b
UNIVERSITÄT
BERN

Institut für Informatik
University of Bern

# THREATS TO VALIDITY IN TDD RESEARCH

BsC Thesis submitted by

**TIMM GROSS**

from **Mannheim, Germany**

for the degree of

BsC in Computer Sciences

Thesis advisors
Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari
Software Composition Group
Institut für Informatik

26.05.2020

# Contents

# Abstract

*Context*: Test driven development (TDD) is an iterative software development technique where unit tests are defined before production code. Recent quantitative empirical investigations into the effects of TDD have been contrasting and inconclusive. Additionally studies have shown that TDD is not as widely used as expected. At the same time the body of research contains anecdotal evidence about the usefulness of TDD in practice. This makes it difficult for decision makers in development teams to use the research as the basis for the decision of whether or not to apply TDD.
*Objective*: We present a study designed to uncover the threats to validity in previous studies that prevent them being usable in decision making processes. In order to do that we first studied what values practitioners associate with software testing.
*Method*: We first conducted 15 hours of ethnographically informed qualitative interviews with a small development team to capture the perceived benefits of testing. Then we analysed the threats to validity mentioned in the body of research in a literature review.
*Results*: The interviewed developers put equal emphasis on quality related aspects (i.e. productivity, internal and external code quality) and non quality related aspects (i.e. collaboration, confidence, knowledge transfer, etc.) of testing. In contrast the analyzed research papers focus almost exclusively on quality related aspects of TDD. In addition we identified the common threats to validity in the following areas: participants selection, task selection, context of the study, threats to validity regarding quality, length of observation, amount of iteration, comparisons to other techniques, measuring the adherence to TDD and a lack of qualitative research.
*Conclusion*: Contrasting the views of practitioners on testing and the common threats to validity in TDD research allows us to highlight opportunities for further research. Especially for researchers aiming to provide scientific support for decision making processes of how and when to apply TDD in practice, this study summarizes important aspects to consider.

Chapter 1

# Introduction

Test-driven development (TDD) is a software engineering technique in which failing tests are written before any code is added or changed. This technique emphasizes small iterations and interleaved refactoring. Since it was first proposed twenty years ago (Beck, 1999) a large body of research has been accumulated to empirically verify its proposed advantages. It was argued that the application of TDD leads to improvements in terms of cost, quality and productivity.

In 2002, Kent Beck (Beck, 2002) stated: "No studies have categorically demonstrated the difference between TDD and any of the many alternatives in quality, productivity, or fun. However, the anecdotal evidence is overwhelming, and the secondary effects are unmistakable." This famous quote seems to hold true today in both ways:

- The benefits of TDD have still not been proven in scientific studies.
- There is a large body of anecdotal evidence and expert opinions advocating the usefulness of TDD.

Several systematic literature reviews (see table 3.1) have been conducted to provide a meta-analysis of the state of research (Bissi et al., 2016; Munir et al., 2014; Rafique & Misic, 2013; Turhan et al., 2010; Kollanus, 2010; Siniaalto, 2006), but the results are often inconclusive and sometimes contradictory or inconsistent (Karac & Turhan, 2018). These inconsistencies are highlighted in table 1.1 and table 1.2. Table 1.1 shows the results of the literature reviews regarding quality and table 1.2 shows the results regarding productivity along with a column explaining the main source of inconsistencies identified by the respective papers. In table 1.1 we can see that most literature reviews agree that TDD tends to improve code quality in general. Only Munir et al. (2014) note that some studies do not show a significant improvement in quality when applying TDD. At the same time four of the six literature reviews give specific categories which they thought contribute to the inconsistencies.

For example Munir et al. (2014) and Turhan et al. (2010) pointed out that the claimed code quality gains are much more pronounced in studies which they classify as low-rigor compared to high-rigor studies. These two reviews classified the papers under study according to the rigor in which they applied their methods and statistics. Munir et al. (2014) also classified the literature

**Table 1.1:** Findings of literature reviews regarding quality (adapted from Karac and Turhan (2018))

| Study | Overall conclusion for quality with TDD | Inconsistent results regarding quality are attempted to be explained by comparing studies that... |
|---|---|---|
| Bissi et al. (2016) | Improvement | |
| Munir et al. (2014) | Improvement or no difference | ...use a methodology that is classified as low vs. high rigor in their application of the methodology<br>...are classified as having low vs. high relevance for TDD research |
| Rafique and Misic (2013) | Improvement | ...use control groups applying waterfall vs. iterative test-last processes |
| Turhan et al. (2010) | Improvement | ...use control groups applying processes with a varying degree of iterativeness<br>...use a methodology that is classified as low vs. high rigor in their application of the methodology |
| Kollanus (2010) | Improvement | ...are done in an academic vs. an semi-industrial context |
| Siniaalto (2006) | Improvement | |

along relevance for the scientific community and practitioners and found inconsistent results for studies with high and low relevance. Moreover the studies describe different effects on code quality depending on the setting, i.e. academic vs. (semi-)industrial studies (Bissi et al., 2016; Kollanus, 2010), or depending on the reference, i.e. TDD vs. waterfall (Rafique & Misic, 2013) or on the type of study, i.e. controlled experiments vs. case studies (Turhan et al., 2010).

When we look at the findings for productivity gains from the application of TDD in table 1.2 we see an even less clear picture. Four of the six literature reviews agree that the existing literature provides both examples for higher and lower productivity when applying TDD and conclude that the effect on productivity is inconclusive. Kollanus (2010) reasons that the analysed papers show a degradation of productivity. They describe this loss of productivity as the price for the increased quality of the software developed with TDD (see table 1.1). Munir et al. (2014) does not make a conclusive claim whether or not the application of TDD leads to a degradation of productivity or does not change the productivity of developers using TDD. The categories contributing to inconsistent results regarding productivity are the same as we discussed for the effects on quality, i.e. the setting (industrial vs. semi-industrial vs. academic), the rigor (high vs. low), the relevance (high vs. low), the experiment type (pilot studies vs. controlled experiments vs. case studies) and what TDD is compared to (waterfall vs. iterative test last). In summary we can say that the existing body of research claims that the application of TDD leads to a higher code quality at the price of lower productivity. But these findings have to be handled with care, since the results have multiple inconsistencies of various types.

Anecdotal evidence on the effectiveness of TDD is overwhelming (Beck, 2002). Without providing an exhaustive analysis of this claim, supporting evidence can be found in various contexts. First, in the scientific literature, experts asked about the effectiveness of TDD emphasize the positive effects TDD results in (Erdogmus et al., 2010; Buchan et al., 2011; Scanniello et al., 2016). Secondly, TDD is an integral part of the software engineering curriculum of universities (Kazerouni et al., 2019). Thirdly, if we look at the discourse around TDD on the internet, be it in the form of blog posts or discussions, it becomes apparent that TDD is seen as one of the primary ways to develop software.

Recently two studies have been conducted to find out how testing is actually done "in the wild". Borle et al. (2018) investigated the testing practices of 256 572 public GitHub projects. They found that only 16.1% of those repositories contained test files and only 0.8% strictly practiced TDD. Similarly Beller et al. (2019) observed the work of 2 443 software developers over 2.5 years and discovered that in only 43% of all surveyed projects were test files present and that only 1.7% of all developers followed a strict definition of TDD. Both studies expected a higher rate of adoption of TDD in the real life scenarios they analyzed.

Considering that the research investigating the effects of TDD is inconclusive and that a strict form of TDD is rarely followed in practice, it is not surprising that different developer surveys Runeson (2006) as well as Begel and Zimmermann (2013) identified the questions "[What] percentage of development time needs to be spent on unit testing to ensure quality?", "How much should we test?" and "At what point does it become too much time spent on writing [a] test for every line?", "When do we stop testing?" among others as key questions that have not yet been answered by the software development research community. These surveys have been conducted 7 years apart from each other, while still finding the same question to be relevant.

In summary we see that there is an inconsistency in the TDD research which we call the three contradictions of TDD research for future references (Karac & Turhan, 2018).

1. Research on the effects of TDD is inconclusive
2. Anecdotal evidence from "champions for TDD" is overwhelming
3. The practice of TDD in real life projects is limited

## 1.1 Goal

First, we conducted semi-structured interviews in a small software development department with 5 developers about bugs they fixed for which they either wrote or did not write unit tests. Our goal is to find out what the reasoning for unit testing in an industrial setting is. Initially we wanted to use this data to evaluate whether or not TDD is a useful technique to incorporate into the development policies and provide a road-map for its introduction based on the state of

**Table 1.2:** Findings of literature reviews regarding productivity (adapted from Karac and Turhan (2018))

| Study | Overall conclusion for productivity with TDD | Inconsistent results regarding productivity are attempted to be explained by comparing studies that... |
|---|---|---|
| Bissi et al. (2016) | Inconclusive | ...are done in an academic vs. an industrial context |
| Munir et al. (2014) | Degradation or no difference | ...use a methodology that is classified as low vs. high rigor in their application of the methodology<br>...are classified as having low vs. high relevance for TDD research |
| Rafique and Misic (2013) | Inconclusive | ...use control groups applying waterfall vs. iterative test-last processes<br>...are done in an academic vs. an industrial context |
| Turhan et al. (2010) | Inconclusive | ...are in a pilot study phase vs repeated experiments<br>...are done in controlled experiments vs. industrial case studies<br>...use a methodology that is classified as low vs. high rigor in their application of the methodology |
| Kollanus (2010) | Degradation | |
| Siniaalto (2006) | Inconclusive | ...are done in an academic vs. an semi-industrial context |

research. We changed the focus of this study after realizing that the contradictions of TDD research made it impossible to conclusively provide scientific evidence on the effects of applying TDD in an industrial context, and therefore rendered the initial idea pointless.

Therefore, the second way to contribute consists of an analysis of the threats to validity of a selection of studies done to provide evidence about the usefulness and effectiveness of TDD. We read the studies through the lens of a decision maker in order to focus our analysis. We argue that some of the threats to validity, which we will later identify, need to be addressed if TDD research aims to be useful for decision makers, if they want to incorporate TDD into company policies and the daily development work. With this approach we want to contribute to a question stated by Pedroso et al. (2010): "Are we measuring the right things?" We further think that if future research addresses these threats, not only will TDD research be more relevant to decision makers but also might provide a deeper understanding of the contradictions of TDD research.

## 1.2 Research questions

We answer three research questions in this study:

1. RQ1: What decision criteria are used by software developers to decide if and how much testing effort should be done?
2. RQ2: What threats to validity lead to the three contradictions of TDD?
3. RQ3: How could future research be designed in order to facilitate decisions about if and how to include TDD into company policies?

## 1.3 Structure of this study

We present the findings of the expert interviews in chapter 2 and then the results of the literature analysis of threats to validity in chapter 3. In chapter 4 we provide a discussion on the results and in chapter 5 we then talk about the threats to validity applying to this study. Finally, we answer our research questions in the conclusion in chapter 6.

Chapter 2

# Qualitative expert interviews

This chapter describes the setup and results of qualitative expert interviews investigating how software developers apply testing strategies in the field. More details on qualitative methods that are the foundation for our analysis can be found in Flick (2009).

## 2.1 Experiment set-up

We conducted a total of 15 one hour long interviews with all 5 developers (see table 2.1) of a small development department of a swiss university. The team works mostly on data-integration solutions and system interfaces.

Around 2.5 years ago, the team decided to use SCRUM and expressed the wish to increase the test coverage of all new and existing projects. Therefore the team members have committed themselves to a code review process that includes special attention to the test coverage of code changes before they go into production.

We selected 45 tasks from the issue tracking system classified as “bug” and created after the introduction of the testing strategy. We first used the REST API from the issue tracking system to download a list of all bugs in the system including meta data. Then we excluded tasks that were not in the described time frame, and we performed further analysis. We were able to do that because all commits from the central git repository have a reference to the task they relate to. This also had the additional benefit of including commits to auxiliary resources like included dependencies, configuration, etc. After identifying all commits related to a bug, we analysed these tasks by whether or not unit tests had been written or adapted during the fixing, i.e. whether files containing the word "test" in their titles were added or changed. We then manually looked for bugs for which further inquiry seemed promising, and chose 9 bugs per developer, 3 for each interview. Each interviewee was presented with these 3 bugs, from either the category “Tests were written” or “No tests were written”, which they were assigned to and asked to familiarize themselves with those. Afterwards they chose the one they thought to be most relevant to talk about, thus making sure that the developer validated that the bug was interesting to talk about. In subsequent interviews the categories were alternated. If a developer was asked about a bug

without associated tests in the previous interview, they were then asked about a bug for which they wrote a test and vice versa in the next interview.

The interviews were designed to get insights into the guiding questions in list 2.1. To do that, we designed a questionnaire (see appendix 6). We iterated the questionnaire before and during the interview process. We presented the first draft during a seminar on software engineering and included the collected feedback. After each interview we reflected on the given answers and the notes taken to clarify or to add questions when needed. We paid special attention to the wording of the questions to encourage broad answers and not to be too specific so as to not curtail the possibility for new or unexpected insights into the developers' reasoning.

**List 2.1:** List of the guiding interview questions

- Why do you test?
- Why do you not test?
- When do you stop testing?
- How did you benefit from earlier testing effort?

We conducted the interviews on-site in a one-on-one setting. The interviewees were encouraged to answer honestly, given the fact that the interviewer and interviewee were colleagues that knew and trusted each other paired with assurances that the collected data was only used in an anonymised form. This also mitigated threats to validity regarding evaluation apprehension.

The interviewees were asked to fill out the questionnaire and actively talk about and reflect on the given answers while the researcher took extensive notes and an active role in the process. The interviewer provided clarification, asked for further explanations or prompted the interviewee to expand on ideas that were interesting or new. Afterwards we compiled documents with the original answers and the related notes and let the developers verify that we accurately captured their opinions and reasonings. The resulting document was the basis for the following analysis.

We applied the grounded theory coding technique (Flick, 2009) to the generated data in order to extract the meaning of the given answers. This iterative approach has two main steps. First we applied codes to the material to break it down into smaller parts by summarizing concepts and ideas expressed by the interviewees. Second we categorised these codes into increasingly more abstract themes. Repeating these steps resulted in the categories presented in this chapter.

We also experimented with recordings and transcriptions but quickly dismissed them, because they did not provide additional insights, especially given the amount of work required.

We present the most relevant categories from the data in the following sections grouped by the guiding questions 2.1. Each category is presented by a short summary and one or more quotes from the participants. These quotes serve to exemplify and to increase the intersubjective

plausibility of our conclusions.

**Table 2.1:** Developer experience

| Participant | Position |
|---|---|
| D1 | Junior developer |
| D2 | Senior developer |
| D3 | Senior developer |
| D4 | External contractor |
| D5 | Junior developer |

## 2.2 Results

### 2.2.1 Why do you test?

The first part of our analysis of the answers collected is the description of why developers write tests. Our approach was to let the developers describe one bug they wrote tests for and explain how they fixed it. We then asked why they wrote tests and what they expect to be the benefits of the added tests. In addition we also asked what they gained from the existing tests, and what their overall opinion on writing tests is to further stimulate reflection on tests.

#### Insurance of quality

The developers agreed that writing new unit tests during bug fixes serves them as an insurance of quality for their fix. They feel that they can rely on their solution and are able to quickly adapt it if needed.

> *D2-wt1: [I test to get] Confidence [in the] delivered code, I know what I have written works because I tested it. If I get a follow up [issue] then I need to complement [the unit tests] or write a new one.*

Also the developers feel that writing tests protects them from the possibility of introducing new problems to the code.

> *D4-wt1: [The benefit I get from testing is] Avoidance of side-effects when changing the code.*

And by avoiding the introduction of unintended consequences the developers feel that they improve the long term stability of the system as a whole.

> *D1-wt1: [Von meinen Tests erwarte ich alle] Vorteile, die Tests mit sich bringen, die da wären: Stabilität des Produkts, [...]*

**Table 2.2:** Summary of selected bugs

| Bug | Developer | Existence of tests | Project type | Summary of the bug |
|---|---|---|---|---|
| D1-wt1 | D1 | wt | Data export | Wrong criteria for the selection of a db field |
| D1-wt2 | D1 | wt | Web service | Valid values are rejected by validation |
| D1-nt1 | D1 | nt | Data import | After processing a file, it is not deleted triggering multiple runs |
| D2-wt1 | D2 | wt | Data export | File was created locally but not copied to destination |
| D2-wt2 | D2 | wt | Data import | End user uses invalid data to correct manual data entry errors |
| D2-nt1 | D2 | nt | Web application | Security configuration had errors, not enforcing it |
| D3-wt1 | D3 | wt | Data export | Error in Filter lead to data being omitted |
| D3-nt1 | D3 | nt | Web service | Missing default value for parameter |
| D3-nt2 | D3 | nt | Web application | Absolute links were invalid |
| D4-wt1 | D4 | wt | Data import | A db field was set during insert but not during update |
| D4-nt1 | D4 | nt | Data import | Validation was too strict |
| D4-nt2 | D4 | nt | Data export | Limitation of file size for uploads was too small |
| D5-wt1 | D5 | wt | Data export | Datatype changes during db migration disabled filters leading to unexpected data delivery |
| D5-wt2 | D5 | wt | Data import | Certain transmitted values should trigger a special case and delete another db field |
| D5-nt1 | D5 | nt | Data import | Mature application suddenly skipped received data sets |

**Table 2.3:** Usage of preexisting tests

| Bug | Existing Tests | Utilization of exiting tests | Specific benefits from existing tests (for the bug in question) | General benefits from testing (for the type of bug in question) | Costs of testing |
|---|---|---|---|---|---|
| D1-wt1 | Y | Y | Set up | Set up, structure | |
| D1-wt2 | N | N | N/A | Speed, knowledge transfer, structure | |
| D1-nt1 | N | N | N/A | Confidence, ease of use, speed | Time |
| D2-wt1 | N | N | N/A | Template, Discipline, Knowledge transfer, structure, documentation, confidence | |
| D2-wt2 | Y | Y | Confidence, no unintended consequences, ease of use, speed, documentation | Confidence, no unintended consequences, ease of use, speed, documentation, structure. constant improvement of quality | |
| D2-nt1 | N | N | N/A | Knowledge transfer, structure, no unintended consequences | Time |
| D3-wt1 | Y | Y | Documentation, knowledge transfer, speed, structure | Documentation, knowledge transfer, speed, structure | |
| D3-nt1 | Y | Y | Nothing | Documentation | Time |
| D3-nt2 | N | N | N/A | Documentation, no unintended consequences, Confidence | Time |
| D4-wt1 | Y | Y | No unintended consequences | No unintended consequences | |
| D4-nt1 | Y | N | N/A | Set up, documentation, no unintended consequences | Busy work |
| D4-nt2 | N | N | N/A | Nothing | Time |
| D5-wt1 | Y | Y | No unintended consequences, knowledge transfer | No unintended consequences, knowledge transfer, constant improvement of quality, structure | |
| D5-wt2 | Y | Y | Speed, set up, knowledge transfer | Speed, set up, knowledge transfer, structure, confidence | |
| D5-nt1 | Y | Y | Nothing | Confindence, knowledge transfer | Time |

### Confidence in solutions

One consequence of having an insurance of quality is the resulting confidence in the delivered solution.

> *D1-wt2: Tests sind [meine] Rückversicherung [, dass alles so funktioniert wie ich es erwarte,] vor [der Übergabe an den Betrieb für die] manuellen Tests. Diese Tests helfen besonders in der Zukunft [falls neue Probleme auftreten]: psychologischer Vorteil*

> *D4-wt1: Es ist nice-to-have dass man es [den Bugfix] der nächsten Teststufe guten Gewissens übergeben kann.*

> *D3-nt1: [...] Gesichtsverlust durch häufige Fehler verhindern [...]*

By writing tests the developers feel that they can eliminate the potential of taking shortcuts and problems that result from flaws in their workflow, like forgetting or skipping critical steps. Therefore they use testing as a tool to increase their discipline to stick to their workflow, regardless of distractions and daily variations in performance.

> *D2-nt1: Devs are prone to forgetting things that have to be done manually [and by writing tests I make sure I did not forget anything]*

### Documentation of assumptions

The nature of requirements is that they often leave room for interpretation about how the description in natural language should be transformed into executable code. In this process the developer necessarily needs to make assumptions to a certain extent. The interviewed developers think that it is important to document those assumptions and those test cases are a good tool for this purpose.

> *D3-wt1: Die Requirements sind an vielen Stellen interpretierbar, so dass wir uns mit Test Cases so etwas wie eine definierte Requirements Baseline geschaffen haben.*

> *D2-wt1: In other words it's reaffirmation of requirements and that we have understood them correctly.*

Additionally in the context of this study there is no defined process for requirement engineering in place. This leads to uncertainty in the requirements and it is possible that manual end user testing reveals that requirements were incomplete or that there was a different understanding of how to fix a specific bug between the developer and the person with the domain know-how. The developers see their testing effort as a way to cope with the uncertainty as well as deal with quickly changing requirements.

> *D5-wt2: Additionally there was uncertainty about the requirement in general and I suspected that it will change in the future. Which it did. And thanks to the new tests I could verify that the old requirements were no longer active but the new [ones] were.*

### Future maintainability

Another point voiced by the interviewees is that a good test suite increases the future maintainability of the software, not only for bug fixing but also for future refactorings.

> *D5-wt1: The next person working with the code have an easier time. And when everybody leaves the project in a better state than they found it, there might come a time when you can refactor the code without fear of introducing unintended consequences.*

> *D5-wt2: I wanted to preserve the already high [test-]quality that supported me in fixing the bug.*

> *D5-wt1: Also for me the difference between working on an untested project vs a tested project has been an eyeopener.*

Unit testing is uniquely suited to guarantee future maintainability.

> *D2-wt2: Debugger for example provides no benefits in the future [in contrast to unit tests, that will be run in the future, highlighting possible bugs].*

### Dealing with complexity

Another point is that unit testing supports developers in dealing with complexity.

> *D2-wt2: Investment in requirements and testing is more valuable the more complex the use cases are. Interestingly there is a correlation in our data between complexity and the willingness of developers to write extensive test cases. The more complex the code the higher the willingness to test.*

> *D1-wt1: Ich schreibe eher mehr Tests bei komplexeren Code. But even more pronounced is the other way around. When the complexity is regarded to be very small, there is a resistance felt against writing tests, and it is seen more of a nuisance.*

> *D3-wt1: Grössten Widerstand [gegen das Schreiben von Tests] habe ich, wenn es um den Test von Banalem geht, wie kleine Util-Klassen.*

### Passive knowledge transfer

First, there is a strong sense that letting a good test base deteriorate is akin to letting your team members down. This social pressure is seen as a good way to keep testing discipline high.

> *D5-wt2: [A good test suite encourages] discipline in the next committer. Good tests motivate not to let quality deteriorate.*

Second, and as an indirect result of the first point, good test coverage is seen as a way to transfer knowledge throughout the team. There are two dimensions of this knowledge transfer. There is the project specific knowledge. The tests help other developers unfamiliar with the code to understand how it was implemented, what the architecture is, or how the testing is structured. On the other hand there is also a dissemination of knowledge about testing strategies and tricks how to test certain technologies or code structures. Especially junior developers learn from what they see when they make changes to the code and even transfer this knowledge to other projects they are working on. And in the long run they tend to develop a positive attitude towards testing.

> *D2-wt2: [Testing helps] to share knowledge between team members, for both: testing in general, and the function of the specific code. It is a blueprint for «How to fix it».*

> *D2-wt1: Especially now or in the future when we have fresh software engineers, it will be good to influence them positively.*

> *D1-wt2: Nächster Developer ist eher motiviert auch Tests zu schreiben und kann Bugs schneller beheben, da die Infrastruktur für Tests bereits besteht.*

#### Enjoyment

Last but not least, testing seemed to not be disliked by developers.

> *D1-wt1: Ich möchte Probleme lösen, bei Tests hat man auch Probleme wie bei dem produktiven Code. Daher würde ich sagen ich mag es Tests zu schreiben genau gleich wie ich es mag produktiven Code zu schreiben.*

And there seems to be a correlation between the enjoyment experienced during testing and the experience or proficiency with testing.

> *D2-wt1: I really like it [to use testing] but it is also said that the fun experienced during testing increases the more proficient a developer is with testing.*

Some developers expressed that they have fun while writing test cases.

> *D5-wt1: The more you know about testing, the more fun it is to write them.*

### 2.2.2 Why do you not test?

To uncover under what circumstances the developers skip writing tests, we asked them to reflect on their decision not to include tests. We also asked if they think it would be possible to write a test for the specific bug at all, and what they estimate the cost and potential benefits would be.

It is important to keep in mind that asking those questions has some significant threats to validity in terms of response bias or social desirability. Even though we tried to explain that the interviewee is not under evaluation and that our goal is just to understand the reasoning for whether or not to write tests without judgement, the developers were still apprehensive and were

quick to answer defensively. We attribute this somewhat to cognitive dissonance of the developers. On the one hand they agreed on writing tests but on the other hand for some bugs they have reasons not to write tests. Since this contradiction has not been made explicit the developers resolve it with different approaches. These approaches are exemplified later in the following quotes. One explanation that the developers gave more than once was that the bug was too trivial for it to warrant a new test. We double checked these statements and could always find bugs with a similar scope for which the developer actually wrote bugs. We then asked the developer about this discrepancy by presenting them the bug that was to trivial to test and the similar bug. Every time they then agreed that it would have been better to write a test for the untested bug.

> *D1-nt1: Gegenfrage: Wie würdest du [das] testen?*

And they gave detailed explanations how they would compensate for the omission of writing tests.

> *D3-nt2: [Der Zeitdruck war zu hoch, um ein bisher ungetestetes Modul zu testen.] Es wurde eine Folgestory erstellt um sich das systematisch anzusehen.*

Or how it would not be cost or time-effective to invest into testing for the bug at hand.

> *D4-nt1: To fix the bug as fast as possible and because it was a simple fix I decided not to write any more tests. Additionally, the code will be replaced in the near future and there are only a few changes to expect.*

The last type of defensive answer was to answer jokingly.

> *D4-nt1: Sure it would be easy [to write tests] but it is busy work that no one likes to do :-)*

In summary we underestimated the emotional weight associated with the question "Why did you not test?" because we feel that the participants added "even though we agreed on increasing test coverage" in their head to the question. And we were not able to combat the feeling of being accused in the interviews, either with rephrasing the questions or with assurances that nobody will be judged or evaluated. There might be a possibility that this got amplified by the fact that the interviewer was part of the development team, but we doubt that the answers would significantly differ when given to a neutral person. That being said, we still extracted some factors that lead to no tests being written.

### External dependencies

Some of the bugs studied were caused by not using external dependencies correctly. This includes external libraries or third-party resources, like a file server with missing permissions. The interviewees agreed that in such cases they rely on these resources to honor the defined contracts and that it would not be feasible to include them in their testing effort. The same holds true for included libraries. The developers also expect them to work as they are supposed to.

*D5-nt1: That is hopefully tested by [the library], so I do not need to test it.*

*D4-nt2: Testing of infrastructure makes no sense because this type of tests are hopefully done by the vendor of the product. [...] Analog: Wir testen nicht ob Java korrekt sortieren kann.*

### Configuration

Closely related to the last point is the problem of a wrong configuration for a specific part of a program. Examples of this include, using a wrong service account without proper permissions to write a file on a network share or a default file upload limit that was too small. When such a misconfiguration occurs the impact might be substantial. Still the developers are clear in their assessment that testing the configuration is unnecessary and has potential drawbacks.

*D4-nt2: Jede mögliche Konfiguration zu testen wäre unendlich*

And the most relevant configuration is the one on production systems, which cannot be tested regardless.

*D1-nt1: Tests für die Konfiguration (vorallem Produktion) ist einerseits schwierig (wenn nicht sogar unmöglich wegen Netzwerksicherheit o.ä.), andererseits würden diese Tests wenig bringen (man würde ja auch nicht CRON Expressions testen in der produktiven Umgebung)*

One developer voiced the opinion that configuration issues cannot be avoided with testing strategies but only with sufficient knowledge.

*D4-nt2: Unexperienced developers will need much more time and effort to find such configuration issues. To avoid these kind of bugs in the future a good advice is to reduce the infrastructure dependencies to a minimum or to use a simpler runtime environment. [...] If you must use certain infrastructure there is no way around learning the details of this particular infrastructure.*

We conclude from this that the developers are reluctant to include external dependencies and configuration in their testing strategies but instead want to test only the code they actually wrote.

### Inadequate existing testing suites

Recently the team started to develop and maintain a few simple web applications. This recent shift was not accompanied by the introduction of an automated front-end testing framework.

*D2-nt1: I would say it's possible [to automate testing here] but it would have taken longer time to write and integrate the testing framework into the application. I thought manual testing was enough at this time.*

The developers agree that such an automated front-end testing framework would be valuable, but cannot be introduced as part of a bug fix. Instead it should be introduced systematically.

*D3-nt2: Gui-Testing-Framework Einführung könnte gut sein. Müsste aber gut geplant werden und dann konsequent in allen betroffenen Projekten eingesetzt werden Insbesondere können solche weitreichenden Entscheidungen nicht während dem Fix eines zeitkritischen Bugs getroffen werden.*

#### Shortcuts

Another reason why the interviewed developers skip writing automated tests is that the developers feel that it is not worth the effort and they decide to take a shortcut.

*D3-nt1: [Der Fix hatte eine grosse] simplicity of change. Gerade weil er so banal ist, [wäre] es einfach alle Edge Cases auch in den Tests zu berücksichtigen.*

Taking these shortcuts is often justified by cost/benefit considerations.

*D4-wt1: Der Dev weiss nicht welche Edge-Cases auftreten können, darum stellt sich die Frage, warum testen und nicht flicken? Man darf allerdings die Risikoanalyse nicht aus den Augen verlieren, vor allem wenn es um Datenverlust geht. Zentrale Entscheidung: Wichtig? Dringend? Kritisch? [...] sonst Geldverschwendung*

### 2.2.3 When do you stop testing?

Another block of questions in our questionnaire was aimed at determining the point when developers feel that they tested enough. We tried to make the implicit knowledge of our participants explicit. Contrary to our assumption this proved to be very difficult, mostly because of two reasons. First, the answers were very different from one interview to the next depending on the bug in focus and second, when either prompted to systematically explain their behaviour or confronted with their contradicting answers all developers cited either experience or intuition as the deciding factor. A pronounced example of the first reason was given by an interviewee:

*D4-wt1: Das mit Abstand wichtigste Kriterium [um zu entscheiden ob ich einen Test schreibe] ist Zeit/Geld, sonst nichts, [...]*

*D4-nt1: But to reflect it would have been valuable for further changes of this legacy code to write tests [...]*

Examples of the second reason were numerous:

*D1-wt1: Falls ich das Gefühl habe, dass der Fix komplex ist (reines Bauchgefühl) schreibe ich Tests für alle komplexeren Randbedingungen.*

*D5-wt1: I always use my very subjective intuition. There are a lot of factors influencing this intuition: experience, rules defined by the dev team, rules of thumb read/learned somewhere, the daily mood you are in, the amount of work in the backlog, etc.*

### 2.2.4 How did you benefit from earlier testing effort?

The question "How did you benefit from earlier testing effort?" also provided mixed answers depending on the bug fix focused on. When asking about bug fixes for which the developer did write tests, one category of answers was concerned with both confidence in the solution, by not introducing unintended consequences as well as a higher efficiency when fixing a bug.

> *D4-wt1: The existing tests help to make sure that the change didn't break existing functionality.*

> *D3-wt1: [Die Tests fungieren als] Vorlagen, so dass geringer Aufwand für den neuen Test [entsteht]*

> *D1-wt1: [Durch die bestehenden Tests gewann ich] viel Zeit, da SetUp komplex war*

On the other hand when no tests were written, the answers changed, and the previously mentioned benefits were discarded.

> *D5-nt1: It would have taken a lot of time to add new tests. And I was not sure what to test in the first place.*

A remarkable observation is that there is no consensus in the team what the purpose of testing actually is. All developers agreed that tests are the way to ensure the functionality of the code. One developer in particular stressed that the only benefit of testing is to avoid unintended consequences.

> *D4-wt1: [The benefit of tests is] avoidance of side-effects when changing the code. Ausser diesem Benefit gibt es keine anderen. Doku ist kein Benefit.*

While other developers assigned a broader spectrum of beneficial effects to testing. For example in codifying requirements and making assumptions explicit that were made during implementation, which co-evolves with the production code.

> *D3-nt1: [Tests dienen der] Absicherung/Dokumentation der Requirements.*

> *D2-wt2: If you omit tests the next person fixing a bug is left with little more than a riddle/trial-error.*

# Chapter 3

# Literature analysis of threats to validity

Threats to validity are usually presented in a dedicated section of scientific papers to discuss the potential limitations of the conclusions drawn by the authors. To achieve the goals laid out in chapter 1 we analyzed the threats to validity sections of different papers. Our goal is to find the common themes in these specific threats to generalize them into threats to validity applying to the whole field of TDD research. The method we propose for doing so is a qualitative literature review. An important technique in this method is focusing the analysis by reading the texts through a specific lens. In other words this means having guiding questions that allow us to identify the threats to validity that hinder the applicability of TDD research in the decision making process in industrial contexts. We used a three step approach to compile a list of the biggest threats to validity in TDD research regarding the applicability of TDD research.

In the first step, we gathered all literature reviews regarding TDD which we could find, and validated that we found the most important ones by comparing our list to a meta literature study (Karac & Turhan, 2018). We used eight literature reviews (see table 3.1) to get an overview of the state of research on TDD. Additionally these literature reviews provided insight into common threats to validity and a starting point for compiling a list of categories of threats to validity. From these literature reviews we used a snowball approach to identify potential primary studies to include in this analysis. In the snowball approach we excluded all papers written before 2009. To also include papers written after the literature reviews were published we searched for the terms “TDD”, “test driven” and “test-driven” in several journals (IEEE Transactions on Software Engineering, Empirical Software Engineering, Journal of Systems and Software, Information and Software Technology and Journal of Software: Evolution and Process) and conference proceedings (International Conference on Software Engineering, Conference on Automated Software Engineering, Foundations of Software Engineering, Software Analysis, Evolution and Reengineering, International Conference on Software Maintenance and Evolution, International Symposium on Empirical Software Engineering and Measurement, The Evaluation and Assessment in Software Engineering and International Conference on Mining Software Repositories) from January 2018 till March 2020. This search led to two additional papers being included (Karac et al., 2019; Tosun et al., 2019).

We then started with the second step, the iterative refinement of the identified categories of

threats. In order to achieve this, we picked one primary study, analyzed its setup, execution, conclusion and threats to validity, and then used the results to firstly, refine our list categories of threats, either by adding a new category or by sharpening an existing category, and to secondly provide examples of the existing categories. Additionally we looked at the citations to identify potential papers to add to our list of candidates for in depth analysis. Afterwards we picked another primary study and repeated this process. The selection process for the next paper chosen to be analysed was based on two criteria. First, we preferred studies that were cited multiple times and for which the abstract sounded promising for our purpose. Secondly, we tried to keep a balance between the different types of studies we identified. Namely:

- Experiments: Studies in controlled environments and with predefined evaluation methods
- Qualitative studies: Studies from the field of qualitative research like interviews or focus groups for example
- Statistical analyses: Studies using existing data repositories (like public code repositories, or IDE plugin data) to be analyzed with statistical methods

To determine when to stop the iteration we used a criterion of saturation, meaning that we stopped adding new primary studies once we felt that each inclusion of a new one did not provide any additional sharpening of the identified categories of threats. The analyzed studies can be found in table 3.2 and table 3.8 alongside columns with examples used further down. We reached saturation after analyzing 17 studies. Afterwards we excluded threats that are not specific to TDD research, for example the influence the researcher has on studies, or threats that apply to specific types of studies, i.e. experiments, case studies, focus groups, etc. Examples include hypothesis guessing or evaluation apprehension in experiments. After these two steps we compiled a list of categories of the most commonly cited and the most relevant threats to validity regarding TDD research.

Finally, in the third step, we looked at papers which more broadly discussed the state of research on TDD (Karac & Turhan, 2018; Pedroso et al., 2010; Torchiano & Sillitti, 2009; Erdogmus et al., 2010). These four papers, which we classified as "theoretical discussions", did not do any primary research, but instead discussed the findings so far. For example Erdogmus et al. (2010) put the state of research in contrast to the opinion of a long term expert on TDD. Pedroso et al. (2010), for example, concludes their discussion of the state of research with the question "Are we measuring the right things?" We used these papers to validate our categories and interpretations and to add threats to validity that are not explicitly mentioned in the previous papers. This has the benefit of including threats that apply to the whole field of TDD research, but are not extractable from the primary papers, since they focus only on their specific part of the field. An example of such a threat is the lack of focus on how to include TDD in company policies (see section 3.10). The papers that conducted a specific experiment did not include this threat in their threats to validity section because it does not constitute a limitation to their conclusion but it still is an important threat that limits the applicability of the field of research in general. Therefore without including the papers that discussed the state of research in general we

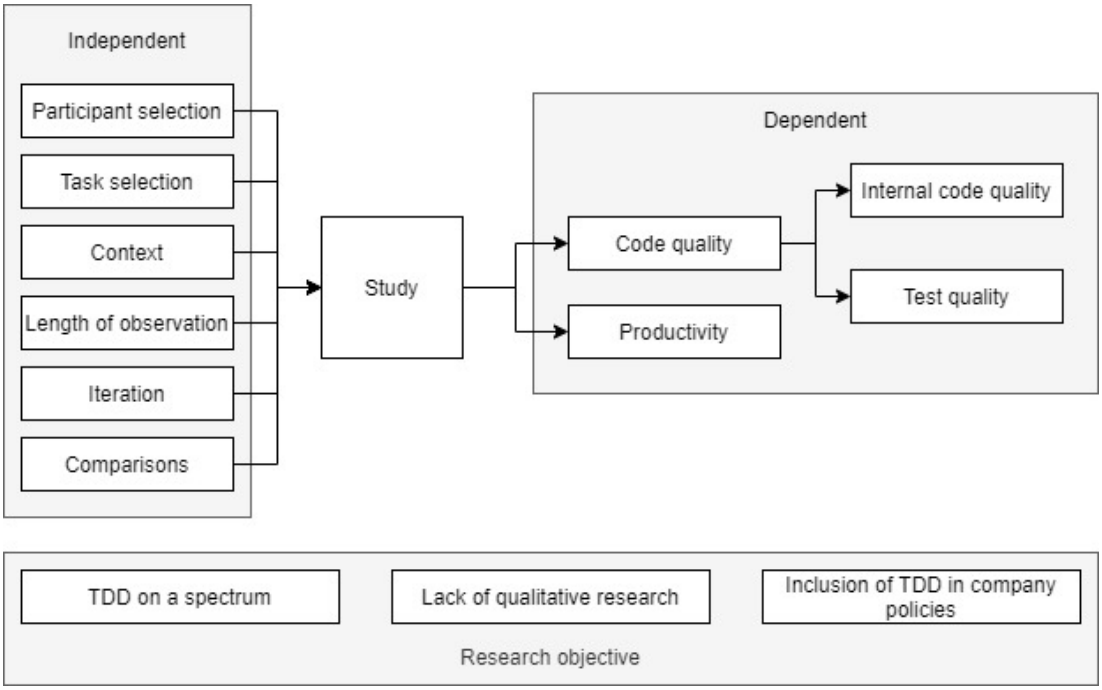


**Figure 3.1:** Threats to validity in TDD research

would not have been able to verify that this is a known and relevant threat.

Combining these sources of information, literature reviews, primary studies and theoretical discussions, we were able to compile a list of potential threats and gaps in the literature that we will present in the next sections by applying a qualitative literature analysis (Flick, 2009). Figure 3.1 gives an overview over the threats to validity identified. On the one hand there are threats affecting the independent variables of studies. These are: participant selection, task selection, context, length of observation, iteration and comparisons. When setting up a new experiment to evaluate TDD these factors should be carefully considered in order to overcome existing threats in the literature. On the other hand there are threats that affect the dependent variables: Internal code quality, test quality and productivity. When setting up a new study special care should be put on the question how these variables are to be measured. Additionally there are threats regarding the research objective. These threats are not specific to one study but instead apply to the whole field of TDD research. As we showed earlier, decision makers lack a scientific foundation for decisions about TDD in company contexts. Therefore if TDD research wants to be applicable to those these threats need to be considered: TDD on a spectrum, lack of qualitative research and inclusion of TDD in company policies.

**Table 3.1:** Overview of literature reviews

| Authors | Title | Method | Studied papers |
|---|---|---|---|
| Bissi et al. (2016) | The effects of test driven development on internal quality, external quality and productivity: A systematic review | Literature review | 27 papers |
| Zubac et al. (2018) | How Does Test-Driven Development Affect the Quality of Developed Software? | Literature review | 10 papers |
| Erdogmus et al. (2010) | What Do We Know about Test-Driven Development? | Literature review | 22 papers |
| Munir et al. (2014) | Considering rigor and relevance when evaluating test driven development: A systematic review | Literature review | 41 papers |
| Rafique and Misic (2013) | The Effects of Test-Driven Development on External Quality and Productivity: A Meta-analysis | Literature review | 27 papers |
| Turhan et al. (2010) | How Effective is Test-Driven Development? | Literature review | 22 papers |
| Kollanus (2010) | Test-Driven Development - Still a Promising Approach? | Literature review | 40 papers |
| Siniaalto (2006) | Test driven development: empirical body of evidence | Literature review | 13 papers |

**Table 3.2:** Primary studies: methods, context and subjects

| Authors | Title | Method | Context | Subjects | TDD Experience of the subjects |
|---|---|---|---|---|---|
| Tosun et al. (2018) | On the Effectiveness of Unit Tests in Test-driven Development | Experiment | Industrial | 24 professionals | 3 day training |
| Pančur and Ciglaric (2011) | Impact of test-driven development on productivity, code and tests: A controlled experiment | Experiment | Academic | 90 students | 2 semester course |
| Fucci et al. (2017) | A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? | Experiment | Industrial | 39 professionals | 5 day training |
| Fucci et al. (2018) | A longitudinal cohort study on the retainment of test-driven development | Experiment | Academic | 30 undergraduate students | 1 semester course |
| Kazerouni et al. (2019) | Assessing Incremental Testing Practices and Their Impact on Project Outcomes | Experiment | Academic | 157 students | 1 semester course |
| Dogša and Batic (2011) | The effectiveness of test-driven development : an industrial case study | Experiment | Industrial | 36 professionals | 3 weeks |
| Fucci and Turhan (2013) | A Replicated Experiment on the Effectiveness of Test-first Development | Experiment | Academic | 58 students | 1 semester course |
| Santos et al. (2018) | Improving Development Practices through Experimentation : an Industrial TDD Case | Experiment | Industrial | 15 professionals | 3 day workshop |

*Continued on next page*

Table 3.2 – *Continued from previous page*

| Authors | Title | Method | Context | Subjects | TDD Experience of the subjects |
|---|---|---|---|---|---|
| Karac et al. (2019) | A Controlled Experiment with Novice Developers on the Impact of Task Description Granularity on Software Quality in Test-Driven Development | Experiment | Academic | 48 graduate students | courses during education (self classified novices) |
| Tosun et al. (2019) | Investigating the Impact of Development Task on External Quality in Test-Driven Development: An Industry Experiment | Experiment | Industiral | 17 professionals | None (short introduction) |
| Thomson et al. (2009) | What Makes Testing Work: Nine Case Studies of Software Development Teams | Experiment/ Qualitative Study | Academic | ca. 36 students (9 teams a 3-5 2-3 year students) | 1 semester course |
| Romano et al. (2017) | Findings from a multi-method study on test-driven development | Qualitative Study | Academic & Industrial | 14 graduate students, 6 professionals | 2 months course |
| Buchan et al. (2011) | Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions | Qualitative Study | Industrial | 5 interviews (4 team leaders, 1 business analyst) | 3 years practice |
| Scanniello et al. (2016) | Students' and Professionals' Perceptions of Test-driven Development: A Focus Group Study | Qualitative Study | Academic & Industrial | 2 focus groups (13 master students, 5 professionals) | students: courses during education, professionals: at least 8 week course |

*Continued on next page*

Table 3.2 – *Continued from previous page*

| Authors | Title | Method | Context | Subjects | TDD Experience of the subjects |
|---|---|---|---|---|---|
| Beller et al. (2019) | Developer Testing in The IDE: Patterns, Beliefs, And Behavior | Statistical analysis | Industrial | 2,443 software engineers monitored over 2.5 years | unknown |
| Borle et al. (2018) | Analyzing the effects of test driven development in GitHub | Statistical analysis | Industrial | 256572 GitHub projects | unknown |
| Bannerman and Martin (2011) | A multiple comparative study of test-with development product changes and their effects on team speed and product quality | Statistical analysis | Industrial | 6 long term open source projects | unknown |

## 3.1 Participants selection

The first category of threats to validity which we identified in our literature review is the problem of participant selection.

First, a lot of experiments and case studies have a rather small sample size limiting the statistical power these studies have. In fact, very few studies have many participants (see column "Subjects" in table 3.2. Table 3.3 shows the number of participants in relation to the context of the study (i.e. academic and industrial). We categorized the number of participants of each paper into fewer than 20, 20-50 and more than 50 participants. For example we can see that 5 studies with participants recruited from an industrial background have fewer than 20 participants, while only 2 studies with students as participants (referred to as an academic context) have fewer than 20 participants. The table 3.3 shows that studies who recruit their participants from companies tend to have fewer participants than studies done with students. We excluded the studies marked as "statistical analysis" because the authors of these studies did not use participants but focused on other data. For example Borle et al. (2018) investigated the testing practices of 256 572 public GitHub projects and therefore the number of participants is unknown. Second, the influence experience has on the proper application of TDD is unclear (see column "TDD Experience of the subjects" in table 3.2).

In table 3.4 we summarized the TDD experience of the participants after the conducted experiment. The two most prevalent categories here are less than one week and one week to half a year. The participants of the studies with less than one week TDD experience, received a couple of days of intensive TDD training in the form of workshops or similar training. Many studies, especially with students, were held during one semester, where the experiments were part of the exercises done by the students. None of the studies we analyzed had participants with proficiency in TDD prior to the start of experiments, with the exception of Buchan et al. (2011). Two studies looked at longer time frames. Pančur and Ciglaric (2011) for example held experiments during two semesters and Buchan et al. (2011) interviewed members of a company three years after they introduced TDD. Again we excluded the "statistical analysis" papers from table 3.4, because the TDD experience of the people involved either is unknown or too diverse.

From table 3.4 we conclude that the experiments have been conducted with participants with little experience in the application of TDD, ranging generally from a couple of days to a couple of months. This experience was gathered as part of the studies. It is noteworthy that no paper conducted experiments with developers that were proficient with TDD and therefore it is impossible to quantify the difference experience makes. Furthermore conducting experiments with developers bears the risk that it is not the impact of TDD that is measured but other factors can influence the results. It is often reported that the application of TDD increases external code quality. But it is impossible to say whether the better external code quality resulted from the application of TDD or from other factors. These other factors might be that the control groups did not write tests at all, or that the group who was asked put more effort in because they knew the were part of an experiment, etc.

There are papers (Porter & Votta, 1995; Höst et al., 2000) which claim that it is a valid approach to extrapolate findings from studies done with students to the behaviour of professionals. Pančur and Ciglaric (2011) argues that students are comparable in skill to junior developers, especially when they are close to finishing their education. But at the same time others (Erdogmus et al., 2010; Scanniello et al., 2016) say that TDD requires a lot of experience and training to be applied correctly and that the benefits of TDD are only manifested after a certain amount of time.

Without contradicting that graduates of computer science and entry-level developers have similar skill levels, we argue that special care should be taken when designing studies to research the impact of TDD. Since the introduction of TDD into the curriculum of universities has proven to be difficult (Kazerouni et al., 2019), and the correct application of TDD requires training and experience, special care should be put on selecting the subjects, depending on what the goal of a study is. Investigating how code quality changes after only a short crash course in TDD might not reveal the whole truth. Furthermore, experience and knowledge of TDD are identified as two of the key factors limiting the adoption of TDD in the industry (Causevic et al., 2011). On the other hand designing experiments with students is vastly easier compared to professionals, not only because of ease of recruitment, therefore it would be unwise to disregard the potential insights gained from experiments with students.

Threats to validity regarding participant selection:

- Small sample size
- Little prior TDD experience
- Transfer of results from students to professionals

**Table 3.3:** Participants by context

| | <20 participants | 21-50 participants | >50 participants |
|---|---|---|---|
| Industrial | Romano et al. (2017), Buchan et al. (2011), Scanniello et al. (2016), Santos et al. (2018), Tosun et al. (2019) | Tosun et al. (2018), Dogša and Batic (2011), Fucci et al. (2017) | |
| Academic | Romano et al. (2017), Scanniello et al. (2016) | Thomson et al. (2009), Karac et al. (2019) | Pančur and Ciglaric (2011), Kazerouni et al. (2019), Fucci and Turhan (2013) |

**Table 3.4:** TDD experience

| | |
|---|---|
| <1 week | Tosun et al. (2018), Fucci et al. (2017), Thomson et al. (2009), Santos et al. (2018), Tosun et al. (2019) |
| 1 week - 0.5 years | Fucci et al. (2018), Kazerouni et al. (2019), Romano et al. (2017), Scanniello et al. (2016), Dogša and Batic (2011), Fucci and Turhan (2013), Karac et al. (2019) |
| 0.5 years - 1 year | Pančur and Ciglaric (2011) |
| more | Buchan et al. (2011) |

## 3.2 Task selection

The next problem identified is task selection. As already alluded to earlier, there is a difference between creating new projects from scratch and the tasks occurring during the long-term operation of an application.

Table 3.5 shows which studies used what kind of tasks during their data collection. Most studies were concerned with between one and four synthetic tasks, like coding katas (the famous bowling score keeper and Mars Rover katas), were used most often. Only two studies were concerned with the work done in real projects (in a company (Dogša & Batic, 2011) or for a client (Thomson et al., 2009)). For two studies no tasks were done. Both are qualitative studies with interviews (Buchan et al., 2011) or focus group discussions (Scanniello et al., 2016). We see that 10 studies use artificially generated tasks or coding katas in their experiments (see also table 3.8, column "Task selection"). This makes sense because those tasks are easily comparable and provide fair comparisons between solutions. Thomson et al. (2009) study students implementing small projects for real clients. The 3 studies from the category "statistical analyses" are evaluating software developers during their normal work (which we excluded from table 3.5). For these types of studies (analyses of GitHub repositories or IDE plugins) we cannot say what kind of tasks the developers were doing. The IDE plugin was used by 2443 developers (Beller et al., 2019). Borle et al. (2018) analyzed 256572 and Bannerman and Martin (2011) github projects for which we do not know the sizes of the development teams. And only one (Romano et al., 2017) of the qualitative studies is concerned with task selection, because they observed students implementing a synthetic task.

Threats to validity regarding task selection:

- Results from real software projects are rare
- Synthetic tasks dominate

**Table 3.5:** Task type: Synthetic vs. Real project

| | |
|---|---|
| 1 synthetic task | Romano et al. (2017), Fucci and Turhan (2013) |
| 2 synthetic tasks | Tosun et al. (2018), Pančur and Ciglaric (2011), Karac et al. (2019), Tosun et al. (2019) |
| 3 synthetic tasks | Fucci et al. (2017), Santos et al. (2018) |
| 4 synthetic tasks | Fucci et al. (2018), Kazerouni et al. (2019) |
| Real projects | Thomson et al. (2009), Dogša and Batic (2011) |
| Not applicable | Buchan et al. (2011), Scanniello et al. (2016) |

**Table 3.6:** Task type: Green- vs. Brownfield

| | |
|---|---|
| Greenfield | Tosun et al. (2018), Pančur and Ciglaric (2011), Fucci et al. (2017), Fucci et al. (2018), Kazerouni et al. (2019), Romano et al. (2017), Thomson et al. (2009), Dogša and Batic (2011), Fucci and Turhan (2013), Santos et al. (2018), Karac et al. (2019), Tosun et al. (2019) |
| Brownfield | Buchan et al. (2011), Scanniello et al. (2016) |

## 3.3 Context

The context in which the studies have been carried out is an important criterion with regards to the results of the studies. We looked at 8 studies with participants from the industries and 7 studies with participants who were still studying. As described earlier (see 3.1) there is a discussion about how comparable these groups are in regards of TDD application. Especially in an industrial context there are fewer participants available to researchers.

The context is also important when talking about the type of tasks that were analyzed. We explained this in more detail in section 3.2. Context in this case describes whether the studies are concerned with projects done from scratch, which we classified as greenfield projects, or with existing projects, called brownfield projects. We summarize this in table 3.6. When only focusing on greenfield projects we risk not taking into account brownfield projects, which are arguably nearer to the daily work of a programmer. Also in brownfield projects testability is very different than when writing new code from scratch. In a real life situation adding a new functionality to an existing project that is largely unrelated to the rest of the project would still be considered a greenfield project. But developers are often changing existing code, either during bug fixing or to implement changing requirements. These types of brownfield projects are not examined by the studies we analyzed. Transferring the results gathered in greenfield project to brownfield contexts is probably impossible and therefore we think the focus on greenfield projects is a remarkable research opportunity, especially because we do not know how TDD performs during bug fixing or refactoring. The promise of TDD is that both should be considerably easier, but from a scientific standpoint that is still unproven. Estimates are that globally developers spend almost half of their time dealing with technical debt, errors, debugging, refactoring and modifying existing code,

with an associated opportunity cost of 85$ billion (Stripe.com, 2018).

Summarizing, we argue that when deciding whether or not to commit to TDD on a company level, the problems and decisions faced by a decision-maker are different from the problems studied in educational greenfield projects, including changing code written by someone else or long term maintainability.

Threats to validity regarding context:

- Focus on greenfield projects
- True brownfield projects are not examined
- Transfer of results from greenfield to brownfield projects

## 3.4 Threats to validity regarding quality

There are several different threats related to different measures of quality mentioned in the studies. In general there are two dimensions of code quality: internal and external code quality. External code quality is usually measured in terms of how well the code covers and implements the requirements or user stories. Internal code quality describes how well the code is structured, how complex it is to understand or how maintainable it is. Internal code quality is only relevant to the programmers that have to work with it, while external code quality is most relevant for the users. When measuring quality an important threat is always how the developers were informed about the goals of the study. For example if the participants are told that their productivity is measured they might optimize for it and only prioritize short term implementation of features and neglect the consequences for long term maintainability. All studies in the experiment category gave strategies how they combated this threat. Therefore we do not study this threat in detail but still want to highlight how important it is to critically think and adjust for this threat.

### 3.4.1 Lack of internal code quality metrics

One of the key promises of TDD is better internal code quality (Beck, 2001). But measuring internal code quality is not a trivial problem. There are different proposed ways, for example Pančur and Ciglaric (2011) measured cyclomatic complexity while Bissi et al. (2016) focused in their literature review only on code coverage, because it is the only metric used by a majority of the analyzed papers. Both approaches are problematic because what they take into account is only a very specific subset of what contributes to high internal code quality. Erdogmus et al. (2010) reported using these metrics in their literature review: “coupling and cohesion, code-complexity measures, and code-density metrics that look at the size of modules or the LOCs required to implement a feature”. They then aggregated these metrics into the categories “better”, “worse” or “no difference” through critical interpretation. Using this classification they report mixed results, with some papers measuring better and others measuring worse internal code quality. When they presented their findings to a senior software developer, he strongly refuted them, claiming that in

his experience TDD significantly improved internal code quality, albeit after a significant ramp up. These examples should highlight the lack of consensus on how to measure internal code quality and what the impact of TDD is on it. For example using lines of code as an indicator for internal code quality has problems. There is a sentiment among programmers that fewer lines of code are fewer places for potential bugs, but it could also be a sign of less readable code.

The used measurements for internal code quality are presented in table 3.8 in column "measurement of internal quality" and a summary can be found in table 3.7. Six studies have used code coverage as a metric of internal code quality. Different measures of code complexity have been used by four studies and a mutation score (based on mutation testing) has been used by two studies as well. Of the studies analyzed six had no measurement of internal code quality whatsoever. In the qualitative studies no internal code quality has been used, because their focus lay elsewhere and is not applicable.

Threats to validity regarding internal code quality:

- No consensus how to measure internal code quality
- Comparing results from different studies is difficult
- Internal code quality is often neglected

**Table 3.7:** Measurment of internal code quality

| | |
|---|---|
| Code coverage | Tosun et al. (2018), Pančur and Ciglaric (2011), Kazerouni et al. (2019), Thomson et al. (2009), Borle et al. (2018), Bannerman and Martin (2011) |
| Complexity | Pančur and Ciglaric (2011), Dogša and Batic (2011), Bannerman and Martin (2011), Tosun et al. (2019) |
| Mutation score | Tosun et al. (2018), Pančur and Ciglaric (2011) |
| None | Fucci et al. (2017), Fucci et al. (2018), Fucci and Turhan (2013), Santos et al. (2018), Beller et al. (2019), Karac et al. (2019) |
| Not applicable | Romano et al. (2017), Buchan et al. (2011), Scanniello et al. (2016) |

### 3.4.2 Lack of attention to test quality

Another threat to validity is a lack of attention to test quality. Often only test coverage is used as a metric, but without regard for the quality of these tests. Similarly how the tests are co-evolving with the production code is not considered, even though that is a crucial component of TDD. Six studies provided code coverage as a measurement of test quality, from those six studies two also used the mutation testing score as an indicator (see table 3.7). The two studies that measured mutation score (Tosun et al., 2018; Pančur & Ciglaric, 2011) saw an increase in mutation score,

**Table 3.8:** Primary studies: task selection and internal quality measurements

| Authors | Title | Generating vs. maintaining code | Task selection | Measurement of internal quality |
|---|---|---|---|---|
| Tosun et al. (2018) | On the Effectiveness of Unit Tests in Test-driven Development | Generation | 2 synthetic | code coverage, mutation score |
| Pančur and Ciglaric (2011) | Impact of test-driven development on productivity, code and tests: A controlled experiment | Generation | 2 synthetic | code coverage, mutation score, complexity |
| Fucci et al. (2017) | A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? | Generation | 3 synthetic | None |
| Fucci et al. (2018) | A longitudinal cohort study on the retainment of test-driven development | Generation | 4 synthetic | None |
| Kazerouni et al. (2019) | Assessing Incremental Testing Practices and Their Impact on Project Outcomes | Generation | 4 synthetic | code coverage |
| Dogša and Batic (2011) | The effectiveness of test-driven development : an industrial case study | Generation | 1 real project each | complexity |
| Fucci and Turhan (2013) | A Replicated Experiment on the Effectiveness of Test-first Development | Generation | 1 synthetic | None |
| Santos et al. (2018) | Improving Development Practices through Experimentation : an Industrial TDD Case | Generation | 3 synthetic | None |
| Karac et al. (2019) | A Controlled Experiment with Novice Developers on the Impact of Task Description Granularity on Software Quality in Test-Driven Development | Generation | 2 synthetic | None |
| Tosun et al. (2019) | Investigating the Impact of Development Task on External Quality in Test-Driven Development: An Industry Experiment | Generation | 2 synthetic | complexity |

*Continued on next page*

Table 3.8 – *Continued from previous page*

| Authors | Title | Generating vs. maintaining code | Task selection | Measurement of internal quality |
|---|---|---|---|---|
| Thomson et al. (2009) | What Makes Testing Work: Nine Case Studies of Software Development Teams | Generation | 3 new client projects | code coverage |
| Romano et al. (2017) | Findings from a multi-method study on test-driven development | Generation | 1 synthetic | n/a |
| Buchan et al. (2011) | Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions | both | n/a | n/a |
| Scanniello et al. (2016) | Students' and Professionals' Perceptions of Test-driven Development: A Focus Group Study | both | n/a | n/a |
| Beller et al. (2019) | Developer Testing in The IDE: Patterns, Beliefs, And Behavior | both | Normal work | None |
| Borle et al. (2018) | Analyzing the effects of test driven development in GitHub | both | Normal work | code coverage |
| Bannerman and Martin (2011) | A multiple comparative study of test-with development product changes and their effects on team speed and product quality | both | Normal work | code coverage, complexity |

but without statistical significance. Since better tests is a promise of TDD this is an avenue of research that might be worth to further investigate.

Threats to validity regarding test quality:

- Test quality is rarely considered
- Co-evolution of production and test code is rarely considered

### 3.4.3 Productivity

Another important category of threats to validity is the measurement of productivity. We identified two different dimensions of productivity that are relevant for software development. First there is the generation of new code, or the short term productivity, and second there is the maintenance of existing code, or long term productivity. It is important to note that the long term view cannot be measured in greenfield projects, as no greenfield study deployed the developed software for any length of time. In the experiment category of studies analyzed productivity is only measured in implemented user stories per time. The majority of studies is concerned with the generation of new code, both the creation of code from scratch and the implementation of new functionality in an existing well tested project (see column "Generating vs. maintaining code" in table 3.8 or the summary in table 3.5 and 3.6). Only 5 non experiment studies are partly concerned with the long-term maintainability of existing code, both in the way new functionality is added to legacy code and the effort required to fix bugs. Those 5 studies are also studying the generation of new code, as are the other 12 studies that only concentrate on new code generation. Other measurements of productivity concerned with long term maintainability are noticeably lacking from the analyzed literature. One example of such a metric would be mean time to fix (MTTF) a bug in a production environment (Erdogmus et al., 2010), which is not covered by any of the studies we looked at. If generating new code is the main measurement of productivity there is the threat that we conclude that TDD results in a lower productivity compared to other approaches because of a higher initial investment (training and development time). Even though it is entirely possible that the true benefits of TDD manifest themselves only during the full lifecycle of an application in the form of reduced effort dedicated to maintenance. Since the initial investment occurs only once it might be worth it if it results in lower investment later on. Therefore we argue that analyzing the long term productivity when using TDD is another way for TDD research to make a contribution.

Threats to validity regarding productivity:

- Focus on short term code generation is prevalent
- Long term maintainability is understudied

## 3.5 Length of observation

According to Beck (2001) the application of TDD should result in immediate benefits in terms of both internal and external quality. On the other hand anecdotal (Erdogmus et al., 2010) as well as empirical evidence (Fucci et al., 2018) suggest that when introducing TDD to developers, the benefits are only manifested after an initial investment and a ramp-up time. If we look at the occurrence of bugs and code maintainability as indicators for external and internal quality, it seems obvious that benefits in those areas would manifest themselves to their full extent only later during the application's lifecycle, since both of these examples are not meaningfully measurable during code generation. The only attempt to measure this is with mutation score which is only done in two studies without statistical significance (Tosun et al., 2018; Pančur & Ciglaric, 2011). This adds to the problem of inadequately capturing some of TDDs proposed benefits, as the studies analyzed are mostly concerned with greenfield projects and code generation (see subsection 3.4.3). Therefore the threat we identified is the length of observation.

There are studies that have tried to analyze the long term effects of TDD, but they are clearly the minority. Fucci et al. (2018) for example is the only experiment that attempts this by designing an experiment aiming at investigating the long term effects of TDD by studying the work of students during multiple assignments over two semesters. Of the non experiment studies there are a couple of different approaches to assess the long term impacts of applying TDD. They used focus groups (Scanniello et al., 2016), interviews (Buchan et al., 2011), IDE plugins (Beller et al., 2019) or a statistical analysis of public github repositories (Borle et al., 2018; Bannerman & Martin, 2011). For example Bannerman and Martin (2011) studied open source projects with a development time greater than 2 years in regards to their application of TDD but was more concerned with how TDD is applied rather than its consequences on productivity and quality.

Threats to validity regarding length of observation:

- Experiments have a short length of observation
- Many benefits of TDD are speculated to manifest only after a ramp up time
- Effects of TDD during the full life cycle of a software project is unknown

## 3.6 Iteration

Another threat to validity is the uncertainty of the factors that are actually responsible for the benefits of TDD. Fucci et al. (2017) ask the question "Does It Really Matter to Test-First or to Test-Last?" Their statistically significant results indicate that the iterativeness of the process is more important than the order in which the tests are written, when measuring quality. This serves as an example of a factor that influences the outcome but is not considered in other studies other than Fucci et al. (2017). Another study (Karac et al., 2019) suggests that the success of TDD is correlated to the division of tasks into small sub tasks. Smaller sub tasks lead to an

increase in iterations. We cannot exclude the possibility that there are other similar factors that are understudied. This leads to the next threat.

Threats to validity regarding iteration:

- Experiments have a short length of observation
- Many benefits of TDD are speculated to manifest only after a ramp up time
- Effects of TDD during the full life cycle of a software project is unknown

## 3.7 Comparisons

Pančur and Ciglaric (2011) speculated that a lot of the superiority of TDD in other studies is a result of a comparison with a coarse grained waterfall process. Compared with a more fine grained iterative approach like iterative test last (ITL) the benefits of TDD would seem smaller. This means not only do we not know what exactly is responsible for the observed benefits of TDD, but also, what benefits we measure, depends on what we compare TDD with. This also makes sense from a historical point of view. Around 2000 (Beck, 1999; Beck, 2001; Beck, 2002) when the agile methodologies, and with that TDD, were first proposed the IT landscape was a different one. Not only were the technologies, like testing frameworks and automated integration infrastructure, not as mature as they are today, but also the development paradigms were mostly akin to the waterfall model, often without any testing. In this context the emergence of advocates for agile methodologies and TDD is not surprising. But now 20 years later, we think it is necessary to reevaluate what factors of TDD we study and what we compare it to.

Table 3.9 shows what the analyzed studies compare TDD to. We identified three categories of comparisons. “Test last” (TL) describes that the tests are written after the production code without specification when exactly. Four studies used TL to compare with TDD. “Iterative test last” (ITL) is similar in that the tests are written after the production code is implemented, but it is supposed to have the same iterativeness as TDD. This means in ITL a small code change is written and the tests are written immediately afterwards. The difference to TL is that tests cannot be written at the end of the development process. ITL was compared to TDD in six studies. The last category is “Your way” which means that the developers do not have any guidelines on when and how to test. This decision is left to the developer. It also means that the developer can write the tests before or after the code, or not at all. Eight studies used this category for comparisons with TDD. In table 3.9 multiple mentions are possible. Fucci et al. (2017) for example compared TDD to both TL and ITL. Karac et al. (2019) does not compare TDD to other methods but instead only compares TDD performance with itself depending on task description granularity.

In a similar vein, Beller et al. (2019) pointed out that we do not have a good and commonly shared definition of TDD. Often TDD means different things for different people. An example would be the role of refactoring in the studies analyzed. Some studies measure it explicitly and even use it to measure how much participants adhere to TDD, while others are not concerned

with it, even though it is supposed to be a key part of TDD. This unclear definition of TDD is even more pronounced outside of the academic world. The application of TDD is often conflated with just the presence of tests, or with the development of tests alongside the actual code.

Threats to validity regarding comparisons:

- No common definition of what TDD includes
- TDD is often compared to non iterative methods

**Table 3.9:** What TDD is compared to

| | |
|---|---|
| Iterative test last | Tosun et al. (2018), Pančur and Ciglaric (2011), Kazerouni et al. (2019), Fucci et al. (2017), Santos et al. (2018), Tosun et al. (2019) |
| Test last | Dogša and Batic (2011), Fucci and Turhan (2013), Bannerman and Martin (2011), Fucci et al. (2017) |
| Your way | Fucci et al. (2018), Thomson et al. (2009), Romano et al. (2017), Santos et al. (2018), Beller et al. (2019), Buchan et al. (2011), Scanniello et al. (2016), Borle et al. (2018) |
| Not applicable | Karac et al. (2019) |

## 3.8 TDD on a spectrum

Furthermore Beller et al. (2019) argued to move away from the term "test-driven development" and towards "test-guided development" after seeing that TDD is very rarely followed to the letter. They claim that this would better reflect the realities of software developers. This leads us to the next threat to validity. As far as we know there is no evidence of what consequences result from not following TDD by the book, but in a variation of it like ITL for example. Given that TDD is rarely done by the book, it is not absurd to assume that the application of TDD is a continuum, where a developer can follow it more strictly or more loosely. New insights into the consequences of applying TDD in non-standard ways would help shape future research. These non-standard ways of applying TDD include for example doing TDD but neglecting the refactoring stage or not adhering to the order in which tests are written as in iterative test last (ITL). This also could support the developers in not stressing about whether or not they follow TDD correctly, but instead they could treat TDD as another tool available to them, applied in various degrees depending on the task at hand.

Threats to validity regarding TDD on a spectrum:

- It is unknown how rigidly a developer must adhere to the TDD process to benefit from its promises

## 3.9 Lack of qualitative research / narrow focus

Additionally, we identified a threat in the narrow focus of the body of TDD research. Most studies analyzed consider TDD as a treatment to a problem and compare this treatment to alternative treatments to evaluate the benefits of TDD, very similar to the way drug treatment research would be carried out in the field of medicine. We do not want to dispute the validity and effectiveness of this approach, but would argue that there are more factors at play when practicing TDD in an industrial context, than are detectable by these studies. Viewing TDD as a treatment of the problem of implementing a functionality is a very mechanical, technical approach. We see TDD also as a way for developers to organize their work and to some extent a way of collaboration in development teams. This point was not studied in depth by the analyzed studies. Two qualitative studies (Buchan et al., 2011; Scanniello et al., 2016) hint at a difference in collaboration when doing TDD. Another point raised by those qualitative studies is that using TDD regularly shapes the way developers think about the problems they have to solve. We cannot say with certainty how TDD influences those points, since it was not the focus on either study. That is why we would argue that it would be worthwhile to do more qualitative research to answer questions not only focused on the effectiveness of TDD but also regarding the "How" and "Why" developers apply TDD in their work. These questions would for example include the individual perception and resistances of developers regarding TDD, the influence of the environment (company policies, development guidelines, workload, etc.) or the influence of power relations in development teams and companies. This could also contribute to closing the knowledge gap which parts of TDD are actually responsible for increases (or decreases) in code quality and productivity, like the iterativeness of the process (see 3.6).

In another field of software engineering research, namely computer supported collaborative work (CSCW), qualitative research has contributed greatly to understanding problems experienced in both research as well as industry (Curtis et al., 1988; Dittrich et al., 2007). In the 1990s researchers as well as practitioners were concerned with the state of software engineering which was perceived as a "software crisis". The contemporary software development practice was seen as problematic: development projects were not finished on time and budget, and some projects had to be terminated without any result. Especially in software for collaboration, there were examples of projects backed by significant amounts of empirical research that still failed. Qualitative research was applied for its ability to understand the reasons for these failures, which were undetected by quantitative research. For an excellent summary in more detail the reader is referred to Dittrich et al. (2007).

There are a few qualitative studies on TDD (Buchan et al., 2011; Scanniello et al., 2016; Romano et al., 2017; Thomson et al., 2009). Romano et al. (2017) used a multi-method approach, consisting of observations, interviews and quantitative methods. Buchan et al. (2011) conducted semi-structured interviews, Erdogmus et al. (2010) applied an expert interview to validate their findings and Scanniello et al. (2016) used focus groups to evaluate a training course. Another promising approach practiced in CSCW research are interdisciplinary research groups. It might be valuable to include researchers with a background in research of work- and organisational

psychology or management theory as the human factor seems to be relevant when applying TDD. We are not aware of any interdisciplinary studies on TDD.

Threats to validity regarding lack of qualitative research:

- Qualitative studies show promising results but are not often done
- TDD is rarely viewed as a means to collaborate in software teams
- Differences in team culture when applying TDD or not is not studied
- TDD is only analyzed through the lens of software engineering. Interdisciplinary studies are non existent.

## 3.10 Inclusion of TDD in company policies

The studies proposed in section 3.9 would focus less on the technical side of TDD and more on how to implement TDD in industrial contexts and what consequences including TDD into company policies has on the company and the people working on code. They could also contribute to questions like: What are the pitfalls and difficulties when introducing TDD to existing teams? How should we handle different understandings of TDD and different approaches to it? When should we use TDD and what tasks are more suitable than others for it? Studies that already try to answer similar questions are from Causevic et al. (2011), who summarized what hinders the adoption of TDD in industrial contexts and identified seven factors: "increased development time, insufficient TDD experience/knowledge, lack of upfront design, domain and tool specific issues, lack of developer skill in writing test cases, insufficient adherence to TDD protocol, and legacy code". Scanniello et al. (2016) used focus groups to achieve a deeper understanding of the problems experienced during the learning process of TDD identified four problems: "applying TDD without knowing advanced unit testing techniques can be difficult, refactoring (one of the phases of TDD) is not done as often as the process requires, there is a need for live feedback to let developers understand if TDD is being applied correctly, and the usefulness of TDD hinges on task and domain to which it is applied to". The results of these two studies already highlight interesting problems but provide no solutions. Finding these solutions are great possible starting points for further research.

As explained in chapter 1 our original intent was to assess the state of research on introducing TDD in companies. From this perspective it constitutes a remarkable knowledge gap that the consequences of inclusion of TDD in company policies is not more widely studied in the literature.

Threats to validity regarding the inclusion of TDD in company policies:

- How to apply TDD in companies is not studied at all
- The side effects of TDD in development policies are unknown

Chapter 4

# Discussion

In the interviews we held during this study (see chapter 3), we asked the developers to provide their perspective on why and when they used testing during bug fixing. We compiled the main drivers for writing tests in table 4.1. The reasons for writing tests fall into two categories. First, we have items that directly concern the quality of the code itself, for example “insurance of qualit”, “dealing with complexity” or “future maintainability”. Second, there are reasons for testing other than quality of the code. These reasons are either related to the collaborative nature of developing software in a team or to advantages perceived by the individuals, for example “documentation of assumptions”, “passive knowledge transfer” or “fun” and “confidence in solutions”. We will refer to the second category of reasons for testing as non-quality related. As we will see later the non-quality related aspects of testing are often not the focus of empirical TDD studies.

We also asked the interviewees for the reasons they do not test, shown in list 4.1. Even though we tried to rephrase the questions and assured the interviewees that they are not judged based on their answers, the answers given were often defensive, and we suspect do not show the whole picture. Questions from the category “Why did you not write tests?” seemed to be understood as an accusation, regardless of how the the question was phrased and assurances that it is not an accusation. For future qualitative research in the area of testing we therefore suggest that special care should be put on assessing the emotional weight of the questions asked. In addition to the pull and push factors of testing, we also found that developers have a strong intuition when to utilize testing in their daily work. At the same time the developers strongly refute the notion that testing should be used during the fix of every bug. Both management of the development team and the developers themselves expressed a wish for techniques or guidelines to better manage their testing efforts and to increase test coverage. Together with the authors of this study they decided to investigate test driven development for its proposed benefits in those areas.

As shown in chapter 1 the existing body of research in TDD is inconclusive and TDD is not as widespread as initially expected, while still being praised by advocates of TDD with anecdotal evidence. Karac and Turhan (2018) already highlight this problem in an excellent paper. They argue that instead of focusing on when tests are written, we should focus on the context. For example the rhythm of development (short and steady development cycles) is more important than when tests are written and different tasks lend themselves better to TDD than others. TDD can be

**Table 4.1:** Categories of reasons why interviewees test

| Reason for testing | Quality related vs Non-quality related |
|---|---|
| Insurance of quality | Quality related |
| Future maintainability | Quality related |
| Dealing with complexity | Quality related |
| Confidence in solutions | Non-quality related |
| Documentation of assumptions | Non-quality related |
| Passive knowledge transfer | Non-quality related |
| Fun | Non-quality related |

a valuable tool for self-discipline during testing, to prevent shortcuts and skipping testing. The developers should be mindful of potential trade-offs when applying TDD, especially regarding productivity and test quality (maintainability and test co-evolution with production code). Faced with the uncertainty about the effects of applying TDD Karac and Turhan (2018) suggest to let the developers decide for themselves when and how strictly they want to apply TDD, because they argue "happy developers are more productive and produce better code". While this is a valid conclusion we want to expand on it, by arguing for another perspective on the problem. Instead of focusing on the single developer we want to put the development team and the associated decision makers at the center of the question, how and when to use TDD. Especially because we did not find any studies on how to incorporate TDD into development or company policies, we think that there is a noteworthy research gap. With this as a starting point, we investigated the body of research on TDD from the last decade, focusing on the threats to validity, which could explain some of the inconsistencies in the results of TDD research regarding its benefits. We identified several categories of threats to validity described in chapter 3 and summarised in list 4.2.

**List 4.1:** Categories of reasons why interviewees do not test

- External dependencies
- Configuration
- Inadequate existing testing suites
- Shortcuts

Most experiments we reviewed use an approach similar to studies from the medical field. They view TDD as a treatment for the problem of producing code with high productivity and high quality. Often this treatment is compared to a different treatment or a control group respectively. This approach has several key benefits but also some shortcomings. For example it is tried and tested thoroughly and used to great success in for example drug effectiveness studies. It also lets the researchers make conclusions with statistical significance. The underlying assumption of this approach is that the application of TDD leads to better results in a mechanistical way. Mechanistical in this context means that TDD is treated similar to a drug or a machine in that given the right inputs TDD produces better outputs. This approach is useful to evaluate

factors that are measurable with appropriate statistical means. But for factors that are either not straightforward to measure or factors that apply to human interaction this approach is less suitable. As shown in section 3 for the case of external code quality this assumption seems to hold, since there are solid ways to measure external code quality, for example by using user acceptance test suites. As we have shown the application of TDD leads to better results compared to many other development techniques. Now we want to talk about the limitations of this approach. Some of the proposed benefits of TDD are difficult to measure, like internal code quality or test quality and therefore establishing causal relationships with certainty is more difficult. The measuring of these proposed benefits would require expensive long term experiment set-ups, such as the long term productivity of development teams, or the long term maintainability of software projects. Another important limitation to this approach is that it is not designed to explain human interaction. As shown in chapter 2 and by looking at the factors that lead to the development of TDD (see chapter 1) there are factors of TDD that help to structure the way developers work together. So these factors are often not studied or studied in a less rigorous way (see for example section 3.4 or section 3.9). We conclude that the research approach chosen by most studies is blind to some of the key reasons why TDD was proposed in the first place. Therefore there is a considerable risk to miss important results that could give a clearer picture to decision makers who ask if and when TDD should be introduced to the development process.

**List 4.2:** Categories of threats to validity in TDD research

- Participant selection
- Task selection
- Context
- Quality
  - Lack of attention to internal code quality
  - Lack of attention to test quality
  - Productivity (short term vs long term)
- Length of observation
- Comparisons
- Lack of qualitative research / narrow focus
- TDD on a spectrum
- Inclusion of TDD in company policies

When we look at the reviewed papers through the lens of a decision maker in development teams, we find another shortcoming of the existing body of research. As described earlier the interviewed developers do not only consider quality related aspects of testing when deciding what and when to test (see chapter 2). Non-quality related aspects, like documentation or passive knowledge transfer, are a big concern for them. In Buchan et al. (2011) the challenges experienced by a development team that introduced TDD as their technique of development

are also not only quality related but also social. Results from Buchan et al. (2011) include that some developers picked up TDD very fast while others needed constant reminders of the benefits. A big challenge was reported to be that upper management needed to be convinced that the effort spend on testing was worthwhile as they were concerned that implementing new functionality was neglected. They also concluded that without developers dedicated to the introduction of TDD it would not have been possible. Three years after the process of the introduction of TDD started all parties involved were convinced that the benefits of TDD overweight the initial investment necessary. These factors are considered to be social and political and equally important as the actual code quality and productivity. And TDD is not only seen as a way for individuals to produce better code faster, but a way of development teams to organize their work in a company context and to collaborate. But besides Buchan et al. (2011) we have not seen any other paper concerned with the social factors of applying TDD in development teams.

We argue that TDD research should be relevant for decision makers, and then these non-quality related factors, like the social and political aspects, need to be studied in detail. How to incorporate TDD in company or development policies is a question that cannot be answered by comparing code quality and productivity of developers using TDD with developers not using TDD as done with the approach described earlier. Instead qualitative research would be necessary to analyze the viewpoints of the practitioners of TDD, the dynamics inside development teams and the cooperation and power relations between actors outside the development team. This need for more qualitative research is also expressed in the qualitative studies we analysed. For future research an interesting approach could be to identify factors that hinder an effective implementation of TDD in real work situations and therefore limit the potential of the benefits promised by TDD. Good starting points might be this study as well as Buchan et al. (2011).

Additionally we think that interdisciplinary studies, incorporating the views of software engineers, economists, psychologists with a background in work organization or researchers from other social sciences might prove very fruitful in uncovering the non-quality related factors when introducing TDD and how to treat them, as explained in section 3.9.

Finally, we want to provide an example study of how we would suggest to tackle the identified threats to validity. An overview of this experiment can be found in figure 4.1. First, we argue that new experiments should keep the threats in mind that were summarized in figure 3.1 under “Independent” factors and “Dependent” factors. When designing such an experiment the participant selection should consist of as many experienced professionals as possible that are already proficient with TDD. We would give these developers tasks to implement in an existing project that is already mature enough to run in production, albeit with a couple of bugs existing. These tasks would simulate changing requirements by constantly iterating on the same functionality of the chosen project simulating a brownfield project as it would occur in real life software projects. Additionally the participants would be asked to fix the existing bugs. Such an experiment could only be designed in an industrial context because students probably are not yet proficient enough with TDD to qualify in the participant selection stage. This experiment would have to be carried out during multiple sessions to address the length of observation threat, as the developers could not optimize for short term gains since they would have to keep the long

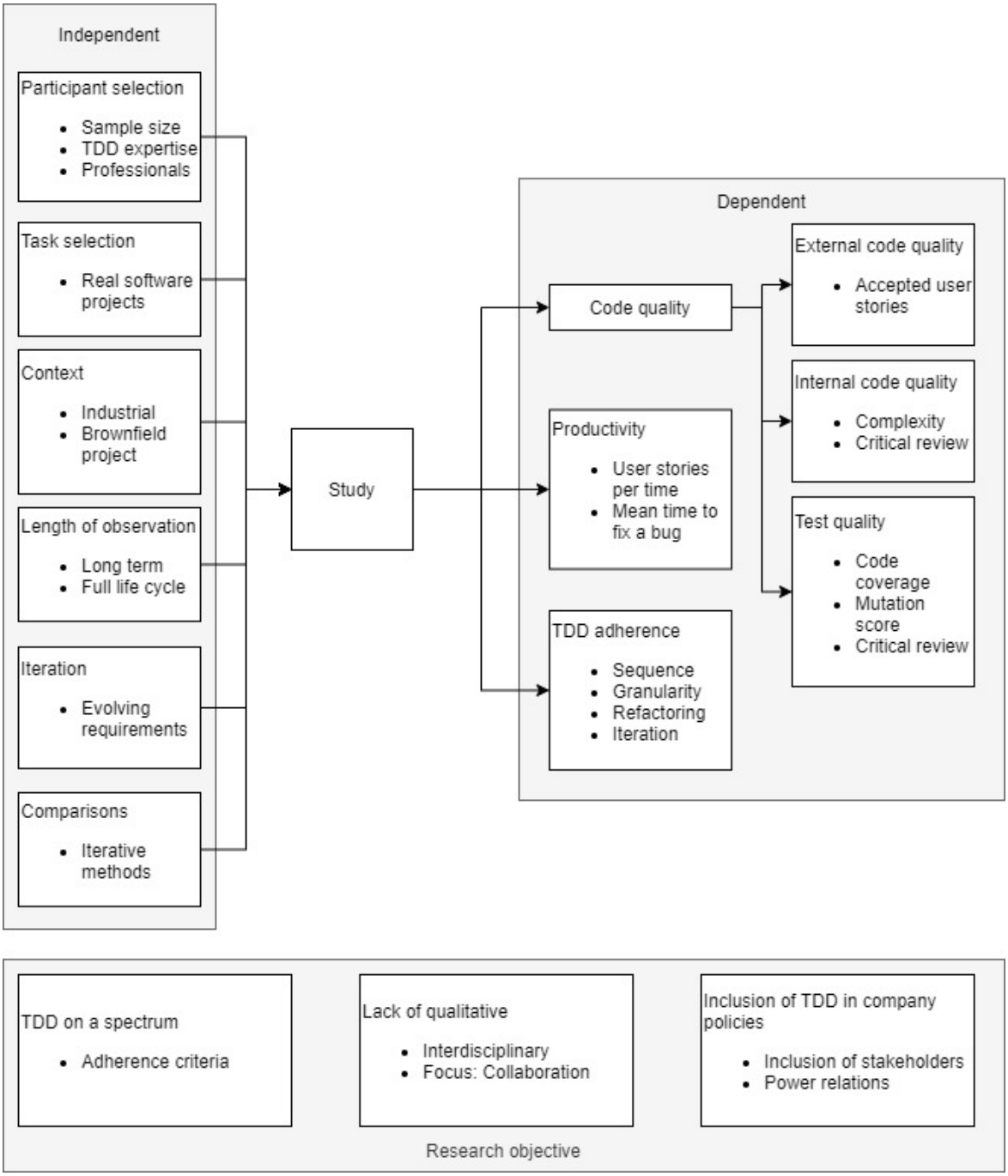


**Figure 4.1:** Threats to validity in TDD research

term maintainability of the project in mind and work on it for longer to simulate how software is developed during its full life cycle. This constant iteration would then allow us to monitor how the code evolves over time. We argue that we should have a control group that implements the same tasks with another iterative development approach, for example iterative test last. We would also compare different teams practicing TDD to each other depending on how strictly they adhere to the TDD protocol to gain insights into the effects of practicing TDD on a spectrum. This brings us to the "dependent" factors of the proposed study or in other words what we would measure. First we suggest measuring how closely TDD is being followed (see Fucci et al. (2017) for a starting point) by looking at the sequence in which the TDD phases are being applied, by looking how granular each task is being worked on, by checking if the refactoring phase is done accordingly to the TDD process and how the developers iterate during development. We would of course also measure the external code quality by means of user acceptance tests. The other factors of code quality, internal code quality and test quality, should be monitored closely. Both factors are not trivial to measure. Internal code quality for example can be approximated with code complexity but we would argue to add code reviews in order to get critical evaluation from experts included in this study. Test quality is also difficult to measure and next to code reviews we would also apply code coverage and mutation score metrics. The experiment setup allows us to efficiently measure both long and short term aspects of productivity. One productivity measurement that was not yet included in any study we looked at is mean time to fix a bug which could be very worthwhile to study in order to capture the proposed benefits of TDD. In addition we suggest also using the often used metric of how many user stories can be implemented during a certain timeframe. Some of the threats to validity in figure 4.1 under "Research objective" are more difficult to include in our example study. We already included the question of whether TDD can be performed on a spectrum in the experiment design. So far we used qualitative methods only to manually evaluate internal code quality and test quality. But it would be worthwhile to include semi-structured interviews with the participants to not only capture what the developers do but also why they do it. The process we used, described in chapter 2 could serve as a starting point for such an analysis. It might be helpful to include researchers from different disciplines, like management theory, psychology and social sciences, to get different viewpoints in the analysis. We argue that the focus of this qualitative analysis should be on collaboration inside a development team and with external stakeholders as these questions are relevant when deciding how to implement TDD in companies. Lastly, the question "How can we effectively include TDD in company policies?" is even more difficult to answer and to design an experiment for. We would need to expand the proposed experiment significantly in order to attempt to answer it. Unfortunately the collected data does not allow us to draw conclusions on how to design such an interdisciplinary study in more detail therefore we want to refrain from making assumptions not based on our findings and will leave this challenge to future researchers. We are aware that the proposed study would be very complex and expensive. Therefore we do not expect this study to be performed but thinking about the threats to validity affecting the field of TDD research can help researchers to mitigate these threats in future studies. Also in this proposed study we only addressed the threats to validity identified in this study as specific to TDD research. Obviously other generic threats would also apply, examples would include hypotheses guessing, selection bias, etc. just to name a few.

# Chapter 5

# Threats to validity for this study

The two distinct methods used in this study warrant two sections of threats to validity. In the first section we discuss the threats to validity regarding the developer interviews, while the threats to validity for the literature review are discussed in the second section.

## 5.1 Interviews

When conducting qualitative studies there are several common threats to validity (Flick, 2009). One of the most important quality criteria in qualitative research is intersubjective plausibility. In other words, is it plausible that other individuals presented with the same data could come to the same conclusions? To increase the intersubjective plausibility we provided a detailed description of the setting, the participants and our approach in chapter 2. We also had discussions about our findings with software engineering researchers as well as with developers, both with the interviewed developers and with developers not in the interviewed team. We believe that this approach minimized the risk that other researchers would draw different conclusions when presented with the collected data.

We put special care in designing the questionnaire. We presented it during an university seminar and to a trained psychologist and incorporated all collected feedback. Additionally we iterated on the questionnaire during the interviews when we encountered unexpected difficulties during the actual interviews. This method of incorporating different data sources to verify our findings is called triangulation in the context of qualitative research.

In the following subsections we want to focus on the threats to validity most relevant for the context of this study.

### 5.1.1 Social factors

As discussed in chapter 2 we encountered the problem that the answers given changed depending on whether tests were written or not. We attribute this behaviour to the fact that the developers view testing as an integral part of their work, especially given that the team agreed on increasing test coverage. This also means that when no tests were written, the developer's answers were

defensive and inconsistent with other answers where tests were in fact written. Even though the developers gave explanations for why they did not write tests for the bugs fixed, we still felt a degree of evaluation apprehension (or social desirability bias) or cognitive dissonance. Evaluation apprehension describes the desire of the interviewees to be viewed as competent programmers by the interviewer. While cognitive dissonance in this context means that there is a discrepancy between what the developer thinks about themselves and what they actually do. This discrepancy is then resolved by retroactively finding explanations for their behaviour. We cannot conclusively say to what degree cognitive dissonance or evaluation apprehension influenced the answers of the participants, but we suspect that a combination of both lead to defensive answers. Additionally, we conclude that developers have implicit knowledge about when to test but struggle making this knowledge explicit. Both factors, the evaluation apprehension/cognitive dissonance and the difficulty to make implicit knowledge explicit, are the most severe threats to validity for the interviews. We tried to combat that in multiple ways. First, we repeatedly explained that the data will only be used in anonymous form and that no evaluation will take place. Secondly, we also repeatedly rephrased and reordered the questionnaire to combat this threat. Finally, the interviewer has a very good personal and professional relationship with all participants, which we leveraged to make the interviewees feel safe and comfortable to talk openly. Our plan to turn implicit into explicit developer knowledge was limited because we were unable to get past their initial defensive reaction. Therefore we cannot confidently say that we eliminated this threat entirely. This in itself, however, is an interesting finding. We recommend to future researchers who intend to do qualitative research with developers to put extra care into these points. In general, confronting experts with their work that goes against what they think constitutes good quality or violates what they consider best practices, was experienced by us as very difficult. We underestimated this, especially given that when the developers made exceptions to their best practices they were able to give sound reasons for the exceptions.

One interesting note is that the degree of defensiveness of the answers varied based on the position of the interviewee relative to the interviewer, who is part of the development team. The team lead and the external contractor, being higher or outside the hierarchy, were more open in their answers, while the other developers, on the same level of hierarchy as the interviewer, felt less confident in giving answers that could be interpreted as violations of best practices. Therefore we cannot exclude the possibility that the answers would be different if the interviewer would have been a neutral person from outside the development team.

### 5.1.2 Author as team member

Another threat to validity is that the author of this study is a part of the development team. This results in two distinct threats. First, the author needs to make sure his involvement in the research field does not influence his research findings and second the author participated in the interview process as an interviewee. How these threats are mitigated is explained in the following.

The threat of a conflation of the role of the author as a team member and an observer, is a common threat with all qualitative studies, where the observer is deeply participating in the field that is researched. To mitigate that threat the author constantly reflected on his two different

roles during this study to keep them separate. Another way to mitigate this threat was informant verification. During the interpretation phase of this study the participants were constantly presented with the interpretations of their answers and asked to verify that the conclusions drawn were valid and represented what they thought. This was done in informal meetings about the findings. During this meetings the participants showed great interest in the findings and how those could be applied to improve their testing strategies or the testing guidelines of the company. This curiosity confirmed that developers are interested in the questions we asked.

The author also participated as an interviewee in the interview process. The benefit of this approach is that the questionnaire could be thoroughly tested before it was presented to the other interviewees. This resulted in better questions because unclear questions were found early, and could be rephrased. Also the order of questions was improved in advance. Since all answers of the author were given before any other participant was interviewed, we were able to exclude the risk of answers being influenced by the knowledge of answers given by other participants. We also checked the answers afterwards for inconsistencies with the rest of the answers, but could not find any meaningful differences that would invalidate the perspective of the author as a part of the development team. Therefore we decided to include the answers into the results of this study.

## 5.2 Literature review

In this section we want to discuss the threats to validity encountered during the compilation of our literature review of common threats to validity in TDD research.

### 5.2.1 Selection

When doing meta literature analyses there is always the risk of omitting relevant papers. We tried to mitigate that risk in two ways. First, we gave a detailed explanation of our decision criteria on whether or not to include a paper in chapter 3 as well as an explanation of our iterative saturation approach. Second we explained the lens we used through which the literature was analysed. In our case the focus was the viewpoint of a decision maker and how the existing body of research can help inform decisions on a company level about the introduction of TDD into their development policies. With this information we aim to make this study replicable by other researchers and relevant for real software development teams.

### 5.2.2 Approach

The literature reviews about TDD we found (see table 3.1) all have in common that they focus on the results of the analysed papers. We took a different route focusing instead on the threats to validity in those papers. This presented a unique challenge since the way the analysed studies report their threats to validity varies greatly. Additionally, we felt like being able to provide additional insights by focusing not only on the threats explicitly stated in the papers but also on more broad threats out of scope of the individual papers an example being the lack of

interdisciplinary research. To be able to do that we needed an approach to analyse the content of the papers, make connections between them and be able to reliably and transparently make our interpretations plausible. To meet these challenges we applied a saturation approach. By treating the papers as artifacts to be understood through qualitative literature analysis (Flick, 2009), we were able to extract what we found to be the most important factors. This is suitable because we do not aim to causally explain but to understand the challenges of TDD research in the context of real world application. Through constant iteration we mitigated the risk of missing aspect in our analysis as well as oversimplifying the results. The use of saturation in our analysis helped us to make sure that we did not prematurely stop including more entries and that the categories of threats to validity were stable even when adding more papers. The threat of missing aspects was further mitigated by looking at papers that discussed the state of research without doing data analysis themselves in an essay form to make sure we include their points of view as well (Torchiano & Sillitti, 2009; Erdogmus et al., 2010; Karac & Turhan, 2018). These essays were written by scientists with many publications on the subject published and we could validate our findings neither contradicted theirs nor missed points raised by them.

Chapter 6

# Conclusion

We now attempt to answer the research questions asked in chapter 1. The first research question "What decision criteria are used by software developers to decide if and how much testing effort should be done?" was answered in the qualitative expert interviews. The main drivers for writing tests we found are "insurance of quality", "dealing with complexity", "future maintainability", "documentation of assumptions", "passive knowledge transfer", "fun" and "confidence in solutions". We grouped the first three factors in the category "quality related" and the remaining four in the category "non-quality related". Even though the interviewed developers put roughly equal weight on these two categories, we later showed that the existing literature on TDD is almost exclusively concerned with "quality related" aspects. The main factors that lead to not writing tests are identified as "external dependencies", "configuration", "inadequate existing testing suites" and "shortcuts". An interesting finding we made during the interviews was the insight that experts in software development either experience cognitive dissonance or an social acceptability problem when asked about why they omitted testing. On the one hand they are convinced that good developers write tests for their code, while on the other hand, they did not write tests for the problem at hand. This leads to defensive answers instead of revealing their compelling reasons for this. It is unclear whether the cognitive dissonance or the social acceptability bias are the main drivers for the defensive answers, but we suspect that both factors are at play here.
The next research question was "What threats to validity apply to TDD research?" And how are those related to the three contradictions of TDD?" and we attempted to answer it by compiling a list of categories of threats to validity that are specific to the field of TDD through the lens of a decision maker. The main threats we identified were "participants selection", "task selection", "context", "threats to validity regarding quality", "length of observation", "iteration", "comparisons", "TDD on a spectrum", "lack of qualitative research / narrow focus" and "inclusion of TDD in company policies". We interpreted the data in a way that suggests that TDD is more than just a software development technique but also a management tool, by structuring and facilitating the collaboration between developers. Additionally, we argued that in order to avoid and explain the three contradictions of TDD research we need to incorporate the "non-quality related" aspects of TDD.
The third research question was "How could future research be designed in order to facilitate decisions about when to include TDD into company policies?". We suggested that future research

needs to be aware of the threats to validity listed here and incorporate this knowledge into further studies. For example, to evaluate the consequences of long term application of TDD on the real life maintainability of code, there is the need for more long term studies in industrial contexts. Additionally, in order to make TDD useful and applicable for decision makers we need to look past the existing research and complement it by doing qualitative or interdisciplinary research, by including researchers from fields such as management and process experts or psychologists specialised in work and organisation.

In summary we were not able to conclusively answer all research questions which were initially proposed but hope that this work provides a starting point and additional inspiration for thinking about solving the contradictions of TDD research.

Appendix A

# Questionnaire for bugs with associated tests

Please look at the list of bugs provided and familiarize yourself with them. Choose the one that you thought to be the most interesting and answer the following questions.

## Bug Description

- Please provide a short description of the bug.
- How was the bug discovered?
- Why did it happen? What was the cause of error?

## Bug Scope

- What was the impact of the bug for the organisation (i.e. number of affected users, urgency of the fix, impediments for productive workflows)?
- What was the scope of the bug in the code/repository (i.e. how much code needed to change, impacts to architecture)? How did you know?

## Existing Test Description

- Did you write tests during the solution of the bug?
- Have there been tests before?
- Did you look at the existing tests before you started on the new ones?
- What did you gain from the existing tests?
- Why did the existing tests not catch the bug?
- Did you consider any code metrics (i.e. Sonar Cube/Lint) during the fix? Or other tools?

## Reasoning for new tests

- Please explain your reasoning behind the tests you wrote.
- Please provide a short description of what you tested.
- How did you test it?
- When did you write the tests?
- Why at this time?
- What benefit do you expect from the tests written?
- Please explain your criteria to decide whether or not more tests are needed? How did you decide that you do not need further testing?
- Do you always use the same criteria?

## Difficulty/Obstacles

- What obstacles needed to be overcome when writing these tests?

## Individual perception of testing

- How do you judge the importance of testing for your team?
- Do you like writing tests?

## Further questions

- Reflecting on your answers, are there any consequences you would suggest to the team? Are there lessons learned you could extract from the bug in question? What would be your suggestion to your teammates when they encounter a similar problem?
- Do you have feedback on the questionnaire, or anything to add? Missing questions etc.?

Appendix B

# Questionnaire for bugs without associated tests

Please look at the list of bugs provided and familiarize yourself with them. Choose the one that you thought to be the most interesting and answer the following questions.

## Bug Description

- Please provide a short description of the bug.
- How was the bug discovered?
- Why did it happen? What was the cause of error?

## Bug Scope

- What was the impact of the bug for the organisation (i.e. number of affected users, urgency of the fix, impediments for productive workflows)?
- What was the scope of the bug in the code/repository (i.e. how much code needed to change, impacts to architecture)? How did you know?

## Existing Test Description

- Did you write tests during the solution of the bug?
- Have there been tests before?
- If there have been tests:

  Did you look at the existing tests before you started on the new ones?

  What did you gain from the existing tests?

  What was your reasoning not to add to these tests?

- If there have not been tests:

  Would you have appreciated existing tests?

  Do you think existing tests would have caught the bug?

## Reasoning for no new tests

- Why did you not write new tests during the bug fix? What were your criteria for the decision?
- Do you think it would be possible to write tests for the bug? Please explain your reasoning.
- Do you think it would be reasonable to invest into tests for the bug? Why?
- If you would write tests for the bugs what would be the costs and/or benefits?
- Are there additional additional resources that could have supported you in writing tests (tools, strategies, communication, documentation)?
- What benefit do you expect from the tests written?
- Please explain your criteria to decide whether or not more tests are needed? How did you decide that you do not need further testing?
- Do you always use the same criteria?

## Further questions

- Reflecting on your answers, are there any consequences you would suggest to the team? Are there lessons learned you could extract from the bug in question? What would be your suggestion to your teammates when they encounter a similar problem?
- Do you have feedback on the questionnaire, or anything to add? Missing questions etc.?

# Appendix C

# Anleitung zum wissenschaftlichen Arbeiten: A real life example of TDD

In this chapter we give an introduction on how to apply test driven development in a realistic software development project. A lot of existing examples focus on the development of algorithms with exclusion of external dependencies. One famous example presented by Beck (2002) is concerned with currency calculations and conversions, which is purely focused on the algorithms behind it and not so much on the wider context of the application in which it is used. For this reason we want to present a short real life project and how a developer can apply TDD when working with frameworks and external dependencies.

Our goal is to develop a web service that serves data from a database. We implement this project in Java with the Spring Boot framework. Our testing framework will be JUnit in conjunction with the Mockito mocking library. In table C.1 we present the dependencies and their versions used in this project. Since we want to show how TDD is used, we assume that the reader has a certain proficiency with the above mentioned technologies. The code can be checked out from the git repository at Gross (2020). We are going to implement the project in several iterations, which can be accessed in different branches in the repository.

**Table C.1:** Used technologies

| Dependency | Version |
|---|---|
| Java | 1.8 |
| Maven | 3.6.1 |
| Spring Boot | 2.2.4 |
| JUnit | 5.5.2 |
| Mockito | 3.1.0 |
| AssertJ | 5.5.2 |
| H2 Database | 1.4.200 |

## C.1 Application of TDD

Test driven development (TDD) is a software development technique where tests are written before the code (Beck, 2002; Gorman, 2016). Before we start writing any code we have to split the requirement we want to implement into small chunks of work which we call tasks. Each of these tasks can then be implemented using TDD. TDD itself is broken up into three stages as shown in figure C.1. In the first stage called the "Red Phase", the developer writes tests which document how the code that implements the task at hand is going to function. Then all the tests are run, and the newly added tests must fail, since no code has been written yet. At the same time the project must compile for the tests to be run in the first place, therefore stubs of newly needed classes and methods are added to the project. In this phase no changes to the code itself can occur, only the tests can be changed. Then the second stage, called "Green Phase", starts. Here the developer adds the minimum amount of code for the tests to pass. At the end of this phase the developer runs all tests again to make sure the newly added code did not break any existing functionality and solves the task being worked on. Finally, in the "Refactoring" phase, the code is improved and changed. The goal of this phase is to keep the code readable and maintainable for example by removing duplication or by making sure that the code adheres to the architecture of the project. The Importance of the "Refactoring" stage is often underestimated (Thomson et al., 2009). Even though we implemented the desired functionality at this point, without refactoring, the code will degrade over time, increasing the effort necessary to maintain it and add new features. We then repeat this process until the user story is complete.

## C.2 Context

The project we are going to implement simulates a real world project that is supposed to provide data from a database via a web service. This project is done in the context of a fictitious company that stores the relationships between its departments and different types of persons in a database. This means every person has one or more roles in different departments. These roles have a validity and a type. Types describe different associations between departments and persons, like employees, external contractors or suppliers. The schema of the database is given in figure C.2. In this project we are going to simulate the database with an H2 in memory database (h2database.com, 2020), which can easily be replaced with a persistent database, as would be the case in a real company. We as developers would get a new requirement from somebody at the company. In this case we get this user story from an HR administrator:

> *As an HR administrator I want a web service that provides the employment details for a given employee identified by their GUID, so that I can check when the employee was part of which department.*

We first set up a new project with the Spring Boot Initializr (https://start.spring.io/) including the following dependencies: spring-boot-starter-data-jpa for the data access, spring-boot-starter-web for the web service and spring-boot-starter-test for the testing dependencies. We also add a "Hello, World!" web service and the data base set up script. This makes up our initial status of the

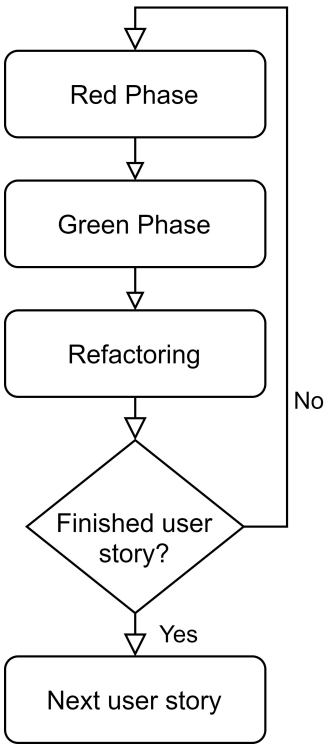


**Figure C.1:** TDD flow chart

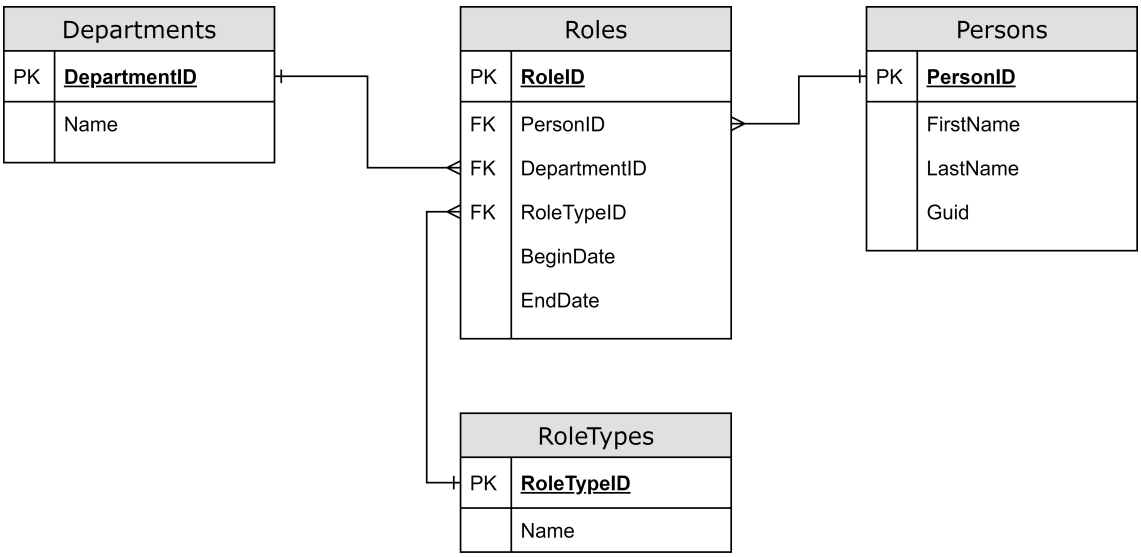


**Figure C.2:** ER diagram of the database

project which can be found in the branch “initial” in the git repository (Gross, 2020). Next this user story will be split into several sub-tasks.

## C.3 Splitting the user story

Now we break the user story apart into several sub-tasks which can be implemented during one TDD iteration (see column “Goal of the iteration” in table C.2). In the context of this tutorial we split the user story according to several techniques we want to show and give the reader an example of (see column “Applied testing technique” in table C.2). In practice however there exists no hard and fast rule on how to split the user story into sub-tasks. How to split a requirement into workable tasks is largely a skill that develops over time. In our case the requirements are very clear and the problem is well understood. In practice it makes sense to try to identify sub-tasks that are as granular as possible, since smaller tasks are easier to understand and implement. Another thing to note is that we decided to implement the database access before the API. We argue that defining the API before the underlying code requires additional abstractions and is therefore more difficult for beginners. The benefit of developing the API first, is that the users can give feedback early to determine that the API fulfills their requirements.

**Table C.2:** Definition of the goals of each iteration and the teaching goals for each iteration

| Goal of the iteration | Applied testing technique |
|---|---|
| Implementation of the database access | Integration testing (inclusion of external resources) |
| Implementation of the API | Mocking (testing in isolation) |
| Implementation of the web service functionality | Testing with framework support |

## C.4 Implementation

In this section we present the implementation of the project. In each subsection we present the different phases from figure C.1. We will not focus on the implementation code for each iteration but instead focus on the tests written.

### C.4.1 Iteration 1: Implementation of the database access

In the first iteration the database access is implemented. We start with the red phase (see figure C.1. The code can be found in Gross (2020) in branch “iteration1-redphase”). The goal of this iteration is to provide functioning data access. That means we want tests that fail when we implement the requirements in a wrong way but also when the data access is impeded. For example when the database schema changes we want to make sure that our tests fail, if it impacts the correctness of our code. Therefore we call the technique presented here “integration testing”, because we test the integration of our code with external resources, in this case the database.

From the user story we know that we need to provide "the employment details for a given employee identified by their GUID". This requirement is easily turned into a test as shown in code listing C.1 for the positive case that there is a person with a relationship stored in the database. The negative case, where no relationship for this person was found, can also be easily be transformed into a test case (see code listing C.2). The format of these tests is the "given-when-then" format. This format describes the test cases in this way: *Given* certain prerequisites, *when* we do something, *then* we expect some conditions to hold. We show this format with the comments: `//GIVEN`, `//WHEN` and `//THEN`. At this point the IDE will inform us that none of the classes were found. In order to run our tests we must fix this issue, so that the code can compile. Many IDEs support the developers here by giving options to create the missing classes. We will create the class `EmploymentDTO` and the class `DbService` with the method `public List<EmploymentDTO> getEmployment(String testguid).` Note that these classes and methods are only stubs which at this point will not have any functionality. To compile the code in order to run the tests the methods must return something, so we make them simply return `null`. In the individual tests we omitted the `//GIVEN` part of the tests, because the prerequisites are handled in the test set up. In our case we want to insert data into the database before running the tests and clean it up afterwards. We use the spring boot annotation `@Sql(scripts = "/testdata.sql")` to achieve this, by providing an SQL script for the framework to handle the database set up and clean up. Details about this technique can be found in Webb et al. (2020).

**Listing C.1:** Test case: found person data in database

```
@Test
public void findsEmploymentdataIfExistsTest() {
  //WHEN
  List<EmploymentDTO> employmentData =
      service.getEmployment("testguid");
  //THEN
  assertThat(employmentData).hasSize(1);
  assertThat(employmentData.get(0)).hasFieldOrPropertyWithValue("firstName",
      "Test-Firstname");
}
```

**Listing C.2:** Test case: no person data found in database

```
@Test
public void noEmploymentdataFoundIfNotExistsTest() {
  //WHEN
  List<EmploymentDTO> employmentData = service.getEmployment("non
      existing guid");
  //THEN
  assertThat(employmentData).hasSize(0);
}
```

Next we start the "Green Phase" (found in the branch "iteration1-greenphase" in Gross (2020)). We write just enough code to make all tests pass. In the "Red Phase" we already created the classes and method stubs we need to fulfill the requirements. Now we only need to implement these stubs. In this example we choose to execute a native SQL query (given in `query.sql`) and map the results to a list of data transfer objects (DTOs; given in `EmploymentDTO.java`). The details of implementation can be found in Gross (2020), and we will not go into them, since the focus of this tutorial lies in the application of TDD and not in the implementation. During this phase we run the tests often to give us a measure of progress for the task at hand. When all tests pass, we immediately go to the next phase, since we know we are finished.
We finish this iteration with the "Refactoring phase" (see branch "iteration1-refactoring" in Gross (2020)). We have not yet thought about the structure or architecture of our code. We will apply the Boundary-Control-Entity architectural pattern (Heineman & Denham, 2009) to our code, because it seems to be a good fit for what we want to achieve. In our case the `EmploymentDTO.java` serves as the entity and `DbSerivce.java` as the control. In the next iteration we will add a Spring Boot RestController in the boundary layer. TDD supports us greatly in the restructuring of our code, because afterwards we only need to run our tests to verify that our changes and refactorings did not break any functionality.

### C.4.2 Iteration 2: Implementation of the API

In this iteration we want to implement the boundary layer of the Boundary-Control-Entity architectural pattern, meaning we implement the API through which potential users are going to interface with our project. While we used integration tests in iteration 1, we now use the testing technique of mocking to test the boundary layer in isolation of the rest of the code. Integration tests explicitly include the external dependencies in testing. Mocking on the other hand serves to exclude all dependencies from the unit under test. A mock object is a fake object that is created during the execution of tests that behaves in a way we define according to the contracts of the dependencies we use (Beck, 2002). How to set up a mock object is shown in code listing C.3 and how to create and define the behaviour of the mock object can be found in the test cases presented in code listings C.4 and C.5 in the `//GIVEN` part of the tests. The tests C.4 and C.5 are written in the "Green Phase" of iteration 2 (see branch "iteration2-redphase" in Gross (2020)). The first test C.4 covers the case where when a person is found, the boundary returns the data of that person. In the `//GIVEN` part of the test we create a list of `EmploymentDTO` objects and the line `doReturn(list).when(service).getEmployment(any());` describes the behaviour of the mock object. The list of `EmploymentDTO` objects is returned if the method `getEmployment` of the `DbService` is called with any parameters. This mock object is then passed to the `Controller`, the class under test. We then call the method `getEmploymentByGuid` and assert that it in fact returns a list with exactly one `EmploymentDTO`. The other test C.5 covers the case that no person is found in the `DbService`. We set up the mock object to return an empty list if the method `getEmployment` is called. We then call this method, catch the thrown exception and assert that this exception is an instance of the custom exception `NoResultFoundException.class`. It would be possible to return an empty list instead of throwing an exception. But since the requirements do not specify this behaviour, we make this design decision in the "Red Phase" of this iteration. The last step of this

TDD phase is again to create stubs for all classes that we need for our project to compile and our tests to fail.
With all design decisions and behaviour definitions out of the way, the "Green Phase" (see branch "iteration2-greenphase" in Gross (2020)) is straightforward. We implement the `Controller` to call `DbService` and depending on the result, return the result or throw an exception. While doing that we run the tests often to verify our progress. At the end of the "Green Phase" it is important that all tests of the project are run again to verify that our changes to the code did not break any existing functionalities.
Because during the "Red Phase" and the "Green Phase" we were only concerned with functionality, in the next "Refactoring phase" (see branch "iteration2-refactoring" in Gross (2020)) we now focus on structure, understandability, clean code etc. For example, we created the `NoResultFoundException.java` without thinking where it falls in the Boundary-Control-Entity pattern. Now is the time to move it to a new Java package for exceptions. Since we have not yet written much code, the "Refactoring phase" is rather short. In bigger projects this phase will have more changes to different aspects of the code.

**Listing C.3:** Initialisation of Mocks

```
@Mock DbService service;

@BeforeEach
void setUp() {
  MockitoAnnotations.initMocks(this);
}
```

**Listing C.4:** Test case: Boundary returns correct value if mocked result is found

```
@Test
void resultGiven() {
  //GIVEN
  EmploymentDTO employmentDTO =
     new EmploymentDTO(
        "testfistname",
        "testlastname",
        "testdept",
        "testroletype",
        Date.valueOf(LocalDate.now()),
        Date.valueOf(LocalDate.now()));
  List<EmploymentDTO> list = new ArrayList<>();
  list.add(employmentDTO);
  doReturn(list).when(service).getEmployment(any());

  Controller controller = new Controller(service);

  //WHEN
  List<EmploymentDTO> result =
     controller.getEmploymentByGuid("testguid");
```

```
  //THEN
  assertThat(result).hasSize(1);
}
```

**Listing C.5:** Test case: Boundary throws exception if mocked result is not found

```
@Test
void noResultFoundThrows() {
  //GIVEN
  List<EmploymentDTO> list = new ArrayList<>();
  doReturn(list).when(service).getEmployment(any());

  Controller controller = new Controller(service);

  //WHEN
  Throwable thrown =
     catchThrowable(
         () -> {
          List<EmploymentDTO> result =
              controller.getEmploymentByGuid("testguid");
         });
  //THEN
  assertThat(thrown).isInstanceOf(NoResultFoundException.class);
}
```

### C.4.3 Iteration 3: Implementation of the web service functionality

In the last iteration we implement the web service functionality. In the requirements it was stated that a web service is desired. Therefore we will now add the functionality that our service is reachable via a URL that the correct HTTP status codes are returned and that the result is delivered in the JSON format. In order to achieve this we will make use of Spring Boot web service capabilities. Spring Boot is a dependency of our project and we do not want to test it, because our assumption is that Spring Boot itself is thoroughly tested and functioning as documented. This assumption can be verified by looking up the test suite of Spring Boot, which can be found in an open source repository. Instead, we want to verify that we use its features correctly in form of a test. By necessity this test will be an integration test since we need to include the Spring Boot dependency in our tests. Before we can start writing tests, we need to decide whether or not to include other external dependencies (the database in our case) in the tests for this task. On the one hand, one could argue that each test should have exactly one potential point of failure, making troubleshooting the tests straightforward and efficient. On the other hand personal experience shows that developers, especially early in their career, appreciate the insurance provided by tests which simulate a user request against the actual code including all dependencies. In the context of continuous integration, before a new version can be deployed, all tests are run. And tests which verify that the user will actually get the data they need, provide the

developer with reassurance that no existing features are broken. Therefore we want to present this technique in this tutorial. Later in section C.6 we will provide an exercise in which tests are added that only verify that Spring Boot is configured correctly without the dependency on the database.

For the "Red Phase" of iteration 3 (see branch "iteration3-redphase" in Gross (2020)) we use the `TestRestTemplate` provided by Spring Boot, which needs to be set up with the annotations `@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)` and `@LocalServerPort` to get the random port under which the application is run locally. The `TestRestTemplate` is used to make webservice calls in our tests and we call it `restTemplate` (see code listings C.6 and C.7). The first test verifies that if we call the service with an existing GUID we get a JSON which is shown in code listing C.6. In practice the text based data exchange format is often defined by a schema provided by the user or defined by the developer, for example JSON Schema for JSON or XSD for XML. In this case we could also validate the answer against said schema. The next test C.7 validates that if no person is found, the web service returns the HTTP status code 404 Not Found.

This time the implementation of the functionality described by the tests during the "Green Phase" (see branch "iteration3-greenphase" in Gross (2020)) consists of only adding two annotations: the method in the `Controller` is annotated with `@GetMapping(value = "/employmentbyguid", produces = MediaType.APPLICATION_JSON_VALUE)` and the exception is annotated with `@ResponseStatus(code = HttpStatus.NOT_FOUND, reason = "Not Found")`. The documentation for these annotation can be found in Webb et al. (2020).

Since we only added two lines in the "Green Phase" it is difficult to find anything to refactor in the "Refactoring phase". That is why we omitted this phase (Gross, 2020). In practice it is almost never the case that there is nothing to refactor after changes to the code have been made.

**Listing C.6:** Test case: Framework is configured correctly: result is delivered in JSON

```
@Test
public void getEmploymentByGuidIntegrationTest() throws Exception {
  //WHEN
  String url = "http://localhost:" + port +
      "/employmentbyguid?guid=cbtestguid";
  ResponseEntity result = restTemplate.getForEntity(url,
      String.class);
  //THEN
  assertThat(result.getStatusCode()).isEqualTo(HttpStatus.OK);
  assertThat(result.getBody())
      .isEqualTo(
         "[{\"firstName\":\"Camillo\"," +
            "\"lastName\":\"Berneri\"," +
            "\"departmentName\":\"God Dept\"," +
            "\"roleTypeName\":\"Employee\"," +
            "\"beginDate\":\"2010-10-10\"," +
            "\"endDate\":\"2015-10-17\"}," +
```

```
            "{\"firstName\":\"Camillo\"," +
            "\"lastName\":\"Berneri\"," +
            "\"departmentName\":\"God Dept\"," +
            "\"roleTypeName\":\"Lecturer\"," +
            "\"beginDate\":\"2008-08-17\"," +
            "\"endDate\":\"2020-12-01\"}]");
}
```

**Listing C.7:** Test case: Framework is configured correctly: no person found leads to 404 Not Found

```
@Test
public void getEmploymentByGuidExceptionTest() throws Exception {
  //WHEN
  String url = "http://localhost:" + port +
      "/employmentbyguid?guid=notexistingguid";
  ResponseEntity result = restTemplate.getForEntity(url,
      String.class);
  //THEN
  assertThat(result.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
  assertThat(result.toString()).contains("Not Found");
}
```

## C.5 Conclusion

In this tutorial we developed a small web service by applying TDD and all its phases. We also gave examples of testing techniques, like integration testing, mocking and testing of and with framework support. By doing so we addressed the problem that a lot of existing TDD tutorials only focus on the development of algorithms without regard to external dependencies, which play a major role in real-world software projects.
Additionally we showed how to approach requirement specifications with a technique called "divide and conquer", namely breaking up the requirements into small workable sub-tasks. This benefits the developer by reducing the cognitive load during implementation. We also highlighted the importance of the "Refactoring phase" of TDD, since it is often neglected in practice (Thomson et al., 2009). The benefit of not skipping the "Refactoring phase" is the decoupling of producing clean and maintainable code from the implementation of functionality ind the "Red Phase" and the "Green Phase". This again helps the developer focus on only one aspect of code while developing.

## C.6 Further exercises

We want to finish this section by giving a couple of suggestions of possible extensions to this project. In these exercises developers can use TDD themselves and improve their skill in it. In section C.4.3 we already discussed whether or not to include the database in the web functionality tests. We opted to include it. Now it would be a good exercise to write a test that verifies that the

web service responses are correct while excluding the data layer from the tests.
For even further exercises we now present two additional user stories. The first user story aims at providing an opportunity to repeat what we learned so far:

> *As an HR administrator I want a web service that provides all employees for a given department identified by the department ID, so that I can check who works currently for this department.*

Now that we have showed how to retrieve data from a database with a web service, the next logical step is to also change the data in a database:

> *As an HR administrator I want a web service that accepts new employment data for existing persons, so that I can post new employment roles.*

# List of Tables

# References


Bannerman, S., & Martin, A. (2011). A multiple comparative study of test-with development product changes and their effects on team speed and product quality. *Empirical Software Engineering*, *16*, 177–210. doi: 10.1007/s10664-010-9137-5

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Beck, K. (2001). Aim, fire [test-first coding]. *IEEE Software*, *18*(5), 87-89. doi: 10.1109/52.951502

Beck, K. (2002). *Test-Driven Development By Example*. Amsterdam: Addison-Wesley Longman.

Begel, A., & Zimmermann, T. (2013). Analyze This! 145 Questions for Data Scientists in Software Engineering Data Scientists in Software Engineering. *Microsoft Research. Technical Report*, 1–13.

Beller, M., Gousios, G., Panichella, A., Proksch, S., Amann, S., & Zaidman, A. (2019). Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, *45*(3), 261-284. doi: 10.1109/TSE.2017.2776152

Bissi, W., Neto, A., & Emer, M. (2016). The effects of test driven development on internal quality , external quality and productivity : A systematic review. *Information and Software Technology*, *74*, 45-54. doi: 10.1016/j.infsof.2016.02.004

Borle, N., Feghhi, M., Stroulia, E., Greiner, R., & Hindle, A. (2018). Analyzing the effects of test driven development in GitHub. *Empirical Software Engineering*, *23*(4), 1931–1958.

Buchan, J., Li, L., & Macdonell, S. G. (2011). Causal Factors , Benefits and Challenges of Test-Driven Development : Practitioner Perceptions. *2011 18th Asia-Pacific Software Engineering Conference*, 405–413. doi: 10.1109/APSEC.2011.44

Causevic, A., Sundmark, D., & Punnekkat, S. (2011). Factors limiting industrial adoption of test driven development: A systematic review. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011*, 337-346. doi: 10.1109/ICST.2011.19

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Commun. ACM*, *31*(11), 1268–1287. doi: 10.1145/50087.50089

Dittrich, Y., John, M., Singer, J., & Tessem, B. (2007). Editorial: For the special issue on qualitative software engineering research. *Information & Software Technology*, *49*, 531-539. doi: 10.1016/j.infsof.2007.02.009

Dogša, T., & Batic, D. (2011). The effectiveness of test-driven development: An industrial case study. *Software Quality Journal*, *19*, 643–661. doi: 10.1007/s11219-011-9130-2

Erdogmus, H., Shull, F., Turhan, B., Layman, L., Melnik, G., & Diep, M. (2010). What do we know about test-driven development? *IEEE Software*, *27*(06), 16-19. doi: 10.1109/MS.2010.152

Flick, U. (2009). *An Introduction to Qualitative Research*. SAGE Publications.

Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., & Juristo, N. (2017). A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering*, *43*(7), 597–614.

Fucci, D., Romano, S., Baldassarre, M., Caivano, D., Scanniello, G., Turhan, B., & Juristo, N. (2018). A Longitudinal Cohort Study on the Retainment of Test-Driven Development. *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018*. doi: 10.1145/3239235.3240502

Fucci, D., & Turhan, B. (2013). A replicated experiment on the effectiveness of test-first development. In *2013 ACM / IEEE international symposium on empirical software engineering and measurement, baltimore, maryland, usa, october 10-11, 2013* (pp. 103–112). IEEE. doi: 10.1109/ESEM.2013.15

Gorman, J. (2016). *TDD*. Codemanship Limited.

Gross, T. (2020). *Tdd tutorial*. GitHub. (`https://github.com/tgross12/tdd-tutorial`, last accessed on 01.02.2020)

h2database.com. (2020). *The H2 database* (Tech. Rep.). (`http://www.h2database.com/html/main.html`, last accessed on 14.02.2020)

Heineman, G., & Denham, J. (2009). Entity, boundary, control as modularity force multiplier. In P. Greenwood et al. (Eds.), *Proceedings of the 24th acm sigplan conference companion on object oriented programming systems languages and applications*. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1639950.1639979

Höst, M., Regnell, B., & Wohlin, C. (2000). Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, *5*(3), 201–214. doi: 10.1023/A:1026586415054

Karac, E. I., Turhan, B., & Juristo, N. (2019). A controlled experiment with novice developers on the impact of task description granularity on software quality in test-driven development. *IEEE Transactions on Software Engineering*, 1-1.

Karac, I., & Turhan, B. (2018). What Do We ( Really ) Know about Test-Driven Development ? *IEEE Software*, *35*, 81–85. doi: 10.1109/MS.2018.2801554

Kazerouni, A. M., Shaffer, C. A., Edwards, S. H., & Servant, F. (2019). Assessing Incremental Testing Practices and Their Impact on Project Outcomes. In *Proceedings of the 50th acm technical symposium on computer science education* (pp. 407–413). New York, NY, USA: ACM. doi: 10.1145/3287324.3287366

Kollanus, S. (2010). Test-Driven Development - Still a Promising Approach? In *2010 seventh international conference on the quality of information and communications technology* (pp. 403–408). IEEE. doi: 10.1109/QUATIC.2010.73

Munir, H., Moayyed, M., & Petersen, K. (2014). Considering rigor and relevance when evaluating test driven development: A systematic review. *Information and Software Technology*, *56*(4), 375–394. doi: 10.1016/j.infsof.2014.01.002

Pančur, M., & Ciglaric, M. (2011). Impact of test-driven development on productivity, code and

tests: A controlled experiment. *Information and Software Technology*, *53*(6), 557–573. doi: 10.1016/j.infsof.2011.02.002

Pedroso, B., Jacobi, R., & Pimenta, M. (2010). Tdd effects: Are we measuring the right things? In A. Sillitti, A. Martin, X. Wang, & E. Whitworth (Eds.), *Agile processes in software engineering and extreme programming* (pp. 393–394). Berlin, Heidelberg: Springer Berlin Heidelberg.

Porter, A., & Votta, L. (1995). Comparing Detection Methods For Software Requirements Inspections: A Replication Using Professional Subjects. *IEEE Transactions on Software Engineering*, *21*, 563–575.

Rafique, Y., & Misic, V. (2013). The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, *39*(6), 835–856. doi: 10.1109/TSE.2012.28

Romano, S., Fucci, D., Scanniello, G., Turhan, B., & Juristo, N. (2017). Findings from a multi-method study on test-driven development. *Information and Software Technology*, *89*, 64–77. doi: 10.1016/j.infsof.2017.03.010

Runeson, P. (2006). A Survey of Unit Testing Practices. *IEEE SOFTWARE*, *23*(4), 22–29. doi: 10.1109/MS.2006.91

Santos, A., Spisak, J., Oivo, M., & Juristo, N. (2018). Improving development practices through experimentation: An industrial TDD case. In *25th asia-pacific software engineering conference, APSEC 2018, nara, japan, december 4-7, 2018* (pp. 465–473). doi: 10.1109/APSEC.2018.00061

Scanniello, G., Romano, S., Fucci, D., Turhan, B., & Juristo, N. (2016). Students' and professionals' perceptions of test-driven development: A focus group study. In *Proceedings of the 31st annual acm symposium on applied computing* (pp. 1422–1427). New York, NY, USA: ACM. doi: 10.1145/2851613.2851778

Siniaalto, M. (2006). Test driven development: empirical body of evidence. *Agile Software Development of Embedded Systems*.

Stripe.com. (2018). *The Developer Coefficient* (Tech. Rep.). (`https://stripe.com/files/reports/the-developer-coefficient.pdf`, last accessed on 15.12.2019)

Thomson, C. D., Holcombe, M., & Simons, A. J. H. (2009). What Makes Testing Work : Nine Case Studies of Software Development Teams. In *2009 Testing: Academic and Industrial Conference - Practice and Research Techniques* (p. 167-175). doi: 10.1109/TAICPART.2009.12

Torchiano, M., & Sillitti, A. (2009). TDD = too dumb developers? Implications of Test-Driven Development on maintainability and comprehension of software. In *2009 IEEE 17th International Conference on Program Comprehension* (pp. 280–282).

Tosun, A., Ahmed, M., Turhan, B., & Juristo, N. (2018). On the Effectiveness of Unit Tests in Test-driven Development. In *Proceedings of the 2018 International Conference on Software and System Process* (pp. 113–122). New York, NY, USA: ACM. doi: 10.1145/3202710.3203153

Tosun, A., Dieste, O., Vegas, S., Pfahl, D., Rungi, K., & Juristo, N. (2019). Investigating the impact of development task on external quality in test-driven development: An industry

experiment. *IEEE Transactions on Software Engineering*, 1-1.

Turhan, B., Layman, L., Diep, M., Erdogmus, H., & Shull, F. (2010). How Effective is Test-Driven Development? In *Making Software* (p. 624). O'Reilly Media.

Webb, P., Syer, D., Long, J., Nicoll, S., Winch, R., Wilkinson, A., ... Bhave, M. (2020). *Spring boot reference documentation.* Spring Boot. (`https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/`, last accessed on 01.02.2020)

Zubac, J., Alpha, F. S., Lindskog, C., & Gagner, I. (2018). How Does Test-Driven Development Affect the Quality of Developed Software?

# Acknowledgments

I want to thank my thesis advisor Prof. Dr. Oscar Nierstrasz and Dr. Mohammad Ghafari. Additionally, I want to thank Vera Bamert, Christine Perreng and Johannes Gross for their incredible feedback and proof reading.

# Declaration of Originality

**Last name, first name: Timm Gross**

**Matriculation number: 07-115-421**

I hereby declare that this thesis represents my original work and that I have used no other sources except as noted by citations.
All data, tables, figures and text citations which have been reproduced from any other source, including the internet, have been explicitly acknowledged as such.
I am aware that in case of non-compliance, the Senate is entitled to withdraw the bachelor degree awarded to me on the basis of the present thesis, in accordance with the “Statut der Universität Bern (Universitätsstatut; UniSt)", Art. 69, of 7 June 2011.

Bern, December 15, 2019

Timm Gross