, 1st ed. Prentice Hall, 2005.

[2] F. Ricca, M. Di Penta, and M. Torchiano, “Guidelines on the use of Fit tables in software maintenance tasks: Lessons learned from 8 experiments,” in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 2008, pp. 317-326.

[3] B. Haugset and G. K. Hanssen, “The Home Ground of Automated Acceptance Testing: Mature Use of FitNesse,” in *AGILE Conference (AGILE),* 2011, pp. 97-106.

[4] B. Ramesh, L. Cao, and R. Baskerville, “Agile requirements engineering practices and challenges: an empirical study,” *Information Systems Journal*, vol. 20, no. 5, pp. 449–480, 2010.

[5] Lan Cao and B. Ramesh, “Agile Requirements Engineering Practices: An Empirical Study,” *Software, IEEE*, vol. 25, no. 1, pp. 60-67, 2008.

[6] G. K. Hanssen and B. Haugset, “Automated Acceptance Testing Using Fit,” presented at the HICSS 2009

[7] B. Haugset and G. K. Hanssen, “Automated Acceptance Testing: A Literature Review and an Industrial Case Study,” in *Agile, 2008 Conference*, pp. 27-38.

[8] P. Saripalli, “Decision support material for incorporating QR technologies,” Masters thesis, NTNU, Trondheim, Norway, 2010.

[9] B. Nuseibeh and S. Easterbrook, “Requirements engineering: a roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 35–46.

[10] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-oriented software engineering: a use case driven approach*. Addison-Wesley, 1992.

[11] J. Vanhanen, M. V. Mäntylä, and J. Itkonen, “Lightweight Elicitation and Analysis of Software Product Quality Goals: A Multiple Industrial Case Study,” in *Proceedings of the 2009 Third International Workshop on Software Product Management*, 2009, pp. 42–52.

[12] G. Kotonya and I. Sommerville, *Requirements engineering*. 1998.

[13] D. Ang, L. H. Lim, and H. C. Chan, “Collaborative requirements engineering: an overview and a proposed integrated model,” in *HICSS*, 1998, p. 355.

[14] F. Paetsch, A. Eberlein, and F. Maurer, “Requirements engineering and agile software development,” in *Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, 2003, pp. 308–313.

[15] G. I. Melnik, “Empirical analyses of executable acceptance test driven development,” University of Calgary, 2007.

[16] S. Ambler, “Beyond functional requirements on agile projects,” *Dr. Dobb’s Journal*, vol. 33, no. 10, pp. 64-66, 2008.

[17] G. Melnik, K. Read, and F. Maurer, *Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective*, vol. 3134. Springer, 2004.

[18] G. Melnik and F. Maurer, *The practice of specifying requirements using executable acceptance tests in computer science courses*. ACM Press, 2005.

[19] P. Gandhi, N. C. Haugen, M. Hill, and R. Watt, “Creating a living specification using FIT documents,” in *Agile Conference, 2005. Proceedings*, 2005, pp. 253-258.

[20] G. Melnik and F. Maurer, *Multiple Perspectives on Executable Acceptance Test-Driven Development*, vol. 4536. SPRINGER-VERLAG, 2007.

[21] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato, and P. Tonella, “Using acceptance tests as a support for clarifying requirements: A series of experiments,” *Information and Software Technology*, vol. 51, no. 2, pp. 270-283, Feb. 2009.

[22] S. Park and F. Maurer, “Communicating Domain Knowledge in Executable Acceptance Test Driven Development,” in *Agile Processes in Software Engineering and Extreme Programming*, 2009, pp. 23-32.

[23] R. Mugridge and E. Tempero, “Retrofitting an Acceptance Test Framework for Clarity,” in *Proceedings of the Conference on Agile Development*, 2003, p. 92.

[24] G. Melnik, F. Maurer, and M. Chiasson, “Executable acceptance tests for communicating business requirements: customer perspective,” in *Agile Conference, 2006*, 2006, p. 12 pp.

[25] S. L. Star and J. R. Griesemer, “Institutional ecology,’translations’ and boundary objects: Amateurs and professionals in Berkeley’s Museum of Vertebrate Zoology, 1907-39,” *Social studies of science*, vol. 19, no. 3, pp. 387–420, 1989.

[26] P. R. Carlile, “A pragmatic view of knowledge and boundaries: Boundary objects in new product development,” *Organization science*, pp. 442–455, 2002.