Enablers, Inhibitors, and Perceptions of Testing in Novice Software Teams

Raphael Pham*, Stephan Kiesling*, Olga Liskin*, Leif Singer†, and Kurt Schneider*
*Leibniz Universität Hannover
Hannover, Germany
{Raphael.Pham, Stephan.Kiesling, Olga.Liskin, Kurt.Schneider}@inf.uni-hannover.de
†University of Victoria
Victoria, BC, Canada
lsinger@uvic.ca

ABSTRACT

There are many different approaches to testing software, with different benefits for software quality and the development process. Yet, it is not well understood what developers struggle with when getting started with testing—and why some do not test at all or not as much as would be good for their project. This missing understanding keeps us from improving processes and tools to help novices adopt proper testing practices.

We conducted a qualitative study with 97 computer science students. Through interviews, we explored their experiences and attitudes regarding testing in a collaborative software project. We found enabling and inhibiting factors for testing activities, the different testing strategies they used, and novices' perceptions and attitudes of testing. Students push test automation to the end of the project, thus robbing themselves from the advantages of having a test suite *during* implementation. Students were not convinced of the return of investment in automated tests and opted for laborious manual tests—which they often regretted in the end. Understanding such challenges and opportunities novices face when confronted with adopting testing can help us improve testing processes, company policies, and tools. Our findings provide recommendations that can enable organizations to facilitate the adoption of testing practices by their members.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Management, Programming teams

General Terms

Human Factors

Keywords

Testing, Adoption, Motivation, Enablers, Inhibitors.

FSE '14, November 16–22, 2014, Hong Kong, China

1. INTRODUCTION

Automated testing has many proven benefits. Test-Driven Development (TDD) [1] can help shape software design [25]. Unit testing is a prerequisite for refactoring [7] and regression tests keep defects from reappearing.

However, adoption of testing practices can be problematic. In a previous study [18], we encountered several commercial as well as open source projects in which testing was either an afterthought or not used at all. Additionally, we have experienced similar situations in industry projects and with students: when resources become constrained, testing is sometimes the first activity to be discontinued.

Our aim is to better understand this special role of testing relative to other development practices. Insights into this relationship should help us design better policies, processes, and tools around testing, which should improve overall software quality. The study presented here is one step towards this larger goal.

In our qualitative study, we focus on teams of novice software developers about to finish their university studies. Most of them will soon begin a career as software developer in industry. Thus, the insights provided by our study can be useful for organizations that employ novice software developers and want to improve how they train and leverage them. By understanding why novices do or do not adopt testing practices, process managers can actively remove barriers and facilitate desired testing behavior.

Researchers can use our contributions to guide the creation of new tools and practices that support novices in adopting good testing practices. On the one hand, we uncover technical challenges for which better tool support is needed. On the other hand, research into policies in software development can benefit from our findings regarding attitudes towards testing.

For educators, our study documents testing-related preconceptions that they might encounter in their own students. We also found a set of weaknesses in current education that we believe if addressed would help prepare students much better for careers as software developers and managers.

This paper is structured as follows: the succeeding section discusses related work. We then describe the software project that we conducted our study in, and provide details about our study design. Section 5 documents our findings, which we then discuss in section 6, along with our study's limitations and threats to its validity. Section 7 concludes the paper.

2. RELATED WORK

We now discuss related work, with a focus on empirical studies that have explored testing behavior in software engineering teams.

2.1 Factors that Influence Testing

Kanij, Merkel, and Grundy investigated which factors affect effectiveness of testing. In a questionnaire-based survey [11], they inquired about the importance of factors such as tools and training, but also more human-centered factors like personality characteristics and experience. They find that, among others, good domain knowledge, experience, and interpersonal skills were considered important. In another empirical study [10] they investigated whether the personality of testers influences how effectively they test. They found several weak correlations between personality characteristics and testing effectiveness. Karhu et al. [12] reported on factors that are important for specifically supporting test automation.

In our interview-based study, we also explore perceptions and opinions, but in addition to that probe into concrete experiences. Further, we add an additional perspective by studying developers who are mostly new to testing.

Rooksby et al. [20] explored how *social and organizational aspects* influence testing. Beer and Ramler [2] investigated the influence of *experience* on effective testing. They further describe the sources and perceived value of *experience* for testing.

Ellims et al. [5] argue that unit testing is often criticized for its *high perceived costs*. They investigated the relations between testing costs and benefits in three case studies. The authors found that the perceived costs were probably exaggerated compared to the benefits of testing activities.

Furthermore, *feedback* can influence the motivation to adopt practices. Kanij et al. explored feedback mechanisms in the testing domain [9]. They report on a lack of methods and research for assessing testing performance. In a literature review, they found several factors and assessed their importance as perceived by practitioners.

All these factors can influence testing behavior. In our study, we explore the factors that were most prominent to the participants by asking about their opinions and experiences.

2.2 Adoption of Software Testing Practices

Ng et al. [16] explored software testing practices in Australian industry. They inquired about currently used testing methods, tools, and metrics, and listed several barriers to their adoption.

Causevic et al. [4] focused on Test-Driven Development (TDD). In a systematic literature review, they identified several factors that limit the adoption of TDD in industry. Limiting factors include higher development time and insufficient experience, but also more indirect problems like domain-specific issues and a lack of upfront design. Benton [3] describes decision categories and strategies that need to be addressed in order to establish a software test organization.

In our study, we likewise find inhibitors of testing adoption. In addition to that, we also report on experiences that positively influence the adoption of testing. We further differentiate between practices for automated and manual testing.

2.3 Perception and State of the Art of Testing

Runeson [21] conducted a questionnaire-based survey on unit testing in industry. He reports on current practices, benefits, and problems of unit testing. He also showed that there are different perceptions of what constitutes a unit test. In another qualitative survey in industry, Engström and Runeson [6] investigated regression testing in order to help understand it better. Zaidman et al. [27] considered testing in more detail through mining repositories. They examined whether repository mining can help find out how test code co-evolves with production code and present their results for three case studies.

Vegas and Basili [26] present a characterization schema that illustrates which testing techniques are suitable for a given project. They note that collecting sufficient information—which is often distributed among different sources—to choose appropriate testing techniques is a major problem for software developers.

LaToza et al. [13] report on developer work habits in general, and also elaborate on unit testing. Although the developers in their study reported to spend about half of their time on fixing defects, unit testing was one of the activities the developers spent the least time on.

Viewing testing through the eyes of a novice—as conveyed in our study—adds further insights into testing strategies and challenges for developers. In addition to this, we also highlight less tangible aspects, like attitudes towards and preconceptions of testing.

3. THE SOFTWARE PROJECT COURSE

Since 2004, the Software Engineering Group at Leibniz Universität Hannover organizes an annual software project course ("SWP") for third-year computer science undergraduates. All participating students are divided into teams of five students per team. In fall 2013, 97 students were divided into 20 teams: 18 teams of five students each, one team of four students, and one team of three students. These 20 teams were assigned to work with various technologies. Six teams created a Java desktop application and five teams implemented an Android app. Seven teams created a Web application and two teams created an EMF/Eclipse-based application.

In the course, every team elects a project leader and a quality agent from its members. The project itself lasts 4 months, in which the team needs to elicit requirements, design, implement, and test a software product. At the end of the project, the customer conducts an acceptance test and either accepts or rejects the product.

Customers are members of our group, offering projects they are interested in. Therefore, some teams had the same task assigned in parallel, but had to work independently of each other, sometimes with slightly modified requirements. Our group tries to conduct software projects that are as realistic as possible, while controlling complexity and size of the projects. This is essential for being able to compare them. In addition to customers, our group also provides each team with a coach for technical and process-related help.

The projects follow a waterfall-like process, with each phase ending in a quality gate. These act as checkpoints and ensure some basic qualities of the products—such as adherence to coding standards, a minimum amount of unit tests, or the existence of use cases in a requirements specification.

If a team fails a quality gate, they can rework their artifacts and attempt the quality gate again. Failing the same quality gate twice will abort the project, however this has not happened yet.

To participate in the software project course, we require students to have passed a programming course (Java) as well as basic courses on software engineering, software quality, and version control. The latter are all taught by our group as well.

At the end of the course—when students can be sure that they have either passed or failed—we conduct interviews with all members of a team simultaneously using the *LID* technique [22]. This is a moderated post-mortem meeting where all team members recall events of the past project. This allows us to capture students' perceptions of the course while they are still vivid, but at the same time without any fear of failing the course.

4. STUDY DESIGN

With the 97 participants of the fall 2013 software project course as participants, we conducted a Grounded Theory study about the testing behavior of novice developers.

We interviewed all 97 students at the end of the course—right after the LID session—and recorded the interviews. The interviews lasted 20 minutes on average. As suggested by Hoda et al. [8], we used an iterative data analysis approach to extract our theory of our students' testing behavior. We report our findings in section 5.

4.1 Research Questions

In our study, we tried to understand the testing behavior of our students in the software project course, in which we try to approximate or simulate an environment that is similar to a software project in industry. We were interested in supporting factors that helped students to methodically test their product and in inhibiting factors that kept them from doing so. In this context, methodical testing describes testing effort that encompasses preparation, guided execution and some form of documentation. For example, carefully thinking about input values and correct output values, deliberately executing the application and noting the results differs from a quick exploratory execution. As is often sensible Grounded Theory, we did not finalize our research questions before starting data collection. However, we had topics of interest in mind. The following are the research questions we arrived at after iterative qualitative analysis using open coding, selective coding, axial coding, and writing memos.

Testing Strategies
RQ1: How did students test their software products?

Enabling and Inhibiting Factors
RQ2: What factors supported students in testing methodically and what factors hindered them?

Testing Attitude
RQ3: What did students think of testing methodically?

Testing in the SWP process
RQ4: How did students incorporate testing in their engineering process?

4.2 Semi-Structured Interviews

The interviews were conducted at the end of the software project course. At that point, students had finished their projects and had already passed or failed the course. Each interview lasted between 20 and 30 minutes; they were conducted by two researchers. The interview was semi-structured: they were based on a script that inquired about test activities and strategies employed by the students, their reasons for choosing them, their prior experiences with development and testing, the problems they encountered and how they solved them, their understanding of automated testing and different testing techniques, and finally the lessons they learned about testing in the project. However, as interesting topics came up in the discussion, we delved into those to gain deeper insights. After each interview, we revised our interview script to de-emphasize topics that we had already saturated, and to add new interesting details to discuss in future interviews.

4.3 Data Analysis

All interviews were recorded and transcribed. Two researcher previewed the interviews thoroughly and took notes. All researchers identified about 1,500 key points in the interviews and noted them. In the next step, similar key points were aggregated and coded into one code. We constructed 308 codes in total. From these codes, different themes emerged and we grouped codes accordingly. The following themes helped us gain an overview of our data:

- Attitude towards testing and software engineering
- Motivation and amotivation regarding testing
- Team-internal organization of test efforts
- Timing of testing in the development process
- The parts of the application under test (AUT) that were tested
- Testing-related challenges
- Strategies to direct test efforts during development
- General automated testing; including strategies, motivations, problems, and benefits
- Manual and automatic GUI-Testing; including strategies, motivations, problems, and benefits
- Tools used for testing

From there, we extracted higher level concepts that connected different codes and explained phenomena that we encountered in our data. In total, we extracted 52 concepts. We grouped these 52 concepts into four categories and one core category. These can be seen in Fig. 1.

We collected quantitative data at several points, but report such numbers for illustrative purposes only.

4.4 Population

The population of 97 students was self-selected: all participants were asked to take part in the interviews, but were also informed that they could decline without consequences. 67 of 97 students were studying in their 5th semester, 10 students were enrolled in their 7th semester, and four students

had been studying for four years and longer. For the remaining students, we have no data available regarding their study progress.

Regarding the experiences of the students, we divided our population into three groups. The first group only had experiences with software development in courses and consists of 39 students. The second group of 36 students had already conducted private software projects. 22 students had worked as software developers in commercial settings and make up the third group.

Considering all the information given by the students at the beginning of the course, 94 of 97 students had passed a basic course on programming in Java. At least 76 students had passed a basic course on software quality, and 91 students had passed one on software engineering. 5 students had passed a non-mandatory course on security engineering.

5. FINDINGS

In our study, we found that the students' actual **testing behavior** is influenced by a wealth of factors, which can be seen in Fig. 1. We differentiate four categories: "**education, experiences, gaps of knowledge**" form a baseline of technical knowledge. This is also the origin of the students' "**attitudes and preconceptions**" towards testing. The students' "**perception of testing**" plays an important role in how they will approach a concrete task. Lastly, the students' testing behavior is influenced by "**external influence factors**", such as management support or resource constraints. In the following, we first explain the observed testing behavior and then move on to discuss influencing factors.

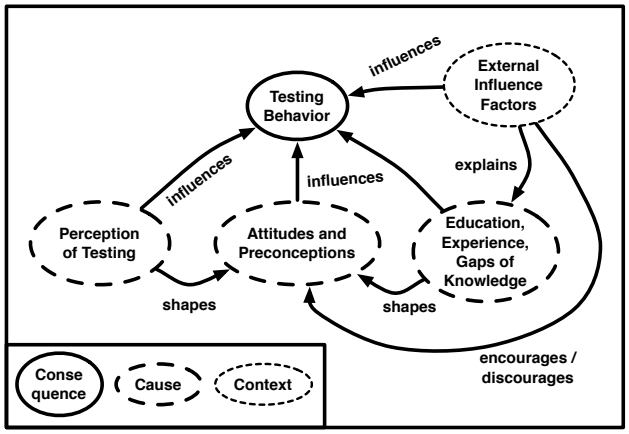

Figure 1: The categories observed in our study, forming our theory of testing behavior.

5.1 Testing Behavior

The software project course is divided into three phases: a requirements phase, a design phase, and an implementation phase.

Testing took different forms in the project. Students distinguished between *automated testing* (we denote this with A) and *manual testing*. Automated testing meant having dedicated test code for the automated execution of some part of the AUT (excluding the user interface) and some assertion. JUnit tests were the most prominent example. Manual testing was divided into different approaches: when it came to GUI testing, students mostly relied on *smoke testing* (S), even though they did not call it that. Smoke testing was the execution of the AUT to verify the correctness of the currently finished or modified code. Smoke testing was done without documentation and *solely* served the selective assertion of freshly made changes. *Manual GUI testing* (MG) differed from smoke testing in that it encompassed several steps for asserting a whole use case in the GUI and not just one step. Sometimes, these manual GUI tests were conducted in a more systematic manner using a protocol or a checklist (MG*). Lastly, some teams automated their GUI tests (AG) through a framework such as *Selenium* or Android's *Robot*.

Out of 20 teams, only four teams *discussed their testing strategy* in the first phase of the development process. Another five teams planned their testing strategy in the implementation phase and only two of those in the beginning of that phase. The remaining 11 teams did not plan testing in advance (cf. Table 1).

When it came to *team-internal organization* of testing efforts, 18 teams implicitly decided to let every author test their own code. In two cases (teams I and L), the quality agent of the team took the role of testing mentor. He managed the test process and provided the team with example tests to work on.

13 teams felt *pressured by the project deadline* into ceasing testing efforts:

> *"[We had] by far too little time to be able to write extensive tests. Then we could have tested the GUI with Selenium . . . Yes, it was mainly the time. [Other member:] Yes, I also felt time pressure."* — *team* T

The testing activity that was omitted the most frequently was the automation of GUI tests. Often, GUI tests implied a learning time that some teams were not able or willing to invest. Only 3 teams conducted automated GUI tests (cf. Table 1).

Table 1 shows all 20 teams' testing efforts in context of the development process. All teams started testing in the last phase of the project—even though some had talked about it earlier. 10 Teams managed to automate tests earlier than at the end of the implementation phase. The rest opted for manual testing and automation at the end of the project. Teams reported to have underestimated the effort required for implementation and testing became a secondary concern.

> *"I have been caught on the implementation until the end. [Other student:] Me too. I have been working on everything and then thought, testing is admittedly nice, but it's of secondary importance. First, something must be up."* — *team* J

Most of the teams that prepared testing beforehand and talked about it early managed to automate their tests *during* implementation and even produce automated GUI tests. Team G specifically decided against automated GUI tests, as they were confident to have tested their GUI sufficiently in manual runs with protocols.

The two-sided dilemma of late tests. Why do students push testing to the end of the project? They are already overwhelmed with programming, and testing becomes

Table 1: Test timetable of 20 teams. Legend: (T) talk, (A) automated testing, (S) smoke testing, (MG) manual GUI testing, (MG*) manual GUI testing with protocol, (AG) automated GUI testing

	Requ.			Design			Implementation		
	B	M	E	B	M	E	Begin	Mid	End
A							A	A	A
B							MG	MG	MG, A
C							S, MG*	S, MG*	S, MG*
D									T, A
E									S, MG*
F							S, MG	S, MG	S, MG
G	T			T	T	T	MG, A	MG, A	MG, A
H									T, A
I	T						A, AG	T, A, AG	A, AG
J							A		S, MG*
K							MG*, A	MG*, A	MG*, A
L									A
M							T, A	A	S, MG
N									A
O							A	A	A
P		T					S, A, AG	S, A, AG	S, A, AG
Q							S	S	A
R							S	S	S, MG*
S		T					A		S, MG*, AG
T							A, MG	A, MG	A, MG

a secondary concern. At project start, the **AUT is still fuzzy** and students explore the problem's solution space. Automated testing seems hindering at this stage:

> *"Especially in the beginning, I always find it difficult to grasp automated test cases. In the beginning, [everything] is still vague. One thinks, 'Yeah, right now I'd like to just see how it works and just start implementing.' Yeah, it's this uncertainty again."* — team S

Later on, when the AUT begins to take form, testing still takes a backseat. Students have the feeling that an **incomplete AUT** is not ready to be tested automatically. They perceive the added test code as a burden that introduces unnecessary test maintenance. Students misinterpret the burden of **test maintenance as technical debt**—and accumulate technical debt in the process. Students opt to introduce automated tests when the AUT nears completion.

> *"[I] thought, testing is admittedly nice, but it's of secondary importance. First, something must be there. Particularly if I start right now—while our work is pretty unstable and we are permanently reconstructing things—I'm finding it relatively pointless to write a test, which one also would have to change every time. Because the things that we were testing were changing heavily. Because everything was still far too dynamic. At the end, there was no time left. That's why we never tested."* — team J

This feeling was especially strong regarding the user interface of the application. The **GUI of an application changes a lot during implementation** and thus introduces high maintenance costs for automated GUI tests.

> *"I found it pretty exhausting to test the GUI directly while implementing it. Since many things were changing on the side, I was like 'I want to change this and then would also have to change tests.' Therefore, one rather implemented the whole GUI first and then wrote test cases for it."* — team S

Some Teams preferred to push GUI automation near the end of the project, after the GUI was nearly finished. The same applied for most automated tests of other parts of the AUT. This strategy, however, introduces the first part of the *dilemma of late tests*. Students rob themselves of the advantages of having automated test suites *during implementation* and not only at the end. Some of the advantages are:

- Regression tests as a safety net against erroneous changes.
- Shaping the software design by thinking about tests.
- Reducing technical debt in the long run.

Students avoid automation by switching to manual testing. By manually smoke testing the application repeatedly during implementation, students uncover a lot of defects—by hand. This, however, is repetitive, time-costly, and monotone. Additionally, students *still* begin to automate tests at the end of development—as the existence of an automated test suite is required in the development process. This introduces the second part of the *dilemma of late tests*: students are then disappointed by the low defect detection rate. As most defects have been uncovered by manual labor before, the newly added test suite does not perform as hoped. *At this point, students clearly recognize the cost of automation but cannot see its benefits.*

One student pinpointed that testing *does* have a cost—a cost one will have to pay anyway. However, the earlier one invests in these tests, the more benefit can be gained from them. He explained that in order to do so, one has to conquer one's weaker self:

> *"One thinks that the actual logic that one writes is the major thing, and not the test cases. But that's this contradictory thing. When one writes automated tests, that's a constant effort, but it also brings infinetly many benefits - especially the earlier you write them. But you have to conquer your weaker self to write tests earlier."* — team S

Six teams agreed that testing earlier in the development process would have been beneficiary:

> *"Well, when one is doing it right from the beginning, it makes much more sense than thinking at the end, 'Now I'll also do a few test cases.' One benefits more."* — team Q

5.2 Education, Experiences, and Gaps in Knowledge

The category **Education, Experiences and Gaps of Knowledge** summarizes how the students' education and experiences shaped their testing behavior. We discuss in how far university courses helped them, whether they had prior private experiences, and whether they had worked on a commercial project before.

Education. University courses provide students with a wealth of testing knowledge. Students were introduced to

the testing technique *test first*. The use of JUnit was practiced in class. Different test design strategies, such as partition testing, white box testing, black box testing, and others were explained and practiced. However, students complained that the importance of testing is not conveyed strongly enough in courses. Additionally, the satisfying feeling of uncovering an actual defect is never experienced—they do not feel accomplished.

> *"You don't realize what a good feeling it is when you run a test. [Other student:] I think in the practical part many things are missing, too. The software quality course was very theoretical." — team* S

University courses do not seem to provide practical knowledge or experience, and students do not develop a habit of testing.

> *"I've heard something about it in a lecture. One has never actually carried it out in practice. One has talked about it theoretically, but I haven't had any practical affinity to it." — team* T

Experiences. Out of 20 teams, only two teams (S, I) had team members with prior experience in testing. Eight teams claimed to have no prior experience in methodical testing and test automation—except for knowledge from lectures and exercises. This heightened the barrier of getting started with testing.

> *"In the beginning, I've had many problems with just reflecting upon the best way to test something. We were all not so experienced to instantly know how to test it." — team* C

Also, most teams did not know any other testing tool besides JUnit, which had been introduced in a course.

Some students self-directed their learning and used books and internet tutorials as resources. However, some topics were harder to find than others, such as GUI automation with Eclipse.

Gaps of Knowledge. Besides their lack of experience with test methods and automation, other gaps of knowledge became apparent. One such gap concerns the *concept of GUI automation* for testing purposes. Six teams admitted to have no idea how to automate and test a GUI. These and others resorted to repeated manual executions of the user interface.

> *"I was responsible for the visualization [aspect of the software] and mostly, I just tested by executing [the program]. I constantly changed something and then looked at it, because I didn't have any ideas at all on how to test that." — team* D

Further, students encountered problems when *debugging*. Five student teams constantly put print statements all over their code to output variables for debugging. This method required a constant clean-up of debugging code and when an error reappears, the statements were reintroduced.

> *"When you're using* `System.out.println`*, you also have to remove it when you're done. And suddenly you notice that it doesn't work again, and you have to add all the statements all over again." — team* R

Also, this kind of error detection proved faulty due to misreads of outputs:

> *"But when I'm doing everything manually—even if I try to not miss any edge cases—I might sill misread the output that I get." — team* T

Another four teams tried to gain insights into their applications' inner workings and states by excessively making use of Java's `Logger` class. This indicates that our population was not familiar with proper debugging mechanisms.

Another relevant topic is the use of *measurement tools* to keep track of a project's testing efforts. Twelve teams admitted that they had never used any coverage tools when testing. Even though they did engage in automated testing, measuring their progress did not seem important to them. According to one team, testing was finished when "all possible errors" were detected.

> *"Yes, when we have tested everything that can occur. So, everything that we think of." — team* T

That same team later complained to have no feeling when to stop testing, as they could not cover all possible inputs:

> *"You don't know exactly when you're done. When have I tested enough? Which inputs do I have to expect now?" — team* T

5.3 Attitudes and Preconceptions

This section is concerned with the students' attitude towards testing and test automation. Further, we discuss preconceptions of testing and their impacts.

Attitude. Students generally saw testing as a part of software engineering. However, they were not particularly fond of test automation. They **value implementing over testing**. A functional product was more important than automation of tests and many teams resorted to manual testing efforts.

> *"I am not really fond of it. I like to implement, but eventually these are fair and help pretty much if there is a defect, so that one can really fast uncover that." — team* S

This attitude towards testing may originate from different factors. Firstly, students were eager to begin writing code and find a solution to the proposed problem. Writing tests instead of source code **does not feel as accomplishing** as creating new functionality. The benefit is not immediately apparent.

> *"Naturally [testing] is less exciting than implementing some function. You get to see less of it." — team* Q

Compared to producing new functionality, writing test code **feels unproductive**. There is no imminent reward.

> *"Well, it feels unproductive if you write tests at the beginning and actually you are thinking 'no, I'd rather start with implementation straightaway; quickly building a website, have a look how this will come out."' — team* S

Secondly, testing is perceived as an additional and **secondary task**:

> *"I thought that this testing was rather secondary. […] well, generally I would do it, sure."* — *team* P

Thirdly, some students displayed an **anxious attitude towards testing**. Testing uncovered defects in their work, and this entailed more work. Students did not like that.

> *"you are a little bit sad or unhappy, when you find a defect. Well, naturally you don't want to find that, that is for sure. [Laughter]. But then, you realize that you will absolutely have to do some more work or you realize that something is missing. To put it short: You are not done yet."* — *team* O

Lastly, some teams recognized the maintenance cost of a test suite and felt demotivated by it. New tests meant higher test maintenance when the AUT changed again. Introducing tests into development felt like **increasing the technical debt** to them—instead of lowering it in the long run.

> *"personally, I found this GUI test even more impracticable. I mean, me writing GUI tests incidentally all the time and because our GUI changes all the time dynamically... I thought that to be too much effort and I did not like that one bit."* — *team* S

Out of 20 teams, four teams explained to feel influenced in their testing behavior because of the educational status of the project. One student explained that his testing effort was reduced as this project would end with the course and would not be developed further.

Preconceptions. Students felt influenced in their testing behavior by different factors that they did not further explain and regarded as common knowledge. We regard these statements as preconceptions.

The **project size** was an important factor for students. According to them, smaller projects would not need to be tested. At which point a project would become large enough to justify systematic testing did not become clear.

The **criticality of a project** was also important to students. They felt that systematic testing would be justified by the potential damage that could be done by software defects. Games, for example, would not necessarily need the same quality assurance as software for ATMs.

> *"Games do not have the same quality requirements as other software, such as software that is required to work correctly. For example, ATMs. With this kind of software there can be great damage, but defects in games are simply annoying but not serious. That's my take on that."* — *team* I

As some of our proposed software projects were indeed games, this attitude may have influenced the teams testing behavior negatively.

Lastly, students adjusted their testing effort according to the **perceived complexity of the code** or their project's requirements. However, at which point code or a problem became complex enough to warrant a test was highly subjective. One student explained that some conditional statements do not need to be tested. This preconception, taken together with the attitude of regarding tests and test maintenance as a burden, might lead to technical debt in the long run.

> *"The program is not complex enough for some error to come up. If it was a complex program, then there could be a hidden defect that one could uncover with a test. But these programs are mostly if-statements or something like that, and you check that it should work—except when you make a mistake. But even your unit test can be faulty."* — *team* A

Another common preconception was that the unit testing framework **JUnit is best suited to test mathematical functions**. Five teams saw the greatest value of JUnit in checking internal calculations. One team member of a Web application project with little calculations explained their lack of JUnit tests with the absence of calculations and the use of third-party libraries.

> *"I think it is reasonable to test when you have many projects that really use formulas or somethings like that, mathematical formulas. In our case, we often used third-party components."* — *team* R

The misconception that JUnit is best suited for mathematical comparisons may be founded in the way JUnit is often presented in a mathematical context when introduced in courses.

5.4 Perceptions of Testing

This section describes perceptions of testing tasks from the view point of students. This includes problems and motivations for their testing behavior.

Students are aware of the benefits of an automatic test suite. Automatic tests can quickly uncover freshly introduced erroneous changes to the AUT. This heightens the confidence in the code:

> *"Well, I will write tests in the future, just because they find defects far quicker. If someone makes changes to the program, it may work without shutting down or some simple stuff stops working, that you don't see straight away. And if you have automated tests, you will notice straightaway. That is its advantage."* — *team* S

Students experienced that thinking about test design also enhances the AUT and helps them find important edge cases.

> *"Having tests was useful, especially if one saw the reason for them. Writing tests makes you think what kind of translation could trigger what kind of failure. Special cases like that you can write tests for and then you can always be sure that if someone changes the code and executes your tests, that in 99% of cases the program will work as expected."* — *team* S

However, students are also aware of the costs of testing. Writing automated tests takes cognitive and manual effort. Often, students did not have the time to read up on testing and did not find adequate tutorials online. From the perspective of a student, writing test code that is nearly as big as the concerned AUT code seems inefficient.

> *"No, I will not go through the effort of writing 20 tests and think them all through, just to cover one line of code or so."* — *team* T

Instead of investing in writing tests, most students decided for manual testing. This leads to faster execution and faster feedback and is cognitively less taxing.

> *"These were tests on the GUI of the application, I did not know how to automate these. Anyways, i was quicker, when we just clicked through it myself and checked whether it worked or not. [...] Perceived, this automation would be more effort than to just do it manually."* — *team* B

The downsides of manual GUI tests are manifold. These kinds of tests have to be repeated after every change to the AUT, which can be tiring. If the AUT needs to be in a specific state to start testing, this setup state needs to be established manually *every time*. One team wanted to test the behavior of a later stage of a card game and had to play through all preceding stages for each manual test. Also, testers tend to forget steps or make mistakes. One advantage of manual GUI tests over automated GUI tests is that humans are more robust against changes in the test script:

> *"If I did it manually, I would just say 'if I click* new task *or if I click* create new task*', that is not important for the human, but if [the test] does not find* create new task*, then it will not click on it, at least in Robotium."* — *team* P

GUI testing is perceived as something very difficult and ominous by the students. 6 of 20 teams regarded GUI testing as generally difficult. Defining the target value for a GUI test is challenging. One reason might be that the purpose of a GUI test can be misunderstood. Students often reduced GUI tests to asserting that some GUI element is placed correctly.

> *"Many defects cannot be found this way, when you consider GUIs. You cannot determine if an icon is misplaced. You really can't."* — *team* G

In other cases, students rightfully wanted to assert certain geometrical attributes in an editor—but did not know how to do so. This team could not find any technical solution to their problem and opted for visual checks.

> *"I find it difficult to check if some pixels are displayed correctly. How could one formulate this correctly? If it is about two lines being perpendicular, I find it easier to execute it and visually check for myself."* — *team* D

5.5 External Influencing Factors

This section describes external influencing factors with impact on the teams. Influencing factors may be originated from the organization, organization policies, advisors, coaches, or mentors. Since the course's team members were undergraduate students, other courses may have influenced their working behavior by limiting their available time and consequently influencing the time they may have spent on their projects.

Organizational Advice. Some coaches of our teams partially **discouraged their teams from testing** their products. One coach advised her team to give up unit testing. Another coach recommended writing tests after coding only because *test first* was hard to apply by novice teams.

One team had been completely excused from automated testing because of time pressure. It had been advised to concentrate on manually executed acceptance testing in the end of their project instead of automated tests during the implementation phase.

Many coaches in the software project excused their teams from testing their GUI automatically. In the case of Eclipse applications or Java-based games, **GUI automation was regarded as too difficult** for undergraduate teams.

> *"Later on, they told us that GUI tests with a bot or so were not even required. "Testing manually by hand" was ok."* — *team* H

Even though one team wanted to automate their GUI tests and had already begun so, one advice form above sufficed and the team stopped all efforts in that regard.

> *" than we thought we were expected to write GUI tests and we started doing so. In parallel, we had a talk with our coach and he said "no, you don't have to do this GUI stuff" and we stopped doing that."* — *team* M

At the same time, teams complained about missing organizational guidelines about required tests as well as missing specifications for testing processes.

> *"We really did not know how much we were supposed to test in order to pass this Quality Gate. It just did not say and that was frustrating!"* — *team* C

Furthermore, imprecise guidelines influenced the teams' testing behavior. Guidelines did not specify a high number of tests, thus students did not feel obliged to deliver many tests.

> *"There was no guideline which contains information about using these aspects. Well, what we have done in total has been relatively little. For me, I was surprised that it hasn't been valued much."* — *team* F

No Time to Learn: Although testing is taught in courses, students have no practical experiences in testing and even so, students may have forgotten how to test correctly.

> *"Well, I did not know about JUnit."* — *team* R

In this case, teams have to learn how to test. Because of the time pressure mentioned above, time is short in their project which leads to having **no time to learn testing**. Especially GUI testing is time-consuming while learning how to test or becoming acquainted with frameworks, which led to missing GUI tests.

> *"Then, one is satisfied with JUnit and uses it. I mean, time was short."* — *team* A

5.6 Summary

This section summarizes our findings with respect to the research questions we asked in section 4.

Our first research question was concerned with how students test their software products, i.e., which ***testing strategies*** they use. Students distinguished between automated and manual testing. Of those who used automated testing, most wrote unit tests with JUnit. For testing the whole application, most teams used unsystematic smoke testing, with only some using more rigorous forms of manual testing with a testing protocol or a checklist. A few teams automated their GUI tests. Only four teams had talked about testing in the first phase of the project. 11 teams did not plan their testing efforts at all. Two teams had a testing mentor among their members who provided examples and guided the other team members in their testing efforts. All teams started testing only in the last project phase. Testing was perceived as a secondary concern.

The second research question inquired about ***enabling and inhibiting factors*** with regards to testing. Teams said they had no time to learn testing during the project, as requirements, design, and development took up most of their time. Students perceived the project as uncritical and not complex enough to warrant testing, putting off learning testing for "real" projects. Many theoretically knew unit testing and JUnit, but had not used it in a real project before.

Students said they had not *experienced* for themselves the benefits that testing can bring. They especially mentioned that there was no feeling of accomplishment which would have motivated them to practice or use testing more. Many had no experience or knowledge in GUI or integration testing and thus were overwhelmed with the technicalities of these practices, so they did not use them.

One team with a more experienced student was relatively successful in its testing efforts. This student actively acted as a testing mentor and diffused the practice in the team. Another team also had a member with prior experience, but he did not actively mentor his team's member—consequently, testing did not become a focus in that project.

Our third research question was concerned with students' ***attitudes*** towards testing. Students are aware of the benefits of automated testing, but at the same time know that testing incurs a cost. Relative to their actual implementation, students said that the effort required for testing seemed very high. Especially GUI testing was perceived as very difficult. Thus, they opted for manual testing, which they said was easier in the moment and gave faster feedback.

Study participants liked the feeling of accomplishment that came from implementing software features, but at the same time said that they did not experience that feeling with testing. This made them feel less motivated to write tests. Some students even said that writing test code makes them feel unproductive, or that testing was a secondary task distinct from actual software development. Others disliked finding defects, as that would mean more work.

Finally, the fourth research question asked how students ***incorporated testing*** into their engineering process. They started testing late in the process, as they perceived their projects to be too unclear and unstable. Thus, students chose not to invest their time into writing tests that they would have had to change later anyway. This notion was especially prevalent towards GUI tests, as they were perceived as significant maintenance burden.

Teams started to test when they perceived their projects as stable. However, they had found most defects by then through manual testing. Thus, even when they finally wrote tests, they did not get to experience the benefits of testing. Even though some students claimed otherwise, we suspect that missing out on that positive experience will make it less likely for students to test early in future projects.

6. DISCUSSION

This section discusses our findings in light of *Diffusion of Innovations* theory as well as their potential impact for organizations, educators, and researchers. It concludes with a discussion of our study's limitations and threats to validity.

6.1 Diffusing Testing Practices

Diffusion of Innovations theory [19] describes how innovations—tools, ideas, or practices *perceived* as new—are adopted by individuals and organizations, and how they diffuse in social systems. Testing can be regarded as one such idea or practice.

Testing is often not taught alongside programming, thus adopting testing practices requires a change in behavior for no immediate or guaranteed benefit. Rather, one of the main perceived benefits of testing is that it *can avoid* the possible—but not guaranteed—occurrence of defects. As such, the consequences of adopting testing are uncertain: it may or may not prove beneficial in the future.

Diffusion of Innovations theory calls such practices and tools *preventive innovations*. Rogers [19] argues that the motivation of an individual to adopt such a behavior is usually rather weak, as benefits are uncertain and behavior change is associated with costs. Thus, individuals need *cues-to-actions* to eventually start adopting a tool or practice. These cues can be both personal experiences or experiences of others, for example watching a colleague or role model be successful with testing or experiencing problems oneself that were caused by not having written test code.

However, as became clear in our study, novices usually have not had much exposure to such experiences, personal or otherwise. The question, then, is: how can we expose students, novices, or junior developers to experiences that help them adopt testing practices?

Singer [23] proposes a set of interventions to facilitate the diffusion of software engineering practices that do not change the development process. This might be a suitable approach when processes cannot be changed.

In more flexible contexts, pair programming and mentoring seem like more powerful means to influence developer behavior. Pair programming has long been known to facilitate learning and student performance [14, 15]. As such, it seems like a suitable approach to help diffuse practices such as automated testing among novices.

We saw a more lightweight, but related approach in our study: a testing mentor that emerged in one team (team *I*, c.f. Table 1) exposed team members to good testing practices, guided them in their efforts, and provided examples that students were able to build upon. Such a mentor would not necessarily need to be restricted to being on a single team: they could visit different teams from time to time, diffusing practices through the whole organization.

Both pair programming and mentoring can be applied in organizational and educational contexts. In addition, students might want to expose themselves to good testing prac-

tices in a self-directed manner. The results of one of our previous studies [18] indicate that participation in open source projects may be a very suitable approach for this. However, open source can feel intimidating for some. In a different study, we found out about an alternative mode of self-directed, peer-based learning: the `#pairwithme` Twitter hashtag [24]. Twitter users use the hashtag to find random strangers for remote pair programming on a problem of their choosing. We suspect that this creates a marketplace for learning that opens up the possibilities for an individual to learn from a diverse set of others.

6.2 Implications

For ***software development organizations***, our study uncovered some phenomena that they quite possibly need to confront as well, albeit in weaker form. Some coaches from our group discouraged teams from automated testing. We conclude that the official policy in our group was thus not clear enough. In addition, the explicit guidelines given to students were apparently not demanding enough. For us, this constitutes a good reminder that policies and guidelines in education and professional organizations alike need to be clear and explicit if employees are expected to enact them.

Even though organization are unlikely to have teams solely composed of novices, they still need to train and accustom new employees to expected practices and conventions. Thus, it seems likely that adopting a method that diffuses desired practices amongst an organization's members, as proposed in the preceding section, would be beneficial.

To ***educators***, our results demonstrate that there is still a lot of potential for optimizing the learning experience. Students were busy learning team-based development, gathering and documenting requirements, designing, and implementing their software. This left them no time to learn testing practices simultaneously. Just as they had been taught programming before, our results suggest that there is a need for purely testing-oriented projects that let students focus on this part of software development. Because of our results and discussing in the preceding section, we suspect configurations with peer learning [17], mentoring, or pair programming to have great potential for such a testing course.

Finally, we hope that ***software engineering researchers*** will be interested to conduct similar studies on the perceptions and attitudes of developers towards testing. As we discuss in the succeeding section, our study has some limitations that warrant its replication in other contexts. Relatedly, a quantitative study on our findings would be desirable.

From a more technical perspective, we found GUI testing to be very challenging for students. Powerful tools and techniques are often the focus of software engineering research. Our study suggests that the practical adoption of software engineering practices also depends on the usability of practices and tools. Thus, we urge researchers to invest effort into lowering the barriers for developers to use their artifacts.

6.3 Limitations and Threats to Validity

We were interested in the phenomena that happened in students' testing behavior. Even though we had 97 study participants, we do not claim statistical significance. We intend to test the strength of our findings in a follow-up study.

As this study was conducted within a student project, some limitations related to this apply. Participants said that they did not take the project too seriously, as it was a course for learning and no critical software was developed. Students were also busy with other courses at the same time, which probably intensified the time pressure they felt. Yet, time pressure applies in commercial projects as well. Our follow-up study will focus on quantitatively validating or invalidating the phenomena we saw in this study with professional software developers of different experience levels.

Our population was self-selected, as students were free to decline the interview without any consequences. Yet, none made use of this option; our population was the complete set of one semester's software project course participants.

The interviews took part when all students had already successfully passed the project. We encouraged them to talk freely and found them to be very candid, e.g., some openly admitted that they only tested their project because at least some testing was mandatory in the course. We have no reason to believe that our participants might have given us an inaccurate picture of the testing-related events.

Our findings cannot be generalized to other populations. This is a known limitation of Grounded Theory studies. More studies are needed to find out which of these phenomena appear in other contexts. However, our findings might still hold for many other novices that are about to or have just completed their university degrees in computer science.

Teams comprised only of novice developers are unlikely to be the norm. However, because of the relatively extreme manifestation of our population in this regard, we suspect that we were able to uncover phenomena that might be present in weaker forms in many individuals or teams.

7. CONCLUSIONS AND OUTLOOK

We interviewed 97 computer science students about their testing behavior in a software project course. These students were about to earn their Bachelor's degrees in computer science and join the software industry.

Amongst other findings, we documented that novice developers may perceive testing as a secondary, cumbersome task that they are not motivated to spend effort on as they have not experienced its benefits before. Time pressure keeps them from crossing the chasm between learning testing and becoming productive with it. Since testing is a *preventive innovation*, motivation to invest effort into learning it is especially low. This sentiment was especially pronounced for GUI testing, as it is even more challenging technically.

Our results suggest that software companies could benefit from being aware of these experiences, perceptions, and attitudes of junior developers. Collaborative development with more senior developers might help mitigate this and similar problems with other software engineering practices. Educators might need to consider providing additional courses solely focused on testing or using alternative learning methods such as peer learning, mentoring, or pair programming. We hope that software engineering researchers will conduct similar studies in different contexts. In addition, the technical barriers we discovered provide problems with testing tools and practices that have yet to be solved by research.

As our next step, we will conduct a quantitative study, comparing our findings with the situations of experienced as well as inexperienced professional software developers from industry. This will allow us to trace how developers' testing practices develop over time and will help identify gaps in both academic education and industrial training.

8. REFERENCES

[1] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[2] A. Beer and R. Ramler. The role of experience in software testing practice. In *Software Engineering and Advanced Applications (SEAA), 34th Euromicro Conference on*, pages 258–265, Sept 2008.

[3] B. Benton. Designing and building a software test organization. In *Software Testing, Verification, and Validation (ICST), 1st International Conference on*, pages 414–422, April 2008.

[4] A. Causevic, D. Sundmark, and S. Punnekkat. Factors limiting industrial adoption of test driven development: A systematic review. In *Software Testing, Verification and Validation (ICST), 4th International Conference on*, pages 337–346, March 2011.

[5] M. Ellims, J. Bridges, and D. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[6] E. Engström and P. Runeson. A qualitative survey of regression testing practices. In M. Ali Babar, M. Vierimaa, and M. Oivo, editors, *Product-Focused Software Process Improvement*, volume 6156 of *Lecture Notes in Computer Science*, pages 3–16. Springer Berlin Heidelberg, 2010.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[8] R. Hoda, J. Noble, and S. Marshall. Developing a grounded theory to explain the practices of self-organizing agile teams. *Empirical Software Engineering*, 17(6):609–639, 2012.

[9] T. Kanij, R. Merkel, and J. Grundy. Performance assessment metrics for software testers. In *Cooperative and Human Aspects of Software Engineering (CHASE), 5th International Workshop on*, pages 63–65, June 2012.

[10] T. Kanij, R. Merkel, and J. Grundy. An empirical study of the effects of personality on software testing. In *Software Engineering Education and Training (CSEET), 2013 IEEE 26th Conference on*, pages 239–248, May 2013.

[11] T. Kanij, R. Merkel, and J. Grundy. A preliminary survey of factors affecting software testers. In *2014 Australasian Conference on Software Engineering (ASWEC 2014)*. IEEE CS Press, 2014.

[12] K. Karhu, T. Repo, O. Taipale, and K. Smolander. Empirical observations on software testing automation. In *Software Testing Verification and Validation (ICST), International Conference on*, pages 201–209, April 2009.

[13] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *28th International Conference on Software Engineering (ICSE)*, pages 492–501. ACM, 2006.

[14] C. McDowell, L. Werner, H. Bullock, and J. Fernald. The effects of pair-programming on performance in an introductory programming course. In *33rd Technical Symposium on Computer Science Education (SIGCSE)*, pages 38–42, 2002.

[15] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. The impact of pair programming on student performance, perception and persistence. In *25th International Conference on Software Engineering (ICSE)*, pages 602–607. IEEE, 2003.

[16] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Chen. A preliminary survey on software testing practices in Australia. In *Australian Software Engineering Conference (ASWEC)*, pages 116–125, 2004.

[17] A. M. O'Donnell and A. King. *Cognitive perspectives on peer learning*. Routledge, 1999.

[18] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *International Conference on Software Engineering (ICSE)*, pages 112–121. IEEE Press, 2013.

[19] E. M. Rogers. *Diffusion of Innovations*. Free Press, 5th edition, 2003.

[20] J. Rooksby, M. Rouncefield, and I. Sommerville. Testing in the wild: The social and organisational dimensions of real world practice. *Computer Supported Cooperative Work (CSCW)*, 18(5-6):559–580, 2009.

[21] P. Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, July 2006.

[22] K. Schneider. LIDs: A Light-Weight Approach to Experience Elicitation and Reuse. In F. Bomarius and M. Oivo, editors, *Product Focused Software Process Improvement*, volume 1840/2000 of *Lecture Notes in Computer Science*, pages 407–424. Springer, 2000.

[23] L. Singer. *Improving the Adoption of Software Engineering Practices Through Persuasive Interventions*. PhD thesis, Gottfried Wilhelm Leibniz Universität Hannover, 2013.

[24] L. Singer, F. Figueira Filho, and M.-A. Storey. Software Engineering at the Speed of Light: How Developers Stay Current Using Twitter. In *International Conference on Software Engineering (ICSE) (to appear)*, 2014.

[25] B. Turhan, L. Layman, M. Diep, H. Erdogmus, and F. Shull. How Effective Is Test-Driven Development? In A. Oram and G. Wilson, editors, *Making Software: What Really Works, and Why We Believe It*, pages 207–219. O'Reilly Media, Inc., 2010.

[26] S. Vegas and V. Basili. A characterisation schema for software testing techniques. *Empirical Software Engineering*, 10(4):437–466, 2005.

[27] A. Zaidman, B. Rompaey, A. Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.