Abstract

Students often voice that the programming assignments are hard and that they spend a lot of time on solving them. Is this true; are we giving them too hard assignments and how much and what do they spend the time on? This is what we want to gain insight to. We constructed a tool that records programming sessions with finer granularity than the existing solutions. The tool has recorded 2643 programming sessions from students. Using that data we found that students spend only 15% of their time writing code, and that on average 40% of their programming effort is spent reading and navigating. We also estimate the time spent outside of the tool to be almost 20%. The increased detail in the recordings can be used to measure the effect of programming source code comments, and we found that the effect of both helpful and redundant comments increases the reading time but did not help to reduce the students writing effort. Finally, we used the tool to examine the effects of an improved programming assignment and found that the total effort was not reduced.
Acknowledgements

I would like to thank both of my excellent supervisors Morgan Ericsson and Anna Wingkvist for sharing their expertise, guiding me when I was lost, and for their patience.

I also want to thank all of my colleagues on both campuses, which have encouraged me, provided endless interesting discussions, read my work, looked at my drawings, listened to my ramblings, collected coffee with me, smiled with me, high-fived, and made every day of my work so much fun.

Special thanks goes to my family for providing me with the time to work, supporting me, and making life worth it.

Finally I want to thank all who have provided me with the opportunity to grow, to be challenged, and to search for my truth.
# Contents

1 Introduction ................................................. 1  
  1.1 Background ............................................. 1  
  1.2 Measuring Student Effort ................................. 2  
  1.3 Problem Definition ...................................... 3  
  1.4 Organization ........................................... 6  

2 Related Work ................................................. 7  
  2.1 Learning Analytics and Educational Data Mining .......... 7  
  2.2 Tools that Collect Programming Events .................... 8  
    2.2.1 Submissions ......................................... 9  
    2.2.2 Compilation ........................................ 10  
    2.2.3 Key Stroke .......................................... 10  
    2.2.4 Interaction .......................................... 11  
  2.3 Usages of Data ........................................... 11  
  2.4 Skill, Quality, and Effort ................................ 12  
    2.4.1 Skill of the programmer .............................. 12  
    2.4.2 Quality ............................................... 13  
    2.4.3 Motivation and Behavior ............................. 15  

3 The CSQUIZ Learning Environment ........................... 17  
  3.1 A Distance Learning Platform with an Online Programming Environment ........................................... 18  
  3.2 Teacher Perspective ...................................... 20  
  3.3 CSQUIZ as an Experimental Tool ......................... 23  
    3.3.1 Replay of Recordings ................................. 24  
    3.3.2 CSQUIZ Experiments .................................. 28  
  3.4 CSQUIZ Architecture ..................................... 29  

4 Methodology .................................................. 31  
  4.1 Evaluate Measuring Tool .................................. 31  
    4.1.1 Time Aggregation Models ............................ 33  
    4.1.2 Granularities ....................................... 33  
    4.1.3 Dividing Time into Activities ....................... 34  
    4.1.4 Comparing Granularities ............................. 36  
  4.2 Experimental Design ..................................... 38  
    4.2.1 Subjects ............................................. 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.2</td>
<td>Redundant Comments</td>
<td>41</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Use a Class With Comments</td>
<td>43</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Informed Change</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Method Discussion and Validity Threats</td>
<td>47</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Field Experiments</td>
<td>48</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Mortality</td>
<td>49</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Ethical Considerations</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>Results and Analysis</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Dividing Time into Activities</td>
<td>51</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparing Granularities</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>Experiments</td>
<td>56</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Redundant Comments</td>
<td>56</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Use a Class With Comments</td>
<td>58</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Informed Change</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>Discussion</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>Dividing Time into Activities</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparing Granularities</td>
<td>69</td>
</tr>
<tr>
<td>6.3</td>
<td>Experiments</td>
<td>70</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Redundant Comments</td>
<td>70</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Use a Class With Comments</td>
<td>71</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Informed Change</td>
<td>72</td>
</tr>
<tr>
<td>6.4</td>
<td>Using CSQUIZ</td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>Examining Skills</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>Future work</td>
<td>79</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

“Measurement is the first step that leads to control and eventually to improvement. If you can’t measure something, you can’t understand it. If you can’t understand it, you can’t control it. If you can’t control it, you can’t improve it.”

— H. James Harrington

As teachers we design programming assignments with the hope that our students will learn programming techniques, methods, tools, and strategies from them. An assignment is designed to give students some knowledge they will need to know for future assignments or to pass the course. It builds on knowledge that students should have obtained in earlier assignments. We design our assignments based on our idea of an average student. As programming teachers we already know the strategies needed to complete the assignment so the solution may be obvious to us. It is difficult for us to predict how actual students will perceive the assignment. To help students, we provide information for that average student in the form of lectures, instructions, tutorials, and tutoring.

When students work on the programming assignment they might differ from the average student we base the course and course material on. The assignment may be harder or easier for them than we planned for. The difficulty depends on the students’ background, but also on what happens during the programming session. Students often run into problems that we did not foresee: they misunderstand the assignment, face issues in their programming environment, introduce errors, and encounter cryptic error messages that they are not yet equipped to handle. The more we know about how the students are doing, the better we can foresee and plan for these problems. If we know that students often misinterpret an instruction, we could write a better one; if we know that they do not read the instructions, we could guide them to it or figure out why.

We struggle to understand the problems our students face. During tutoring, we collect information on how the students perceive the assignments. We use this information to patch and tweak the assignments to better fit our goals.
However, when we listen to students we often get mixed messages: some students say the assignments are too hard, take too long to solve, and that they get stuck. Other students seem not to perceive an obstacle at all, and may think that our assignments are too easy. Discussions with individual students do not give us a good overview of the class, and are thus not a good ground for improvements on their own.

Fenwick et al. [FNB+09] did a survey on novice programmer behaviors. They found that the students’ perception of how much time they spent differed significantly from the teachers’ perception. Students thought they spent much more time on programming than their teachers believed they did. Without measuring we do not know how much time students spend on our assignments, and if the group of students who struggle is large or small. Our perception of what is going on in the programming class is also lacking since we get an increasing amount of students, offer courses online and on multiple campuses, etc. We are thus unable to spend as much time with each student as we once did; we instead rely on teaching assistants to handle most of the student contact. Our tools, pass rates, and course evaluations do not provide us with good enough information on what problems the students encountered. This information is also provided too late in the course to fix any problems.

We need to know how much effort (measured in time) students spend on assignments to improve our courses. From this measure of effort, we can plan our courses and advice students on what workload they can expect. We also need to understand the obstacles students face when we give them programming assignments. If we understand the obstacles we can guide students through them, or deliberately construct new obstacles to challenge them.

1.2 Measuring Student Effort

In an attempt to measure the students’ effort, we asked our students to report how much time they spent on an assignment [TOWE14]. Figure 1.1 shows that the time needed to solve the assignment varies greatly: the fastest student spent less than four hours while slowest spent a staggering 56 hours. We have no information on the time spent by those who did not submit, so we cannot tell whether those students attempted and gave up, or did not even attempt.

We also attempted to improve a programming assignment based on more information about the students’ strategies. We constructed an experiment with two factors: source code comments and type-hinted arguments. Our hypothesis was that students who received more information would also solve the assignment faster. We created four different versions of the code, with and without the source code comments and with and without type-hinted arguments. Each student was randomly assigned one version of the code.

The time to solve the assignment varied greatly but independent of version, so we could not determine whether there was an effect for any of the factors. The median time is close to eight hours independent of factor (c.f., Figure 1.2). We found two reasons for the variations: first, the students measured time and requirement fulfillment manually. They were given instructions, but the students interpreted them in different ways. Second, the students reported that
they were frequently interrupted in their work. Both of these reasons result in time measurements that were not accurate enough.

1.3 Problem Definition

We see a need to increase the quality of our programming assignments but have no way of evaluating if changes we make are for the better or for the worse. A programming assignment with higher quality should better fit the need of the students and may for example have more clear instructions, higher quality source code, or give better feedback. We state the following hypothesis.

**Hypothesis:** Given that we have two programming assignments with the same functionality and same solution but one has higher quality, we expect the effort for a student to complete the assignment to be lower in an assignment of higher quality than to an assignment of lower quality.

In the simplest of cases this hypothesis holds true. Obfuscation is a technique where a piece of code is changed so that it becomes unreadable and is commonly used in the JavaScript community to hide code in plain sight. The obfuscated code is definitely harder to read than the original code even if it fulfills the same requirements. The effort to solve a programming assignment in obfuscated code should therefore be much higher. Compare the normal (top) and obfuscated (bottom) parts of Listing 1.1 on page 5; the top is easier to understand because it is designed to be read by a human. To work with the
obfuscated code would require an effort to translate back to a readable form in order to make changes.

The effort to complete an assignment should reflect the difficulty and the amount of work, including learning. If effort can be measured, difficult assignments can be identified, analyzed, and improved. By measuring the effort again we should be able to observe the effect of the improvement. In short, we want to be able to do make informed decisions and observe the result.

However, to measure effort is not trivial. We define the effort as the time students are actively working on the assignment. So, to accurately measure effort we must know if the student is somehow working on the assignment or is involved in another activity. Our students are distributed in time and place, so we cannot directly observe them during their programming sessions. We also cannot ask students to measure effort, since their measurements were not accurate enough. We need a tool that allow us to objectively measure the student effort with high accuracy, handle interruptions, and allows us to view programming traces so that we can investigate differences.

**RQ1.** How can student programming assignment solving effort be automatically measured with high accuracy?

When effort can be measured we want to evaluate how we can improve the assignment w.r.t. effort without affecting the learning. One such improvement could be to improve the quality of the provided source code by adding good comments. These should help the students understand how a piece of code can be used, and let them focus on the learning goals of the assignment.

---

**Figure 1.2:** Results from an experiment in 2013 that compared two factors, existence of source code comments and type-hints. If there is an effect, it is hidden in the large variation of measurements.
Listing 1.1: The JavaScript obfuscator produces unreadable code that is harder to understand and work with. Top and bottom contain the same code but code on top is easier to change than the obfuscated code at the bottom. Example taken from http://javascriptobfuscator.com

RQ2. Do source code comments in a programming assignment significantly decrease the effort to solve that assignment?

Another improvement of an assignment would be to divide one large and complex assignment that students struggled with into two smaller assignments to let the students focus on one problem at a time and provide feedback halfway through.

RQ3. Does dividing a programming assignment into two smaller assignments decrease the total effort of solving the assignment?

While research questions 2 and 3 are interesting on their own, they also serve as examples of how the accurate effort measurements from research question 1 can be used in practice.
1.4 Organization

The rest of the thesis is laid out as follows: in Chapter 2 we describe other tools and related work relevant for this thesis. In Chapter 3 we describe our tool capable of measuring effort with higher accuracy than other tools. In Chapter 4 we describe how we use the tool to answer the research questions. First we describe how we intend to evaluate the tool and how to compare it with tools based on other granularities. We also construct three experiments based on detailed effort estimations to answer research questions two and three. After that, we present the results and analysis in Chapter 5. The results are interpreted and discussed in Chapter 6. In Chapter 7 we answer the research question and state our conclusion. Finally, we give the direction of our future studies in Chapter 8.
Chapter 2

Related Work

In this chapter we introduce the fields of Learning Analytics and Educational Data Mining. Next we dive into articles on capturing events from student programming sessions and take a look at tools that are used to record such events. We also give an overview of how collected data has been used by other researchers. Next, we show research on programming skill and effort. Finally, we show research that has been done on observing programmers.

2.1 Learning Analytics and Educational Data Mining

Learning Analytics is the process of measuring on students learning environment to gathering data on students learning. Educause, a non-profit organization that strives to improve education through technology describes Learning Analytics as: “L[earning] A[alytics] collects and analyzes the ‘digital breadcrumbs’ that students leave as they interact with various computer systems to look for correlations between those activities and learning outcomes”[Rit11, p. 1]. These digital breadcrumbs can come from many different sources, from learning environments, web-sites, or entered by the teacher or student. It could be results from exercises or access information on online resources. A similar field is Educational Data Mining in which data from students is also collected. Learning Analytics and Educational Data Mining is two relative new fields that share many connections. According to Ryan Baker [RB14], a Learning Analytics researcher aims to empower and inform teachers and students by human-led methods for visualizing and exploring educational data, and strives to provide a holistic understanding of the complete learning-system. Educause defines the goal of Learning Analytics as to improve the learning process by improving the learning environment and allowing the teacher to intervene. These interventions could either be targeted at students that are deemed to be “at-risk”, or by identifying assignments that are problematic for a group of students, and improve them [Rit11]. Greller and Drachsler [GD12] identifies two main objectives in Learning Analytics: prediction and reflection. Prediction is used to
support decision making. For instance, prediction of early drop-outs leads to an early intervention or to an adaption of learning material. Reflection on the other hand is supposed to change the data clients view. A student may get personalized information on their progress, and teachers may use the information to reflect on their teaching style.

In contrast the Educational Data Mining researcher strives to let software personalize the learning experience by automated adaptions. Thus the researcher may use automated methods for finding constructs and their relationship in the data [RB14].

There are Learning Analytics and Educational Data Mining tools that collect different types of events that can be used to estimate the students’ effort.

### 2.2 Tools that Collect Programming Events

We conducted a limited search to find Learning Analytics and Educational Data Mining tools that capture students programming process and would allow us to measure the student’s effort with high accuracy. We searched ACM, IEEEXplore and OneSearch databases using query strings and excluded studies that fall outside of programming or learning domain or did not mention a tool in the abstract. In total 161 studies were examined and we found 38 papers that mentioned recording program process events. Some of the papers share both tools and authors. We extracted the programming language of existing tools and most tools support either Java (64.1%) or Python (12.8%) and we found no tool that supported PHP programming language. Why this is important will be revealed later.

In a more recent ITiCSE Working group, that the author participated in, 3571 research papers were scanned for automatic collection of programming process data. The different studies were categorized into six granularity levels, presented in Figure 2.1. When more events are collected, we call it a finer granularity of events. On the coarsest level submission events are collected, and on the finest granularity level individual key stroke events are logged [IVA+ p]. The results from the literature review confirm that there is no support for PHP [IVA+ p].

Vihavainen, Luukkainen, and Ihantola [VLI14] investigated the frequency of events in the context of capturing the process. They compared collections of data on submission, snapshot, and key stroke granularities. They found that much information on how the student edited the code is lost when collecting data on lower granularities. Since we want to capture the effort of students, the frequency of events matters.

For example: Submission events can be recorded each time a student submits their solution for testing. These events could be saved in a log or database from which statistics on effort can be generated. Using submission data, we can estimate the effort as the time from the first submission to the last. There is an uncertainty to such estimation since no information is available before the first submission. We can only guess how much time a student spent before it. The students probably interacted with the environment before their
first submission. By collecting data on a finer granularity, perhaps including compilation events, we can build a better estimation of the students’ effort.

We extracted and categorized events captured by tools in the 38 papers.

2.2.1 Submissions

A submission system automates the process of gathering and assessing student programming code. Students submit their assignment solution to an automated system, the system saves the submitted code-snapshots and also saves metadata like submission time-stamp and student-identifier. Typically a number of tests are run against the student solution and feedback is returned to the student. The result of these tests are saved for later analysis [Edw04; SHP+06; EPQ08; AE10; BE13; SFSR13].

The most common tool for collecting submission data in our dataset is the Web-based Center for Automated Testing (Web-CAT) [Edw04; EPQ08; BE13].

The frequency of submissions may depend on if the tool allows the student to submit as many times as they like or if there are restrictions. Spacco et al. [SHP+06] stimulate early work by giving the students a limited amount of release tokens. The release tokens can be traded in for a release test to be run and feedback will be given [SHP+06; SFSR13]. The frequency of submissions may also depend on the amount of feedback that students receive from submitting. Marmoset by Spacco et al. [SHP+06] has three types of tests: Public tests that are handed out to the students before first submission, release tests that produces feedback on submissions, and secret test were no feedback is given to the students.

Coarse grained data with a frequency similar to submissions can be collected from student source code repositories. The frequency of data in repositories depends on the students habits of committing code into the repository [MLRW05].
2.2.2 Compilation

While submission events can be collected using a centralized server, recording compilation events requires either the editor or the compiler to be instrumented. The collected events also need to be moved from the client to a centralized storage. Either the events are aggregated on the client [NBFJ+08] and uploaded when student submit or the events are directly sent to a server when they occur. Several authors instrumented the popular BlueJ editor to submit compilation events [Jad06; TMRJ08; WLG13]. Murphy et al. [MKLH09] choose instead to instrument the java-compiler “javac” and created plugins for both BlueJ and Eclipse. Their tool, Retina, collects compilation events together with metadata about the student and time without any special interactions by the students [MKLH09].

By collecting compilation events researchers are able to collect errors from the programming sessions that students would have removed before submitting data [Jad06; TMRJ08; MKLH09; JCC05; AEH05; DLRT12]. Using such data researchers like Jadud [Jad06] have compiled lists of common compilation errors and found that students tend to make the same mistakes over and over again.

The frequency of compilation data largely depends on editor. Eclipse compiles every time a file is saved, thus this data is as fine grained as file save granularity. In other cases compilation is initiated by the programmer and is thus sparser.

There are many tools that combine compilations with other types of events like code edits, save, and commits, or combines the data with manually edited information like midterm score or manual observations of students [PSK+12; Bli11; FNB+09; RB09; TMRJ08].

File save events may be recorded by instrumenting the editor or writing plugins to popular editors. For example Spacco et al. [SHP+06] instrumented the Eclipse editor with a plugin to allow the Marmoset tool to collect data on each save event. This is also the case with Piech et al. [PSK+12] who collect compilations, commits, and also file saves using Eclipse.

File saves are often used in combination with other events. For example the tool “Test My Code” combines file saves with test run and submissions [Vih13; HVB14].

2.2.3 Key Stroke

We have found two tools by Vihavainen, Helminen, and Ihantola [VHI14] and Matsuzawa, Okada, and Sakai [MOS13] that capture all key stroke events. The papers are a bit vague on if they include keys that do not edit the content of files such as arrow-keys. For that reason we do not know if other tools capture text-caret (sometimes called text cursor) events or not.

There are also tools that collect edit events in programming languages that are not text based or edit the texts in larger chunks such as the states in Kara or Parsons problem [HIKA13; HIKM12; KSBR10].
2.2.4 Interaction

A number of tools report to collect other events from the editor such as button clicks, time spent on pages, events from the debugger, application focus, and console activity [Bli11; HIK13; HDDI+04; HIK13]. For example Helminen, Ihantola, and Karavirta [HIK13] developed a Python tool with both code editor and a console that students can test their application through. Their environment records detailed traces with code edits, code runs, commands run in console, submission, and application-focus events. Such events are important to capture more of the programming process.

Norris et al. [NBFJ+08] combines submission, compilation, and file saves by collecting programming traces using a plugin to the BlueJ editor. To get a better estimation on how much time students spend in the tool they also collect tool interaction events like: start, exit, project, and package events.

We found tools that captured fine grained data, but we find little support for recording mouse, or text-selection events. The only tool we have found that captures mouse information is the Eclipse plugin “Watcher” that collect time for Browsing, Scrolling, Mouse Events and Coding. “Watcher” is not designed as a Learning Analytics tool but to monitor programmers [ME04]. We find it unclear to what extent the data is recorded and if it could be played back.

2.3 Usages of Data

The collected data have been used to achieve different things:

Some of the tools visualized of the programming process so that researchers may find high-level information in a time-line of events [MKLH09; MOS13; BS13; HIK13]. Helminen, Ihantola, and Karavirta [HIK13] developed a trace exploration tool with snapshots, showing change differences and test or compilation feedback. They used the visualization to observe how students tested their code and to what extent they used tests given in assignment or wrote their own. Heinonen et al. [HHLV14] has developed a web based application for snapshot analysis called CodeBrowser with an API for easy integration by other researchers. Matsuzawa, Okada, and Sakai [MOS13] developed the “Programming process visualizer” that visualize data recorded from the editor. It can animate and replay recorded data. The visualization is aimed to students so that they can reflect on their personal programming process [MOS13].

Other tools are instead aimed to support the teacher by giving grounds for interventions or to support the student by giving recommendations on better habits [HWM05; NBFJ+08; MKLH09].

A number of studies report lists of common errors that students encounter, and records how students react on them, or attempts to help the students by explaining the error [SS86; Jad06; TMRJ08; TRJ11; KN09; VHI14]. Spacco et al. [SSHP05] also reports run-time errors.

There are also tools that are used to study undesired student behaviors, such as copy paste, remove the line with the error, or students who mindlessly test their way to the solution state. To do this, some model the problem solving effort using states and state transitions to find patterns [MKLH09; AE10;
HIKM12; MOS13], while others try to extract programming strategies [KN09; HIKM12; KSBR10; Jad06].

There are also tools that attempt to group students into categories depending on their success-rate and use that to predict the outcome of their effort [Vih13; TMRJ08], or that attempt to model the students’ skill or frustration [Jad06; RB09; TRJ11; FF12].

2.4 Skill, Quality, and Effort

The effort of programming depends on several factors; skill of the programmer, quality of the instruction, tool, and code, and motivation and behavior of the programmer.

2.4.1 Skill of the programmer

A student who is less skilled needs to learn to solve an assignment. Learning takes time and we expect students who learn to spend more effort. Researchers within the two related fields of Learning Analytics and Educational Data Mining use data collected from the educational context to estimate the skill of students and predict their results [RB14]. Lack of skill may cause students to struggle by hindering them from making progress.

Finding struggling students is a goal of both Learning Analytics and Educational Data Mining [RB14]. Jadud [Jad06] created a predictor called Error Quotient (EQ) for finding struggling students. The model was built using grounded theory. Jadud created visualizations of where in the code an error was found, and where students changed their code in relation to the error. Jadud found that students are not always making progress, instead they recompile with the same error. Jadud saw this as a sign of frustration. EQ is built from a series of compilation snapshots with error messages. EQ is a value between zero and one where zero means that no compilation error has occurred and one means that every compilation ends in the same error type. The model was tested on 56 students and a “distinct correlation” between EQ and grade was found. Unfortunately a very low fit of the data indicates that there are unknown confounders and thus Jadud does not make any strong claims of the models usefulness as a predictor [Jad06].

Tabanao et al. also used EQ as a predictor using an updated version of the algorithm that takes location of error and the location of edits into consideration. Tabanao et al. tested this on snapshots from 143 students. EQ was found to be significantly negatively correlated to midterm score and students who get high EQ scores may be at risk of failing the programming course [TMRJ08].

In a later study Tabanao, Rodrigo, and Jadud [TRJ11] used the students’ midterm score to divide their students into three categories: At-Risk, Average and High-Performing. Student’s one standard deviation above mean was assigned to the High-Performing group, while students with scores one standard deviation below average were assigned to At-Risk group. Five regression models were tested to predict midterm score by using snapshot analysis on 24 000 compilation events from 143 students. The first model, based on the total
number of errors, was found to explain only 14% of the variance. Tabanao, Rodrigo, and Jadud [TRJ11] second model built on the occurrences of three common errors. This model had problem with predicting the High-Performing group but was better at predicting At-Risk students. The third model used time between compilations, but this model was very weak. The fourth and fifth models by Tabanao was built on EQ scores, these models explained roughly 30% of the variance. They report however that these models were also poor when trying to predict At-Risk students [TRJ11].

Piech et al. [PSK+12] used a Hidden Markov Model that was trained on snapshot data collected using an Eclipse plugin. The model was built up from a state transition graph illustrating the students’ code progression through the assignment. In order to compare snapshots from different students Piech et al. [PSK+12] calculated a distance metric comparing how similar two programs were. The snapshots were grouped into a set of high level states using clustering techniques. Transition probabilities represent the probabilities of ending up in a state from the previous state. They found that students took similar paths through the states. They also identified states that they called “sink states”. Sink states are dead end problematic states from where a student does not easily recover. Students were clustered into groups with different state-transition behavior. Students in the alpha group built their solutions in small iterative steps, while students in the gamma group tend to get stuck in sink states. A significant difference in midterm score between the different groups was found.

We would expect strugglers and students at-risk to benefit from higher quality in instructions and more readable code.

2.4.2 Quality

The quality of the instruction, tool, and code given to the student should also affect the effort. For example: Better code should lower the effort by being more readable. We have studied articles on code readability and experiments on code quality and effort.

Time has been used as an indicator for the usefulness of static typing versus dynamic typing [Han10; KHR+12; HKR+13]. In one notable experiment Hanenberg [Han10] compared development times in a programming language specifically constructed for the experiment. The programming language was constructed to reduce the number of confounding factors in the experiment. Even so the author could find no negative or positive impact of the treatments. The individual differences of the subjects were too large. The fastest student did the programming in one hour while the slowest student did it in 16 hours. Both programmers were in the same group. Kleinschmager and Hanenberg [KH11] claims that being able to estimate the skill of programmers is important to design blocked experiments, generalize from experimental findings, and to relate an effect to a specific level of developer. In a similar experiment Kleinschmager et al. [KHR+12] used within subject experiment design were each subject is given both treatments. Within subject design allowed Kleinschmager et al. [KHR+12] to reduce the effect of individual differences in
programmers skill. A problem with within subject design is that the second task may be solved much faster since the subject learned how to solve it in the first task. To reduce the learning effect, Kleinschmager et al. [KHR+12] renamed all classes and methods in the second task. Using such experimental design, static typing was shown to have positive effect.

The skill of programmers is thus very important to make claims of improved quality of tools or code. Sjoberg et al. [SYA+13] hired six professional programmers that did adaptive maintenance and added functionality on four different systems with identical functionality. They used pretest scores to find programmers of similar skill to include in the experiment. Effort was used as an inverse indicator of code quality by testing if programmers spent more time in files that contained code-smells. To measure time they instrumented Eclipse using the Mimec tool and measured effort in the form of time spent in files. Sjoberg et al. [SYA+13] also used within subject design to control for developer differences. They controlled for system, round, file size, and revisions and correlated occurrences of code smells to effort using multiple regression techniques. Interestingly they find that none of the code smells increased effort, and that one smell even decreased effort. They suggest that file sizes explain most of the variance in their model and suggest that file size may have a big impact on code maintainability. Thus if we add source code comments, we might increase the students effort.

Ko et al. [KMCA06] has shown that code comprehension is a large part of the programmers total effort. To successfully extract information from code requires skill, since the students need knowledge of syntax, code conventions, and camel notation to make sense of the code. Buse and Weimer [BW10] defines readability as the “human judgment of how easy a text is to understand”. We believe that readability of code may play a bigger role for students than for experienced programmers since they may lack the skills needed to interpret code. Lopez et al. [LWRL08] divided the skill of reading code into two separate sub-skills. The first skill is to explain code in plain English and the second skill was to manually trace code (determining values of variables after a number of instructions have been executed). Lopez et al. [LWRL08] found a strong correlation between being able to explain a piece of code in “plain English” and the skill of writing functional code. They also found a strong correlation with code-tracing skill and writing code. In other words students who can explain or trace code are also good at writing code.

Struggling students that lack code reading skills may be helped by introducing comments into the code. Comments can contain plain English explanations and Tenny [Ten88] hypothesize that comments can be used to improve otherwise unreadable code. However in a study by Buse and Weimer [BW10] comments had only a moderate correlation to readability and had much less influence than for instance keeping lines short. Their hypothesis was that comments are more common in unreadable code. They also found that having too many identifiers or too many characters per line had a strong negative influence on readability.

Improving code readability by adding comments could thus be a way of improving a programming assignment, but it is unclear if the students will
solve the same requirements with reduced effort.

2.4.3 Motivation and Behavior

Finally, we take a look at observational studies on programmer behavior and motivation. Students may aim for different qualities in their code and it may affect the effort needed to produce that code.

Effort is a rough measurement on what happened during programming. By only measuring time we are unable to separate between skill and quality as the reason behind a change in effort. To find causes for variation we need to observe students behavior and motivations. We have found two studies from the field of program comprehension that observed professional programmers in their work [KMCA06; MTRK14]. These are interesting since they show us what knowledge could be extracted if even more fine grained recordings than existing tools were available.

Ko et al. [KMCA06] observed ten programmers working on Java maintenance tasks and found that a large part of their effort was spent on other activities than writing code. They observed developers by capturing full screen recordings. The recordings were however not of individual events but in the form of video-frames. This allowed the researchers to see what happened even if the editor was not in focus. The programmers in the study either started their programming tasks by conducting a text-search, or by navigating files in Eclipse. They used cues like class-names, variable names, or features in the interface to find their way in unknown code. Unfortunately 88% of these searches did not lead anywhere. In fact, 36% of the developer’s time was spent on searching and inspecting code that was irrelevant for their current task. When the programmer finally found a relevant part of the code, it was explored and its dependencies were navigated to other relevant parts. When the programmer had accumulated enough information a change could be introduced. Similar search behavior has been found by Maalej et al. [MTRK14]. They conducted a study with the aim to find out how developers comprehend software systems and how developers find and share knowledge about software. They observed 28 professional developers behaviors, strategies, channels of information, and tools. The developers were observed during 45 minutes of programming during which programmers used a think-aloud protocol. The developers talked aloud what they were doing and what they were thinking. Think aloud gave Maalej et al. [MTRK14] even more information than just what happens on the screen. The programming session was followed by an interview to allow the researchers to ask deeper questions on what they had observed. They observed the programmers copying code to avoid breaking existing code instead of adapting and refactoring the old code. The programmers used pragmatic comprehension strategies to avoid unnecessary comprehension. For instance, they used knowledge about the architecture of the system to find a point to search from.

During the programming sessions Maalej et al. [MTRK14] found examples of active information search strategies. The programmers stated hypotheses about the system and then tried to test or answer these. For example, the programmers were observed to intentionally create compiler errors to find im-
portant dependencies in the code without reading it. Maalej et al. [MTRK14] also conducted a complementary survey with 1477 responses to triangulate the findings from the observational study. The survey revealed that the developers rather spent time with the code than documenting it, and that they valued source code over documentation. Code standards like naming conventions, style guides, and common architecture were said to help aiding comprehension, while abbreviations, meaningless, or cryptic names were reported to hinder comprehension. The developers reported that semantic names help to understand the meaning of code but if semantic names are misused they can lead to misunderstandings. Instead of focusing on documentation the developers frequently used colleagues but also external sources to find and share knowledge. Maalej et al. [MTRK14] conclude that developers rather would get the task done than to comprehend the software. They settled for non-optimal working implementation, since an optimal adaptation would require a much deeper understanding of the requirements, code, and how to test and was deemed too risky and costly.

We believe that the opportunistic behavior seen by Maalej et al. [MTRK14] may be enhanced when the students know the assignments are automatically assessed. The students just want to get the job done. A student that settles for an incomplete solution may spend less effort than a student that is more motivated. This would make it hard to compare their effort. For this reason we think that it is important to have a test suite that restricts the student’s solutions. We also believe that in order to understand the students information collection strategies we need to capture the reading and navigation behavior.
Chapter 3

The CSQUIZ Learning Environment

The Computer-Science-Quiz (CSQUIZ) is a programming and learning environment. In short CSQUIZ automates tutoring of programming, collects information from the students programming process, and can automatically conduct distributed quasi-experiments.

Figure 3.1: CSQUIZ is distributed programming and learning environment. It was designed to allow experiments to be automated (gray) but also to automate tutoring and increase the flow of information from students to teachers (white).
3.1 A Distance Learning Platform with an Online Programming Environment

We wanted to enhance the course “Web development with PHP” with a Learning Analytics tool. Figure 3.1 shows a collection of main requirements for CSQUIZ. In the course we wanted to automate tutoring of programming and create high level statistics from that. We also wanted to be able to replay students programming sessions. To answer our research questions we needed to conduct experiments. But since we lacked control in our experiment in 2013, we needed to measure students effort more accurately. Ko et al. [KMCA06] found that the time programmers spend editing was a smaller part of the entire effort. To increase the accuracy we wanted to capture more events than just editing events. For example mouse movements and other events that indicate students are working.

We decided to implement our own tool since it was judged that implementing mouse-level events and to provide PHP-support in an existing tool would be both cumbersome and hard. Mouse events has a much higher frequency than other event types, so extending a project designed for submission or compilation level data was considered too risky. We had an earlier project with in-browser code-edit functionality with color-coding, submission, and test suites that we could build on.

Before CSQUIZ, students of the course “Web development with PHP” had to install tools and servers before they could start working on our programming assignments. This was a bottleneck for many students and they had trouble getting started. We designed CSQUIZ as a web-application so that the students may work within their web-browser and provide easy access for online students. One potential drawback with this design decision is that the student must have a working internet connection. However this is a prerequisite for taking the programming course online and onsite students have free wireless access on campus.

When the students start CSQUIZ they are presented with an overview of the assignments as can be seen in Figure 3.2. The students unlock assignments by completing a prior assignment. A student may return to a completed task to review or change a previously submitted solution.

When a student starts an assignment, CSQUIZ presents instructions, relevant theory with code examples, and source code files to work with as in Figure 3.3 on page 20. The instructions may contain any HTML5-content such as images or video. Clicking on any file link brings up the code editor for that source code file.

The code editor shown in Figure 3.4 on page 21 was implemented using a JavaScript component called CodeMirror\(^1\) with syntax highlighting. CSQUIZ supports multiple files so that students can practice code maintainability scenarios with multiple classes. The navigation between the files is responsive and quick since they are pre-loaded into the client. Syntactical errors that can be detected from a static analysis of the current file is directly shown in the editor.

\(^{1}\text{www.codemirror.net}\)
CSQUIZ

My tasks

A1. Task 2: Errors
Start task...

A1. Task 3: PHP Tags
not available, please finish the previous task first.

A1. Task 4: PHP Variables, getters, and setters
not available, please finish the previous task first.

A1. Task 5: Foreach over an Array (Part one)
not available, please finish the previous task first.

A1. Task 5: Foreach over an Array (Part two)
not available, please finish the previous task first.

A1. Task 6. var_dump
not available, please finish the previous task first.

Completed Tasks

A1. Task 1: Learn how to use the tool
Completed, go back to task...

Figure 3.2: CSQUIZ presents a series of programming assignments (tasks) to the student. The student can return to a previously completed task.

as in Figure 3.4. This static analysis is run every time a file is edited and syntactical errors show up as red dots in the margin as in Figure 3.4. When the student moves the mouse over the red dot the error message is displayed.

Students may at any point save, run and test their code by pressing “Save & Run”. The students are then presented with output from the teacher’s test-suite. The assessment of the code is done automatically and quick. In PHP it is common to use the api-functions echo or var_dump to debug or to writing variable values to the PHP-output-buffer therefore we present code output. Since CSQUIZ is used in a web programming course this output buffer consists of two parts raw-HTML output and an inline frame (iframe) containing rendered output as it would appear in a browser, see Figure 3.5 on page 22. CSQUIZ also presents feedback from the teacher supplied test suite and from the PHP-interpreter (compiler).

When all teacher-tests pass and no PHP-errors or exceptions are thrown, the student receives feedback that they have completed the assignment and may return to the list of tasks. When all assignments are completed the student is presented with a message that they have passed and are done.

Programming Assignment
1. Modify the GameView, Game, and Gamer classes.
2. Since the GameView needs information from the Gamer and Game objects a set of access methods (getters) must be constructed.
3. The Gamer and Game private members should not be made public!
4. The GameView:toString() method should return a HTML string that represents the Gamer supplied in GameView constructor.

Theory
Create an access method (getter)

```php
class Gamer {
    private $name;
    public function getName() {
        return $this->name;
    }
}
```

Create a string from variables

$name = "Daniel";
$anotherString = "My name is $name"; // will contain "My name is Daniel"

Links

**Figure 3.3:** A task is described with a programming assignment instruction and relevant theory for the task. A list of source files is presented in the top left corner.

### 3.2 Teacher Perspective

CSQUIZ is designed to automate tutoring of programming assignments. Teachers construct programming assignments for CSQUIZ by providing four items:

1. A title is used to present the assignment in the overview.

2. An instruction with relevant theory to describe the programming assignment and to support the student. Any HTML can be used; movies or code-examples can be included as well as links to external resources.

3. A set of hand-out code files that should be given to the students. The first file is always called `index.php` and it is always run when the students run their tests. But the teacher can also supply more classes or files for the student to work with.

4. An automated unit tests suite that is used to assess and give feedback to the student. If the test-suite code either results in a PHP-error, output, or in an exception thrown; the information will be presented to the student.
Just like Spacco et al. [SHP+06] CSQUIZ may have public, private, and secret tests to vary the amount of feedback to the student.

The four items are placed in a folder and the name of the folder is used as a key for the programming assignment. The teacher can specify in which order the students should receive the assignments. Different assignments can also be randomly assigned to different students. Today this is done by pairing an assignment with all variants. For example: A teacher may configure CSQUIZ so that the students first receive the same assignments 1, 2, 3. The fourth assignment may have two versions 4.1 and 4.2 and CSQUIZ randomly distributes the students to a version.

The teacher has two main views to monitor how the student group is doing. An activity view and a teacher view are used to extract useful information from the students programming efforts. The teachers get to know how much average time students spend on assignments, and which assignments students struggle with. With this information comes the ability to make informed changes if needed.

The activity view was introduced in 2015. We wanted to see which students that were working at any point in time. From the activity view in Figure 3.6 on the following page, the teacher can see the last day of student activity. A list of students who are actively working on an assignment show up here. Only students who are working with an unsolved assignment show up on the list.
Output

<table>
<thead>
<tr>
<th>Test errors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code did throw an exception when it should not with message: [The getTotal returns [0] should output [0] for [0,1,2]]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sum is <code>&lt;strong&gt;</code>4&lt;/strong&gt; and the lowest item is <code>&lt;strong&gt;</code>9233728369475807&lt;/strong&gt;`</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Browser output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The sum is <code>9</code> and the lowest item is <code>9233728369475807</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code errors/exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception: (This line should never be called) File: /index.php Line: 45</td>
</tr>
</tbody>
</table>

**Figure 3.5:** Three types of output from a test run in CSQUIZ. Test errors are feedback the automated test suite supplied by the teacher. When students write to the PHP output buffer the result ends up in the Code and Browser output. Syntactical errors and exceptions are presented in “Code errors/exceptions”.

**Teacher Menu**

Show Activities Show Report

**Activities during the last day**

<table>
<thead>
<tr>
<th>Student Name</th>
<th>Activity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0_1_compile_error</td>
<td>Working (1 hour ago)</td>
<td></td>
</tr>
<tr>
<td>views</td>
<td>Working (26 minutes ago)</td>
<td></td>
</tr>
</tbody>
</table>

**Not active users**

<table>
<thead>
<tr>
<th>Student Name</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.6:** Activity view showing students who are working on an assignment and students who has not yet started CSQUIZ. Blurred out information is student name and email.

This has proven useful in tutoring. When a student asks for help the teacher may choose to inspect the current state of the students code by clicking a link.

The teacher may also see which students that have not yet started CSQUIZ. They are listed as “Not active users” in Figure 3.6. In 2015 we sent an email to students who did not start the assignment a week before the deadline. We believe that it resulted in more students that started the assignment in time to complete it. Of 85 students, 3 students never started CSQUIZ or any other activities in the course, and 2 students started too late to complete the assignments before the deadline.

The teacher view show all students who are registered on the course. In Figure 3.7 we blurred out the student name and email, but we can see which assignments they have passed, when they passed, or if they are currently working with one. As in the activity view, the teacher may inspect a working student’s latest submission to improve the communication during tutoring ses-
**Teacher View**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0_1_compile_error</th>
<th>1_tags</th>
<th>variables</th>
<th>5_foreach_part_one</th>
<th>5_foreach_part_two</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>2015-09-04</td>
<td></td>
<td></td>
<td>2015-09-04</td>
<td>2015-09-07</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2015-09-01</td>
<td></td>
<td></td>
<td>2015-09-01</td>
<td>2015-09-01</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2015-09-01</td>
<td></td>
<td></td>
<td>2015-09-01</td>
<td>2015-09-02</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2015-08-31</td>
<td></td>
<td></td>
<td>2015-08-31</td>
<td>2015-09-01</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>2015-09-05</td>
<td></td>
<td></td>
<td>2015-09-05</td>
<td>2015-09-05</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>2015-09-03</td>
<td></td>
<td></td>
<td>2015-09-03</td>
<td>2015-09-04</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>2015-09-02</td>
<td></td>
<td></td>
<td>2015-09-02</td>
<td>2015-09-03</td>
</tr>
</tbody>
</table>

*Figure 3.7:* Teacher view shows a list of all students in the class and which assignments they have completed.

The teacher view also provides statistics on the whole class of students. It shows average time for the assignment, how many students that have completed the assignment, and box-plots of the students time to complete the assignments. We have used the teachers view to understand what assignments that students spend more time on. In 2014 we used the teacher view to try and predict which assignments would be problematic. We did that to see if it made sense to add further instructions (lecture content) on the problematic assignments. We found a strong correlation between the effort of students who completed the assignment during the first day and the students that completed it later [TOEW15].

### 3.3 CSQUIZ as an Experimental Tool

When designing CSQUIZ we wanted to get accurate measurements of student effort. In 2013 we conducted an experiment where student measured their effort themselves. That experiment lacked control over the measurement situation and had too low accuracy and precision. This resulted in very large variations and little means of investigating the causes behind the variation. Today we let CSQUIZ automatically measure the students effort for us by collecting many different events from the programming session. This allows us more control and also allows us to inspect the recorded events. We can in detail study causes of individual differences between students and are able to replay the programming session from the recorded events.

CSQUIZ collects all student interactions to provide teachers with detailed information for each student. This allows for both generating statistics and replay of a student’s problem solving effort. The recorded material consists of
log files from the programming sessions. The log files are named using a unique identifier for the student and the name of the task. The log files describe events with time-stamps. There are a number of different events.

- Running the application and the resulting output and error messages from compilation or tests.

- Changes to the source code, which file, what type of change (writing, removing, cut and paste), the position of the change, and the changed characters. Each key stroke is logged.

- Reset file events, when a file is reset to its default value (all changes are removed).

- The text-caret position in the text file. The text caret is sometimes called text cursor indicates the position of where text may be inserted.

- The text selected, both start and end position in the text.

- Loading and reloading of the application.

- Switching between different files, and which file is currently active.

- Mouse movements are recorded at eleven samples per second, position of the mouse relative to the editor.

- Mouse clicks are recorded (added 2015).

- Activity/passivity and if the browser is in focus or the editor is faded or visible.

To detect passivity we fade the editor after fifteen seconds of inactivity as in Figure 3.8 on the next page. The student has to move the mouse or a key to reactivate the editor and make the text visible again.

### 3.3.1 Replay of Recordings

A recording can be replayed using CSQUIZ, this way a teacher may inspect how a student solved an assignment. The replay can be paused and played back in different speeds. The replay is intended to be similar to what the students saw in their browser. Movements of both text-caret and mouse are replayed. The mouse pointer is colored red to distinguish between the researcher’s mouse and the replayed mouse. When the mouse-button is pressed the mouse icon becomes hollow. Events in the editor like text-selections and text edits are all replayed as if watching a screen recording. Files are automatically swapped so that information-collection from different files can be followed. This allows the viewer to for example see that the student copied code from one file and pasted it into another. Longer breaks are automatically fast forwarded. The time of recording is also replayed.

Figure 3.9 shows the replay interface with buttons for play, pause, speed up, and speed down. A recording cannot right now be played backwards. The
Figure 3.8: The CSQUIZ programming area fades after fifteen-seconds of inactivity, allowing us to put a time cap on the reading activity. Top: the programming editor during active use. Bottom: the programming area after fade.

area below the gray is the same or very similar output on what the student saw with a small exception: Since mouse is recorded at “only” eleven samples per second it sometimes appears to jump forward if the student moved the mouse really quick.

When playing the recording a timeline is animated so that we can foresee events in other files. The recorder also shows current location in the timeline and time stamp for when the recording occurred. Using the detailed information available in recordings we developed a timeline-image as displayed in Figure 3.10 that can be used to inspect an entire recording. We use this visualization to give the viewer of the replay a sense of what will come next.

From the timeline we can show activities in different files and what type of events that were recorded in those files. Each file has its own horizontal row. Events that occur in all files, such as breaks and compilation events are shown as vertical lines. When the student press the Save & Run button the code is compiled (interpreted), tested, and executed. A successful run shows up as a green vertical line in the right hand end of the timeline. The red vertical lines are failed compilations that resulted in PHP-errors, e.g., syntax errors. Since PHP is an interpreted language even some runtime-errors show up as red lines. The vertical line is drawn in orange if the test suite detects an error. Fat gray vertical lines indicate that the student took a break and the editor has faded. We print the duration of the break on top of the break lines so that the viewer can distinguish between short and longer breaks as can be seen in Figure 3.11. Longer breaks than a minute are depicted as darker lines, and really long breaks over fifteen minutes show up as red fat lines.

When a student work in a file the events from mouse, text-caret, edits and text selections are drawn on that line as can be seen in Figure 3.11. The close-up shows activity in a single file. The top row shows visibility information,
Figure 3.9: Programming sessions can be observed after they were recorded using CSQUIZ replay functionality. The gray top area shows a timeline and control buttons for the replay, below the gray area is the editor as seen by the student showing student mouse in red. Below that is the output area in green showing error messages as when the students saw them.

Figure 3.10: A CSQUIZ timeline of a recording. Each row is a file. Different colors indicate recorded events like mouse or edits.
Figure 3.11: Close-up showing activity in a single file from the timeline. Top row indicates visibility events, second row is interactions by mouse and text-caret movements, third row contains text selection events and fourth row code edits.

white is activity in other files, dark gray is that the editor starts to fade while light gray is that the text is visible. The second row shows interaction events with mouse events in green and text caret events in gray. Third row shows text selections (blue) and pastes (orange). The fourth row shows code edits (yellow) and deletes (red).

Figure 3.12: Timeline for a student solving the assignment we later decided to divide into two smaller assignments.

From watching a replay of a programming session we get much better idea on how the student worked. As an example we have watched a recording from a student from 2014 that did the assignment we later decided to change in the third experiment in Chapter 4.2.4. We watched the recording and broke down the student’s session into eight steps below. The timeline for the recording can be seen in Figure 3.12.

1. First the student reads instructions and interprets the code in *index.php*. The student runs the tests without doing any changes to the code (the first orange vertical line).

2. The student begins coding by implementing the constructor of the class described in the instructions. After the constructor the student continues with the first method of the class. In the method he starts writing a foreach statement. He stops in the middle of the statement, returns to the instructions, and checks the foreach example given there.

3. The student returns to *index.php* and completes the foreach statement and the method but accidentally introduces a bug. Without testing the student continues with the third method. The method is ended with a test-run that results in an error. The error specifies that an exception must be thrown from the third method.

4. The student becomes inactive almost directly after the error and takes a four minute break (vertical dark gray line). During this time the editor fades completely. The break is followed by a quick mouse movement, and then second shorter break (vertical gray line).
5. The student returns to the code, and writes code that throws the missing exception. Unfortunately the student introduces a syntactical error that was discovered when testing (red line). He fixes the error, retests and get a new error (orange line). The new error indicates that an illegal argument exception should be thrown from the constructor if the arguments contain faulty data.

6. The student adds new code into the constructor to throw the new exception but accidentally makes the exception to be thrown every time.

7. The student spends time debugging the frequently thrown exception, rewrites, and simplifies the code for the constructor with frequent test-runs (orange vertical lines). Finally gets the constructor working and retests. This time he gets an error that indicates the bug he introduced in step three.

8. The student fixes the bug introduced in step three, and finally retest with succeeding tests (green vertical line).

From the story we derive that the student understood the instructions, could successfully use exceptions, and possessed other skills he was supposed to learn in this particular assignment. However we learned that the student needed to look up how to syntactically write some statements. This was one of the faster students on this assignment and the students session was very uncomplicated compared to many other students. This student first wrote most of the code in a big chunk (step 2 and 3), and then used the automated unit test to test his way to a correct solution. This seems to be a strategy deployed by some students. Watching replays allows us to identify such strategies.

### 3.3.2 CSQUIZ Experiments

We wanted to make it easy for the researcher to construct and replicate experiments, so that they can be repeated with little effort. The experiments are generated in the same way as the programming assignments. They have a title, a description, code, and a test suite. All of these may differ between the treatments. The experimental assignments can be ordered to occur between or before the ordinary assignments.

CSQUIZ will automatically randomly assign students into the experimental groups. It can also be configured to conduct more advanced multiple step experiments for within-subject-design. CSQUIZ automate experiments by collecting participation information, dividing students into experimental groups, and it also records the events from the sessions.

Currently the results presentation is done by writing a specialized view class that extracts and presents the information. The data can be viewed in graphs or in tables. The data presented in this study was extracted into .csv files and imported into statistical software.
CSQUIZ has three components, a client web browser application, a CSQUIZ web server, and a sandboxed server. The code $C$ produced by the student is executed on a sandboxed server together with the teacher created test-suite $T$. CSQUIZ writes events to a log file, and an answer is returned to the student.

3.4 CSQUIZ Architecture

CSQUIZ is divided into three main components, see Figure 3.13. The student interacts with a client written in JavaScript, HTML5, and CSS. We use the JavaScript component CodeMirror as a text-editor and it is configured to provide syntax highlighting. The JavaScript application communicates with the server asynchronously by AJAX commands using HTTP. Different browser events results in AJAX calls to the server for logging.

The complete student code is submitted to the server when the student clicks Save & Run. On the server the code is combined with test suite and written to sandboxed server. The sandboxed server is a web server configured so that the student script cannot do any mischief. It runs both the index.php code and the teacher generated test-suite on the student code. The test feedback and output is returned to the browser, formatted, and displayed to the student. The result is presented as both HTML code output and rendered HTML in an HTML-iframe. JavaScript is not allowed in this iframe for security reasons.

Both the teacher and the researcher views requires authentication. Students access their session through a unique identifier that each student is assigned from the course web page. This identifier is a twenty character long hash-string. The teacher has access to information from all student activities with identifiable student names. In researcher-view the researcher only sees student hashes and not their names.

The security-restrictions on the sandboxed server are many. The sandboxed web-server runs with a separate user that has limited access. The sandbox user does not have write-access to the file-system, network, or databases. The execution time is limited to three-seconds to make sure one student does not use all CPU resources and to kill infinite loops. The memory limit is set to
32 Mb per script. The sandboxed server is only accessible from the CSQUIZ server. We block usage of a lot of the API-functions and classes in order to force the students to write their own solutions and not use the PHP API. This is done by blacklisting functions in the PHP configuration file `php.ini`. To capture errors messages we wrote a PHP error handler, capable of capturing errors, exceptions, and syntactical errors.

A number of design decisions were made to speed up the execution of CSQUIZ. We use log-files instead of a database since append-writes take very little time compared to database queries. We use a cache that saves all code-test-runs since it is common for students to recompile with little or no changes to the code [Jad06]. The first time a script is executed it runs in the sandboxed server, the second time the same script is executed we collect it from the cache. We have had no complaint about sluggishness of the system except from students with slow internet-connections.

The CSQUIZ server component is written in object-oriented PHP using Model-View-Controller architecture. Students, teachers, and researcher-roles has different controller-classes that represents their separate use-cases. Most of the code is located in view-classes that generate HTML and JavaScript from the Model classes. We aimed to keep the Model classes free from View specific functionality.
Chapter 4
Methodology

To answer our research questions we use quantitative methods like descriptive statistics, experiments, and statistical tests to compare the effort of groups of students. We do this to contrast and complement discussions with smaller groups of students in our programming course. With CSQUIZ replay feature we are able to observe what happens during a student’s programming time. We have in some isolated cases used this feature to contrast or validate the quantitative results with qualitative observations.

We have divided our effort into three parts. First we evaluate the accuracy of CSQUIZ as a measuring tool in Chapter 4.1. We evaluate CSQUIZ by comparing its accuracy in estimating effort to other tools. We also explore how to separate the time students are active from the time they are working on other tasks. Second, we use CSQUIZ to measure effort in three experiments. If the tool is useful we should be able to use it to observe small changes and be able to compare different versions of the same programming assignments. In the two first experiments we observe differences in effort when we introduce source code comments and the third experiment is a quasi-experiment where a programming assignment is changed and we use CSQUIZ to determine if the change was successful. The experiments are described in Chapter 4.2.

4.1 Evaluate Measuring Tool

RQ1. How can student programming assignment solving effort be automatically measured with high accuracy?

To answer the first research question we divide the effort into activities and separate activities that students do to solve the assignment from activities that are spent on other things. In Table 4.1 we divide the effort into activities that are done in the tool from activities conducted outside of the tool. When students use the tool to write their code, CSQUIZ can measure the Editing ($E$) effort. Students also interact with CSQUIZ in ways that do not change the code. They select text, flip between source code files, move the text-caret, and move the mouse pointer over the editor. We call the effort spent on such interactions Active Use ($AU$). By fading the text on inactivity CSQUIZ can
Table 4.1: Dividing the effort of working on a programming assignment into measured activities. The measure Passive Use captures both reading activities in the tool but may also capture time spent on other activities.

<table>
<thead>
<tr>
<th>Working on assignment activity</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing code, paste</td>
<td>Editing (E)*</td>
</tr>
<tr>
<td>Navigation, scroll, read, select, copy</td>
<td>Active Use (AU)*</td>
</tr>
<tr>
<td>Reading and reflecting, without interacting</td>
<td>Passive Use (PU)**</td>
</tr>
<tr>
<td>Work in other applications.</td>
<td></td>
</tr>
<tr>
<td>Working on paper</td>
<td>Text faded (OT)**</td>
</tr>
<tr>
<td>Discussing task</td>
<td></td>
</tr>
<tr>
<td>Draw on whiteboard</td>
<td>Tool closed (OT)**</td>
</tr>
</tbody>
</table>

determine the difference between the time text is visible and the time when the text has faded. If the editor is open with text visible we call the effort Passive Use (PU). It could be that the student was reading the code in the editor, but the student could also be doing something else. The fading timeout was set to fifteen seconds. The time was set by testing CSQUIZ with colleagues and former students and fifteen seconds was determined to be a good middle-ground between giving accurate activity measurements and not to disturb the programmer.

A student may be active and work on the assignment but not using the computer. For example: working on paper, discussing with other students, or draw graphs on a whiteboard. The student may also use other applications or searching for information in other sources. We call the sum of those Out of Tool time (OT). Since there is no interaction during the Out of Tool time we cannot determine what happened. We hypothesize that if the time is in close relation to time spent inside the tool that students are working on the assignment, but we really cannot tell.

There is uncertainty to all our measurements of student effort, we cannot be certain that an interaction means that the student is working on the assignment. If the user is moving the mouse over the editor we will capture that as a sign of Active Use even if the student did not work on the assignment. The students may also edit the text by mistake, or start the tool without intending to work with the assignment.

The tool CSQUIZ was designed to capture more of the students’ effort than other tools by implementing more intrusive instrumentation. This was done to capture the Active and Passive Use time of the student effort. However, extra instrumentation comes with a cost. Logging more events increases log file sizes and the decision to fade the text on inactivity do cause inconvenience for students. Therefore it is relevant to ask if the extra instrumentation is useful.

To validate the usefulness of our increased accuracy we divide the effort into activities. If the Editing activity is the dominant part of the total programming effort, we may be able to build a simpler time aggregation model based on a
less intrusive granularity and still get accurate results.

To calculate effort, we first need to aggregate time from the discreet events we collect.

4.1.1 Time Aggregation Models

A time aggregation model is used to calculate time from a log of discrete events. Each event has a timestamp so that they can be ordered. The model uses a threshold \( T \) to group events that are closely related in time into series of events. In Figure 4.1 we connect events that are closer than the threshold into separate series of connected events. The first three events are connected into a single series \( s_1 \). The gap to the fourth event is longer than the threshold so the following two events are not connected to the first series. Instead we form a second series \( s_2 \) with two events. The last event is separated by more time to other events than the threshold, and becomes a separate series on its own \( s_3 \). A series with just one event is counted as 0.5 seconds. We compute the time for a series with more than one event as the period from the first events to the last event in the series. The effort is calculated as the sum of the times of all series.

\[
\sum_{i=1}^{n} s_i
\]

4.1.2 Granularities

CSQUIZ collects many event types that are also collected by other tools. By using a subset of our event types we can compare the accuracy of different granularities. Ihantola et al. [IVA+ p] found six levels of granularity in a recent systematic literature review of tools that capture the programming process. From the coarsest granularity that only collects submission events to the finest granularity that collects each key stroke. These can be viewed on the right hand side of Figure 4.2 on the following page.
Figure 4.2: We merge granularities from Ihantola et al. [IVA+ p] into G1 and G2 and add two new granularities. Each finer granularity level includes all events from the coarser so G2 includes all events from G1.

In CSQUIZ a save is always followed automatically by a submission, compilation, and a test suite run. The student must actively run this by pressing (Save & Run). We therefore merge the Submissions, Executions, Compilations, and File Saves granularities from Ihantola et al. [IVA+ p] into one single granularity that we call Compilation (G1). The Compilation granularity is our coarsest level. We include application start-up events in the Compilations granularity since in many of our recordings there are just a few compilations in the end of a session. When we include a startup event we capture more of the students’ effort. This level is comparable to tools like ClockIt [NBFJ+08; FNB+09] but is on a finer granularity level than the Retina tool [MKLH09]. Since individual key stroke events are recorded by CSQUIZ, we let the second Key Stroke (G2) granularity capture those. This level is comparable to Matsuzawa, Okada, and Sakai [MOS13] who record individual key strokes.

We add two new granularity levels and compare these to Compilation and Key Stroke. In our third granularity level Interaction (G3) we include events from mouse movements, text selections, and mouse caret. The fourth and finest granularity Visibility (G4) adds application visibility events so that we can determine if the application has become inactive or that the text has faded. Each higher granularity level includes all events from all lower levels of granularity. For example Interaction granularity includes all events from the Key Stroke granularity. We will use this property to divide the effort into different activities.

4.1.3 Dividing Time into Activities

We derive the four measures of the programming assignment effort from Table 4.1. First we measure the effort spent editing code. This time can be accurately measured by all tools that capture individual key events or saves changes with a high frequency. In Table 4.1 on page 32 this is the Editing time. Secondly, we want to measure how much of the programming time stu-
udents spend reading and navigating the source code. In Table 4.1 this is a combination of the time for Active Use and the Passive Use. This is the time other tools either miss or must approximate using a larger threshold. Note in Table 4.1 that the Passive Use might also capture a bit of time spent on other activities outside of the tool. We will therefore present the time for Passive Use separately. Lastly we want to estimate how much time students spend Out of Tool. In Table 4.1 time spent Out of Tool is a combination of the time for when the text area has faded and the time when the application is closed.

By combining different time aggregation models, granularities and use different thresholds we derive the time for these activities.

We begin by computing a Total Time (TT) representing both the Time in Tool (TiT) but also include the shorter breaks. We compute Total Time that by measuring on the Visibility granularity level but set the threshold to fifteen minutes (T : 15m). A threshold of fifteen minutes means that we will overestimate the effort. We will thus capture the short breaks that students spend outside of the tool but returns to working in the tool within fifteen minutes. We know that students use online resources and course literature to find answers to their programming questions. We selected fifteen minutes since it seemed the longest time we would consider a student need to collect information.

\[ TT = G4(T : 15m) \]

Next we compute the Time in Tool (TiT) by using the Visibility granularity again but with a three second threshold. This is the time that students spend in the tool with the text visible. This is thus the combination of time for Editing, Active Use and Passive Use.

\[ TiT = G4(T : 3s) \]

We subtract the Time in Tool from the Total Time to get the time Out of Tool. To compute a fraction we divide by the Total Time. A high fraction of time spent Out of Tool could indicate that the students either get frequently interrupted or needs to collect much information from other resources to solve the assignment. The fraction time spent on the assignment but Out of Tool is calculated as:

\[ OT = \frac{(TT - TiT)}{TT} \]

Next, we compute the fraction of time spent editing the source code using the Key Stroke granularity with three seconds threshold. By dividing the time for editing with the total time we get fraction of time spent Editing.

\[ E = \frac{G2(T : 3s)}{TT} \]

We compute the time spent interacting with the tool on Interaction granularity with three seconds threshold. Since all events in Key Stroke granularity is included in the Interaction granularity we can subtract the Editing time and
end up with only the time exclusively spent on tool usage that did not change the code. We call this fraction Active Use.

\[ AU = \frac{G3(T : 3s)}{TT} - E \]

Finally, we compute the fraction of time the students spend passive but with text still visible. We can do this since we have derived both the Editing time, Active Use time and Out of Tool time, the time remaining is the Passive Use time, and again we compute it as a fraction.

\[ PU = 1 - (E + AU + OT). \]

Since CSQUIZ does fade the working area on inactivity we are able to put a definite time cap on the time students spend inside the tool. During the Passive Use time we do not know what the student does. We only know that the tool is active and the text is still visible, included in this time is all small breaks between other events that are longer than three seconds and shorter than fifteen seconds. The true time for reading and navigating may be as big as the sum of Passive Use and Active Use but not longer.

To determine the size of Out of Tool, Editing, Passive Use, and Active Use we do an empirical study using data collected by CSQUIZ from two years of tutoring experience. These values are presented with descriptive statistics and compared to existing research.

We calculate these for each recorded programming session in both 2014 and 2015 and present average fractions in percent with median, min, max, and standard deviation values. These average fractions will thus be computed from a combination of many students and assignments and should represent average fractions of time. To see if the values vary on different assignments we compute the different values for each of the assignments in 2015.

### 4.1.4 Comparing Granularities

|--------------|------------------------|----------------------|----------------------|----------------------|

\[ G1. \text{Compilation, T:30m} \]
\[ G2. \text{Key Stroke, T:3s} \]
\[ G3. \text{Interaction, T:3s} \]
\[ G4. \text{Visibility, T:3s} \]

**Figure 4.3:** Estimating student effort from collected events. Events are aggregated into estimations (gray boxes) of the true effort (white boxes) using different granularities and thresholds (T). A better estimation model can be created when more events are collected.
Even if the reading and navigation is a large part of the programming effort, it could be that we can capture the effort accurately with more coarse grained models. It depends on if the editing events are evenly spread out over the time that students work, and if larger gaps indicate that the students are not working. If editing events are instead collected in small lumps with larger gaps we need a finer granularity to accurately estimate the effort. We suspect that editing comes in short bursts of events with larger gaps. Therefore it is important to test this with empirical data from real programming sessions.

We give examples of how events can be aggregated in Figure 4.3. In the top row of this fictional example we present the ground truth of how a student spent his effort. White boxes indicate three periods of activity divided by two breaks. The rows below are examples of time aggregation models on different granularities and thresholds. The gray area indicates the effort estimation. In the second row we have three Compilation events that are collected with a threshold of 30 minutes. On the third row we add Key Stroke events and use a too short threshold resulting in an underestimation of the students effort. The fourth row includes Interaction events and can better capture the beginning of the session but cannot determine what happened in the gaps. On the last row we measure on Visibility granularity. We can now differentiate between when the text was visible from where the editor had faded. Note that the Visibility fails to capture the first small break.

We determine the CSQUIZ accuracy by comparing fine grained models to models based on lower granularity that require less instrumentation. We first compute CSQUIZ best estimation of the students programming effort. Again we call this period Time in Tool (TiT) and as before we compute it with 3 second threshold. This is our baseline and the best estimation we can do of the time students spend in the tool.

\[ TiT = G4(T : 3s) \]

For each granularity we experiment with different thresholds until we find thresholds that minimize the average error when we compare to Time in Tool. We present the following thresholds: three seconds, ten seconds, one minute, five minutes and fifteen minutes. These thresholds were selected to compare with related work, and to reach an overestimation on each granularity level. We combine each granularity level with each threshold to form twenty different Time Aggregation Models (TAM).

\[ TAM(x, t) = \frac{Gx(T : t)}{TiT} \]

We compare each of these models to the Time in Tool model by computing time estimations for each model using the 1028 recorded programming sessions from 2014 and 1573 programming sessions from 2015. Each time estimation is divided with the Time in Tool estimation to find the fraction of time the model can approximate. We present the average percentage of time and standard deviation for each of twenty models. We would consider an acceptable model to produce accurate effort estimations on average and have a low standard deviation.
4.2 Experimental Design

When student programming effort can be accurately measured we see a possibility to evaluate effects of changes done to an assignment. We investigate the usefulness of using the detailed effort measurements by conducting three such evaluations. We want to see if a change in the assignment caused an improvement, for example a reduced student effort. We use the experimental method since experiments are constructed to find cause-effect relationship. Finding such relationship requires a controlled investigation where the outcome of treatments can be determined [WRH+00].

We decided to measure how the effort changes when comments are introduced to source code. Since we have not seen other tools collect events associated with reading and navigation it should provide a good test of CSQUIZ usefulness.

The existence of source code comments is an interesting feature to investigate since they do not affect the execution of the program. We can thus remove comments and the program will still run in the same way. Thus the usefulness of comments can be manipulated by either containing helpful information, redundant information (same as the code), or we could even put misinformation into the comments.

Comments are introduced in code to help build knowledge of a software system. The general usefulness of comments is not clear. There are studies that report that programmers prefer to ask a colleague instead of reading documentation [MTRK14]. There are also studies on code readability that find little influence of comments on readability [BW10]. In the 1980’s, Tenny [Ten85] conducted programming experiments investigating if source code comments would increase readability. In the first experiment Tenny [Ten85] did not find a significant difference between having and not having comments. In the second experiment Tenny [Ten88] let 148 students read PL/I code with and without comments. After reading, the students answered questions about the code. He found that comments could indeed improve the readability of non-modular programs. Tenny concludes that the comments may have “rescued” a poorly divided unreadable program and made it more readable. However, Tenny specifically constructed the comments to improve the readability of the code [Ten88].

In 2013 we did an experiment on the effect of comments but due to the lack of control and high variation in values we could not see any effect. We got a high variation in the measured values and think this was due to lack of control of the measurement situation. Students measured time themselves, tested manually, used different tools, and reported to be interrupted. We also learned that the comments in the previous experiment contained lots of redundant information that also could be found in the code.

CSQUIZ was specifically designed to measure time more accurately, handle interruptions, and do testing; therefore it is interesting to see if CSQUIZ can accurately conduct experiments on source code comments. Therefore we ask:

**RQ2.** Do source code comments in a programming assignment significantly decrease the effort to solve that assignment?
To answer the question we construct two experiments in CSQUIZ using two scenarios and collect data from students participating in the experiments. The first experiments examine the effect of introducing redundant comments. We would expect the reading time to increase since more text is introduced and all information in the comments also exist in the code. In the second experiment we test what we consider useful comments. Useful comments are designed to help students solve the assignment. We would expect an increased reading time since there is more text to read but a decreased editing time since the comments should help code comprehension. If the students ignore the comments there would be little difference. The two experiments are described in more detail in Section 4.2.2 and Section 4.2.3.

In 2014 one programming assignment was judged to be too hard compared to the other assignments. We decided to divide the assignment into two smaller assignments. The new version of the assignment was introduced in 2015.

**RQ3.** Does dividing a programming assignment into two smaller assignments decrease the total effort of solving the assignment?

A third quasi-experiment is constructed to answer the question and to evaluate the effect of the change to the programming assignment. We call the third experiment a quasi-experiment since it does not use randomization to divide students into experimental groups. This introduces an interesting sub-question: Can two cohorts from different years be compared in effort spent? The experiment is described in more detail in Section 4.2.4.

---

**Figure 4.4:** The student group is divided into two experimental groups and given one version of the programming assignment A or B. Students program until they reach the solution state S. The effort is measured by CSQUIZ.

All three experiences use CSQUIZ as an experimental platform. The two experiments on source code comments are repeated a second time with a new group of students. The experimental design used in the experiments is described in Figure 4.4. One group of students are assigned one version of the assignment called A, another group is assigned version B. Both versions of the assignment have the same solution space S, so both groups must solve the same programming problem. The solution state S is defined by suite of automated unit-tests. When all test succeed the solution space is considered reached and the measurement stops.
4.2.1 Subjects

True experimental design use randomization to select subjects to include in the experiment from a larger population [WRH+00]. This randomization is important since the subjects included in the study should represent the whole population. To be able to generalize a possible finding from the experiment we must determine what population we want to draw conclusions on.

In the source code comments experiments we are interested in the effects of introducing comments to programming students in general. Out of convenience we only include students who participate in the courses we aim to improve. This gives a rather limited possibility to generalize to a bigger population of all programming students. In the third experiment we aim to achieve a change to an assignment in a specific context. The programming assignment is part of a course that is part of a larger context with four different study programs. The general population of that experiment is all students participating in second or third year programming study programs at Linnaeus University. That population gets new students each year. In the case of the third experiment we have a better opportunity to generalize to future students if the study programs stay the same since we sample from the correct population.

Students that participate in the experiments did so in the context of the course “Web Development with PHP”. The course is given on half speed and both groups of students are with few exceptions taking another course at the same time. The students are also with a few exceptions on their second or third year of their study programs. Approximately 15% of the students who are taking the Web development with PHP course are female.

In 2014 the students consisted of campus and online students from two study programs in Kalmar. The online students participated in a “Web programmer” study program. They study all of their courses online and there are no mandatory physical meetings. The second study program was “Developer of digital services” and is composed of campus students. Before starting the course the students have taken approximately eight 7.5 HEC programming courses, including a course in object oriented programming and a course in databases that both must be finished. The prerequisite programming courses has been given in C# and JavaScript. Some students have reported to have experiences with PHP from other courses or work.

In 2015 we got approximately the same classes students from Kalmar but also a group of student studying on Växjö campus. These students took the course as an online distance course with only two campus events. The Växjö students are participating in the “Software Technology” or in the “Network security” study programs and take the course “Web Development with PHP” in their third year. The students from Växjö have mainly used Java in prerequisite courses.

The students who participate in the experiments have freely chosen to participate; we have informed them of the experiments, its goals and given them an opportunity not to participate. Wholin et al. argues that students that choose to be a part of experiments also tend to be more motivated [WRH+00]. If only the motivated students participate, we consider this a threat to internal validity. To mitigate this threat, we tried to make sure students feel confident
to participate in the experiments regardless on programming ability by informing them in text and during a lecture. We explained the goals, the experiments procedure, and the measurement procedure. We did not discuss the exact research questions or the experimental treatments with them. The students did choose to participate by opting in when starting CSQUIZ the first time, and were given a choice to opt-out during the experiment if they so wished. The percent of students participating in the experiment of the students working on the course is presented in Table 4.2. The participation was made without us knowing who did participate or not in the experiments.

<table>
<thead>
<tr>
<th>Course Context</th>
<th>Year</th>
<th>Participation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1dv408</td>
<td>2014</td>
<td>62%</td>
</tr>
<tr>
<td>1dv608</td>
<td>2015</td>
<td>77%</td>
</tr>
</tbody>
</table>

In the experiments we used different ways to randomize students into experimental groups. In the two experiments on source code comments we randomized after students opted in to participate. The randomization was made by CSQUIZ. It is double blind so neither student, or we knew which student got what treatment. In the fifth experiment we did NOT use randomization to divide students into experiment groups since it compared efforts of one class of students with the next year’s class in a change scenario. With a new group of students, changes to the course, and changes in prerequisite courses, we might measure a difference between the cohorts instead of an effect of the change. This seriously affects our ability to make causal claims from the experiment. To mitigate this we will compare the two classes on other assignments to see if there are differences between the groups.

4.2.2 Redundant Comments

In this experiment we examine the research question: Do source code comments in a programming assignment significantly contribute to the effort?

From this we derive two hypotheses, one null hypothesis and one alternate hypothesis:

- Redundant H0. There is no difference in effort between having and not having redundant source code comments in a debug scenario.

- Redundant H1. There is a significant difference in effort between having and not having redundant source code comments in a debug scenario.

The usefulness of comments can be debated and in this experiment we are interested in when redundant information is introduced as comments. We ask this in a debug scenario to limit the amount of code written by the students and maximize the fraction of time needed to read the code. Thus we created a single class with a single method “max” that is intended to return the largest of its three inputs. A bug was introduced into the code by making two small
changes. The bug was created to be “easily” fixed by reading the if-statements in the end of the method and thus not relying on the information in comments. The instruction was “Correct the logical bug(s) in the max method. Use the comments and test results to guide your work”. A test suite makes sure the solution space only consists of valid solutions where the bug has been removed. The tests are automatically run when the student press “Save and Run” and feedback is given in the form of input, actual output, and expected output in the Listing 4.1.

| wrong output [0] for [1, 3, 0] should be [3] |
| wrong output [1] for [3, 0, 1] should be [3] |

**Listing 4.1:** Test feedback giving input, expected output, and actual output.

Before beginning the experiment the student had to complete six programming assignments in PHP. This was done to make sure that all students were used to using CSQUIZ as an editor and had been introduced to all syntactical elements used in the experiment. For example: The use of exceptions, if-statements, return values and comparison operators was practiced in the prerequisite assignments. Also the function “is_numeric” was practiced in an assignment before. This was done to try and minimize the effort spent learning syntax and functions during the experiment, since we are interested in the effect of redundant comments.

We created two versions of the code, A and B. The students were randomly assignment one of the two versions by CSQUIZ.

**Treatment A.** Code is shown in Listing 4.2 on the next page. The code was written to be minimalistic but to contain a few lines of code that tests the input for correct type and input domain. Identifier names x, y, z was chosen to represent the three input numbers. The bugs were introduced in the if-else-if statements. The version A consists of 23 lines of code and no comments.

**Treatment B.** Code is shown in Listing 4.3 on page 44. When writing the second version of the code we started with treatment A and added comments and also type information in identifiers. The comments are designed provide good documentation for the intended implementation but not to reveal the bug. The redundant information given in comments is also available by either reading or running the code, and is thus not needed to solve the task, the question is if the added lines of comments and longer identifier names reduce the readability and thus would contribute to longer assignment solving times. The version B consists of 41 lines of code and comments.

During the experiment we let CSQUIZ measure the students’ effort. From that effort we remove the time students spend reading instructions. The effort is computed using the Visibility granularity with three second threshold ($G_4(T : 3)$). As explained before the coding editor area fades out when students are inactive for more than fifteen seconds. We do include the Passive Use time since we find it is likely the students will stop interacting, and read code for a couple of seconds.
class Math {

function max($x, $y, $z) {
    if (!is_numeric($x) ||
        !is_numeric($y) ||
        !is_numeric($z))
        throw new \Exception("Input must be numeric");
    if ($x < 0 ||
        $y < 0 ||
        $z < 0)
        throw new \Exception("Input must be larger than or equal to 0");
    if ($x > $y && $y > $z)
        return $x;
    else if ($x > $y && $y > $z)
        return $y;
    else
        return $z;
}
}

Listing 4.2: Experiment One, code version A with no comments. The bugs are located in the last two if-statements

Descriptive statistics will be used to show the results and statistical tests will be used to test for significant differences between the versions.

4.2.3 Use a Class With Comments

The second experiment conducted in CSQUIZ also examines comments but differs from the debug experiment in three ways. First, in this experiment the students need to write their own code (instead of debugging). Second, the students only read and do not need to change an existing class. We have two versions of that class, with and without comments. The students write their code in separate file. This will allow us isolate the reading effort spent in one file, from the code writing effort in another file. The third difference is that the comments are written to be helpful instead of redundant.

A first file contains instructions and is the same for both treatments. The second file index.php is also the same in both treatments and contains data and some instructive comments. The students should write all their code in that file. The third file HTMLPageView.php contains a class that the students should study and use. This file exists in two versions. Version A contains 25 lines of code. And the second treatment B contains the same code but with 22 added lines of comments. The comments were designed to be helpful. The comments were constructed as follows:

- The comments consists of English prosaic description of class functionality, member variables, and methods.
<?php

class Math {

/**
 * max() returns the largest value of its three input arguments
 * @param float $numberX A positive number
 * @param float $numberY A positive number
 * @param float $numberZ A positive number
 * @return float
 */

function max($numberX, $numberY, $numberZ) {

    //Make sure only numeric inputs are allowed
    if (!is_numeric($numberX) || !is_numeric($numberY) || !is_numeric($numberZ))
        throw new Exception("Input must be numeric");

    //Make sure all inputs are positive
    if ($numberX < 0 || $numberY < 0 || $numberZ < 0)
        throw new Exception("Input must be larger than or equal to 0");

    //Find the largest of the inputs
    if ($numberX > $numberY && $numberY > $numberZ)
        return $numberX;
    else if ($numberX > $numberY && $numberY > $numberZ)
        return $numberY;
    else
        return $numberZ;
}
}

Listing 4.3: Experiment One, Code version B. code with redundant comments. Same bug as in version A.
- The comments also include type information, return values, and exceptions thrown in the form of PHP-Documentor comments.
- The comments contain usage-examples describing how to instantiate and use the class with its namespace and how to call the method `echoHTML`.

```php
<?php
//use these variables to create your instance of the class!
$title = "My HTML title";
$body = "My HTML body";

//write your code here:
$view = new \view\HTMLPageView($title, $body);
$view->echoHTML();
```

**Listing 4.4:** Experiment Two. The Solution code to the third experiment.

Listing 4.4 shows the file `index.php` with code needed to solve the task. The solution was designed to be simple. To correctly solve the assignment the students should write two lines of code. To do that the students must extract namespace and parameter information from the `HTMLPageView.php` file. In order to solve the task an object must be instantiated and correct parameters must be sent to the constructor. After the object has been created a method should be called on that object. A test suite was constructed to make sure only valid solutions were accepted.

We expect the reading effort to be longer in the file with comments but choose to test both tails. We state the following hypotheses on the effort spent in the `HTMLPageView.php` file:

- Read H0. The effort to extract information from a class does not rely on the presence of comments.
- Read H1. The effort to extract information from a class is significantly longer in the presence of comments.
- Read H2. The effort to extract information from a class is significantly shorter in the presence of comments.

We expect the effort for writing and editing code to be shorter with comments. For example the students could just copy paste the example given. We state the following hypotheses on the effort spent writing code in the `index.php` file where students should write their code:

- Write H0. The effort to solve a task in a use-class scenario does not rely on the presence of comments in that class.
- Write H1. The effort to solve a task in a use-class scenario is shorter if the class used has comments.
Write H2. The effort to solve a task in a use-class scenario is longer if the class used has comments.

CSQUIZ aggregates individual effort spent in the two files using the Visibility granularity with three second threshold. Time is thus only measured when students are interaction with the tool and the window fades and becomes unreadable after fifteen seconds of inactivity. Descriptive statistics will be used to show the results and statistical tests will be used to test for significant differences between the versions.

4.2.4 Informed Change

We used CSQUIZ in the “Web Development with PHP course” in 2014 and identified one problematic assignment. The assignment on foreach-loops took much more effort than we expected, and we got more tutoring questions on it than on the other tasks. When the course ran again in 2015 we tried to improve the assignment by dividing it into two smaller assignments and introducing better instructions. Therefore we ask:

RQ3. Does dividing a programming assignment into two smaller assignments decrease the total effort of completing the assignment?

We want to compare the effort in 2014 ($T(2014)$) with the combined effort of solving two assignments in 2015 ($T1(2015) + T2(2015)$). The design of this research is similar to an experiment in that we use statistical tests to determine significant results. The difference is that students are NOT randomly assigned to experiment groups. This makes it harder to claim causal relationship between a change and the effect. In order to attribute the differences of the two groups as belonging to the change made we need to be able to carefully compare the two groups of students. A second difference is that the assignment was not designed like the previous experiments to avoid learning. We introduce new topics in this assignment. We divided the assignment into two smaller ones to reduce the amount of learning needed in each sub-assignment.

We measure effort as the time taken to solve the assignment.


We also measure the number of breaks ($B$) taken during the assignment(s). Since students in 2015 get a natural break in between the assignments, they might be less prone to interrupt themselves in the middle of an assignment. We believe that students who are not making progress on a complex assignment may take more and longer breaks. Perhaps the number of breaks is a sign of frustration. We set the length of the break to be more than fifteen minutes. This is based on another assumption that students who spend less time than
fifteen minutes outside of the tool may be searching for information and still work with the assignment.


Finally we compare the number of forum posts on the task in both years. In 2014 the course-press forum was used and in 2015 the slack discussion board was used. We see the forum posts as a sign of how self-sufficient the students were.

Descriptive statistics will be used to show the results and statistical tests will be used to test the hypotheses.

4.3 Method Discussion and Validity Threats

With larger student groups and online students we lack the time to communicate with each and every student. We still discuss the programming assignments with individual student’s onsite, but we feel that the students who choose to communicate with us are not always representative for the entire class. To capture the effort of the whole class, and not just individual students we need to collect data from a larger group. The quantitative approach was chosen to complement and contrast the communication with students.

We use experiments to validate the usefulness of accurate measurements. The experimental method forces the researcher to make explicit hypotheses and define treatments that realize those claims. The use of experiments was inspired from code quality and code maintainability research [Han10; KHR+12; HKR+13; KMCA06; DSF09; Pre00; Ten85; Ten88; BW10; MTRK14].

Experiments are often conducted on students with the intention to generalize findings to experienced programmers. We think that students of programming are a population with its own needs. High code quality or instruction quality may be something different for students compared to experienced programmers. Also, there is no sharp line between experienced programmers and students, many of our students have years of experience from working before they start their study programs. Other students have no experience at all and only learn programming the academic way. We think it is the diversity of our students that makes the experiments extra interesting. A negative rejection of the null-hypothesis show us that either there is no effect or that we do not yet fully understand what is going on and need to examine further what lies behind the variation. We can use the recordings to observe student programming sessions in detail. This allows us to inspect sources of variation in the quantitative data set.
4.3.1 Field Experiments

The experiments take place in the students’ naturally occurring environment and not in a controlled laboratory. As such they can be considered field experiments or on-line experiments [WRH+00].

In a field experiment we cannot rule out that students share solutions with each other. In fact we have observed students ask each other or us for help. The statistical tests we use assumes independent samples, if the samples are dependent it may lead to the wrong conclusions [WRH+00]. To mitigate this we may inspect the recordings for copy-pasted solutions, and find if many students share the same solution. In the data from 2014 we inspected the shortest recordings and removed one student from the data set that we suspect cheated since the solution was copy-pasted into CSQUIZ. Without detailed recordings we would not have detected this. We consider the tasks in the two first experiments to be very easy and the students have practiced the needed skills before the experiment. We do not think cheating and dependent samples to be an issue in those. In the third experiment we think sharing of solutions might be a much bigger issue in 2014 since the assignment was deemed to be hard by us and the students.

Another problem with field experiments is that we might capture effects that are due to other factors than our treatments, like students who are working during the night might be slower than students who are working during the day [WRH+00]. In experiment three the experimental setting between groups is probably very different since a year has passed. In that experiment we take extra precautions to test for differences between the two groups using measurement from other assignments.

We choose to do field experiments since many of our students were online students, and we cannot easily bring them to the laboratory. Field experiments suffer from a lack of control [KC07]. We cannot control for time of day, distractions, and interruptions. CSQUIZ allows us to reclaim some of that control. We know the time of day, we may detect interruptions, and we may also identify students who cheat. Since students were randomized into experiment groups we assume that such effects are evenly distributed between the groups.

Field experiments were also chosen to lower the cost of each experiment both for us and for the students. The effort for each student to participate in an experiment becomes much lower when they can choose the time and place. Automation of experiments enables us to do more experiments and allows easy repetition of old experiments. The extra effort of repeating the two first experiments in 2015 was very small. Field experiments are also considered more realistic than an experiment in laboratory setting [KC07].

Finally we do field experiments since we want to understand and improve a specific course context. Conducting experiments in a course context makes it hard to reproduce the exact conditions to repeat the experiment. The course context changes over time, and so do the student groups. CSQUIZ allows us to compare and study these changes by comparing the relative effort of classes on assignments that does not change.
4.3.2 Mortality

When conducting experiments the students may choose to drop out of the experiment at any point. Wohlin et al. [WRH+00] calls this “mortality” and it is a threat to internal validity. Mortality does not have to cause a problem if it happens “randomly” distributed among the experiment groups, but may be an issue if the mortality is connected to the experimental treatment. For example if students who receive the “worst” version of an assignment fail to complete the experiments. We never intended to cause mortality due to giving out an impossible task. In all the experiments we made sure that students would have the information needed to complete the assignment. In the experiments conducted using CSQUIZ we can see the amount of work done by the student before quitting. We are able to differentiate if they started and how long they worked before quitting. We know that we had no mortality in the first two experiments. In 2015 we did have two students who failed to complete the third experiment before the deadline. We do not think this will affect the experiment since both students started working on the task close to the deadline and simply ran out of time.

4.3.3 Ethical Considerations

The vast amount of data collected by Learning Analytics tools provides opportunities for research. We have made sure that students understood that data is collected. Before the experiments and the data-collection took place, students were informed both in text and in person during an recorded introduction. We informed them about the intent of the experiments, the extensiveness of the data collection, how this information can be used by us, and also how we intend to share it with other researchers. We also informed the students that it is voluntary to participate in the experiments and that they can refuse to participate without any consequence. In fact we designed CSQUIZ so that the teacher would NOT get information on who participates or not. To be extra careful we also delayed some of the data-extractions until after course grading.

Greller and Drachsler [GD12] sees a risk with data that contains hidden structures, for example relationships between students that may lead to false correlations. Competence is needed in order to critically interpret data otherwise we risk that decisions are made from a too simplistic representations of the underlying complex reality. We have considered the dangers of abuse of the data, and made changes in how we present our data for teachers. For example, when we designed CSQUIZ we believed it might help the teacher to know which students who struggled and spent long time on assignments. Therefore in 2014 we allowed the teacher to get access to estimations of each student’s effort. We learned how easy it was to jump to conclusion based on such oversimplified statistics. As an example: A long programming session does not necessarily mean low skill, it could very well be a long session of experimenting, or an attention to details. In 2015 we removed that feature since we thought it might affect the teacher’s judgment on individual students. In 2015 only information on the average performance and spread of the group is displayed together with date of completion for each individual student. We
still find it interesting to identify students who struggle but when they have completed an assignment they should not be singled out. For the same reasons, when trying to improve the assignment in experiment three we looked at randomized anonymized recordings since it felt very personal to watch so detailed recordings.

Even with voluntary participation there are ethical considerations that must be made with extensive data-collection. Greller and Drachsler [GD12] identifies a number of challenges for using such data. First, the lack of open data is a problem for evaluation of Learning Analytics methods. If data is going to be shared openly it must first be anonymized. Open data also need format, policies and extra documentation. To make the data easier to store and share we worked with guidelines provided by Swedish National Data Service\(^1\) to describe our data and plan for storing and sharing. The custom data format used by CSQUIZ is not ideal for long time storage since the data must be extracted with a compatible version of CSQUIZ. This means that a version of CSQUIZ must be stored together with the data. At Linnaeus University there is an ongoing process to facilitate storage of digital research data that we contribute to. Research data is governed by laws and regulations. In Sweden we have the Personal Data Act\(^2\) that governs how personal information may be stored and used. We consider the collected information as not being personal information. However, since we collect free text input; students may by accident add personal information to their log files. To mitigate this risk we asked them explicitly not to do this, and also that they later were allowed to opt-out should they wish for their records to be deleted. No students have asked us to remove them from the data. However this can only be done before any publications has been made on the data. The student names and email is not included in the data, instead they have a twenty character long unique randomized ID code.

\(^1\)http://snd.gu.se/en/support-researchers/data-management

\(^2\)http://www.datainspektionen.se/in-english/legislation/the-personal-data-act/
Chapter 5

Results and Analysis

5.1 Dividing Time into Activities

We computed percentages of the total effort that students spent on Editing, Active Use, Passive Use, and time spent Out of Tool on each of the 2,643 programming sessions with recorded events from both 2014 and 2015.

Table 5.1: Estimations of time spent on different activities based on 2,643 observations from 2014 and 2015 on a wide range of programming assignments. The Out of Tool time consists of breaks up to fifteen minutes.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Min</th>
<th>Average</th>
<th>Median</th>
<th>Max</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editing (E)</td>
<td>0.2%</td>
<td>14.8%</td>
<td>12.7%</td>
<td>74.6%</td>
<td>9.9%</td>
</tr>
<tr>
<td>Active Use (AU)</td>
<td>1.4%</td>
<td>39.5%</td>
<td>38.5%</td>
<td>96.8%</td>
<td>16.6%</td>
</tr>
<tr>
<td>Passive Use (PU)</td>
<td>0.0%</td>
<td>27.2%</td>
<td>25.6%</td>
<td>75.0%</td>
<td>12.5%</td>
</tr>
<tr>
<td>Out of Tool (OT)</td>
<td>0.0%</td>
<td>18.6%</td>
<td>11.6%</td>
<td>93.5%</td>
<td>20.3%</td>
</tr>
</tbody>
</table>

The amounts of effort spent on different activities are presented in Table 5.1. The values are percentages of assignment solving Total Time, including the time spent Out of Tool on breaks up to fifteen minutes. We find that the time for Editing is very similar between the years and is 14.8% in 2014 and 14.7% in 2015 even if the number of assignments increased, and some assignments changed. The top left histogram in Figure 5.1 show that editing time is positively skewed. The Active Use is the time that students are actively using the mouse, navigating between files and are doing text selections. The distribution for Active Use looks normally distributed around the average but has a right tail. We find that students spend 37.9% of their effort on Active Use in 2014. In 2015 Active Use is a little larger, 40.4%. The overall average for both years is 39.5% of the programming session time. This is the amount of programming

---

1This data and part of the results will be published in the International Journal of Engineering Education [TOEW p] and has been previously published in the IFIP TC3 Conference in Vilnius 2015 [TOEW15]
Figure 5.1: Histograms on the assignment solving activities based on 2,643 observations from 2014 and 2015.

...effort that tools on lower granularity miss and that can be further investigated by CSQUIZ replay functionality.

Passive Use is time when we only know that the tool is open and that text has not yet faded. During this time there are no events collected from the students. The amount of effort for Passive Use is 29% in 2014 and 25.9% in 2015. We do not know what the students are doing, but when watching replays we see that the mouse suddenly stops for a while or moves outside of the application window.

The average time for Passive Use decreased with 4% in 2015 in comparison to 2014 but the amount of short Out of Tool breaks did increase. These breaks can be up to fifteen minutes long since we used a fifteen minute threshold to calculate Total Time. The time spent Out of Tool with either the tool closed or with text faded was 18.2% in 2014 and increased to 19% in 2015, on average it ends up to 18.6%. Figure 5.2 plots Time in Tool versus time Out of Tool and shows that in longer sessions students spend more time Out of Tool. The amount of time spent Out of Tool is strongly correlated ($R = 0.76, p < 2.2e-16$) to the amount of Time in Tool.

In Figure 5.3 on page 54 we show how the percentages of activities vary on different assignments in 2015. We can see that the percentage of time spent on reading time (Active Use + Passive Use) remains a large part of the programming effort independent of assignment. The Editing time varies but
Figure 5.2: Students spend more time Out of Tool on long assignments. The Out of Tool consists of breaks up to 15 minutes long. Linear regression line.

remains a smaller part of the entire effort.

5.2 Comparing Granularities

Table 5.2: Different combinations of sampling granularity and thresholds are compared the Time in Tool. Each value is a percentage of the Time in Tool model. Based on data from 2,643 programming session recordings. $G4(T:3s)$ is the baseline that the other models compares with.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>3s</th>
<th>10s</th>
<th>1m</th>
<th>5m</th>
<th>15m</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1. Compilations</td>
<td>1.3%</td>
<td>2.7%</td>
<td>24.8%</td>
<td>76.4%</td>
<td>109.2%</td>
</tr>
<tr>
<td>G2. Key Stroke</td>
<td>17.6%</td>
<td>27.9%</td>
<td>60.1%</td>
<td>99.5%</td>
<td>117.1%</td>
</tr>
<tr>
<td>G3. Interactions</td>
<td>65.1%</td>
<td>80.9%</td>
<td>101.7%</td>
<td>118.8%</td>
<td>133.5%</td>
</tr>
<tr>
<td>G4. Visibility</td>
<td>100.0%</td>
<td>100.9%</td>
<td>107.0%</td>
<td>121.4%</td>
<td>138.1%</td>
</tr>
</tbody>
</table>

20 time-aggregation models based on four different granularities and five thresholds have been used to compute effort estimations for 2,643 programming sessions. Each effort estimation has been divided by Time in Tool to produce the fraction of effort the model computes. Table 5.2 presents the values as percentages of Time in Tool. From the table we can read that when we sample using Key Stroke granularity and use a three-second threshold only 17.6% of
the Time in Tool is captured. If the threshold is increased to five minutes on average 99.5% of the time in tool is captured. Coarser sampling models results in either an underestimation (values under 100%) or an over-estimation (values over 100%). By increasing the threshold we can reach an acceptable (close to 100%) time estimation on average. However the error in individual measurements does increase when the thresholds gets longer since we start to capture time that was spent outside of the tool. Having a too large threshold results in overestimation.

Figure 5.4 on the facing page shows a box-plot for each model. The average time for Compilation granularity close in on an acceptable time estimation (on average 109.2%) with a threshold of fifteen minutes. The individual errors on these estimations are severe. For one student we still capture only 0.08% of the effort, for another student we overestimate the effort by almost seven times (683.1%). The standard deviation for computing effort on Compilation granularity with fifteen minute threshold is 41.3%. On Key Stroke granularity we get an average of 99.5% with a five minute threshold and the standard deviation is down to 24.2%. Increasing the granularity level further to Interactions lets us lower the threshold further down to one minute. This also decreases the standard deviation to 6.8%. The last boxplot in Figure 5.4 shows what happens when we overestimate time on Visibility granularity and start capturing time Out of Tool.

When we exclude the application start-up event from the Compilation granularity (not shown) the Compilation granularity still underestimates the average time spent even on large thresholds (we tested up to three hours).
Figure 5.4: Y-axis is the fraction of effort Time in Tool ($G_4(T : 3s)$) when comparing twenty aggregation models. The X-axis on the five box-plots represents different thresholds (3s, 10s, 1m, 5m, and 15m). Values over 1 (100%) are over-estimations and values below are underestimations.
5.3 Experiments

Here are the results from the three experiments examining redundant comments, useful comments, and a change scenario.

5.3.1 Redundant Comments

Table 5.3: Time in seconds from the comment/no comment experiment with redundant comments. Version A has no comments while B has redundant comments. Time is measured on $G4(T : 3s)$ and time for viewing instructions and breaks is not included.

<table>
<thead>
<tr>
<th></th>
<th>2014</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Average</td>
<td>410.4</td>
<td>775.1</td>
</tr>
<tr>
<td>Median</td>
<td>309.0</td>
<td>472.0</td>
</tr>
<tr>
<td>Min</td>
<td>42.0</td>
<td>100.0</td>
</tr>
<tr>
<td>Max</td>
<td>1576.0</td>
<td>4351.0</td>
</tr>
<tr>
<td>SD</td>
<td>369.1</td>
<td>1039.4</td>
</tr>
<tr>
<td>N</td>
<td>21.0</td>
<td>21.0</td>
</tr>
</tbody>
</table>

The first experiment with redundant comments and type-hinted arguments has been conducted twice, first in 2014 and then in 2015. Looking at Table 5.3 we see that in 2014 group B with comments took on average almost 6.0 minutes (365s) longer than group A without comments, to complete the assignment. However the data from 2014 is positively skewed as can be seen in the QQ-plots in Figure 5.5 on the facing page. A few large observations drag up the mean value, making the median a better candidate for comparison between the groups. A Students T-test assumes normally distributed data, compares averages, and is unreliable for this data. For that reason we use a Mann-Whitney U-Test that has the advantage of not requiring normally distributed input [WRH+00]. A Mann-Whitney U-Test for independent samples on the data from 2014 indicates that the time to complete version B (with comments) ($Median = 472$) was NOT significantly longer than for version A ($Median = 309.0$), $p = 0.22$, and $U = 171$. A NOT significant difference means that the +163s difference we see in medians may just be due to the variation of values. A high p-value like 0.22 means that in 22% of a large set of tests we would still see a difference of this magnitude when the null hypothesis is true.

In 2015 we repeated the experiment and Table 5.3 shows that the average and median effect differs in the other direction. Another Mann-Whitney U-Test test for the 2015 data fails to indicate a significant difference between A ($Median = 431.0$) and B ($Median = 386.5$), $p = 0.8$ and $U = 512$.

We note that for both years the standard deviation is large in comparison to the differences in median indicating large individual differences within the groups. We have tried different ways of using other tasks as a pre-test, but even if we use a pre-test, we still cannot see a difference between the groups.
Figure 5.5: QQ plots show that the data is not normalized but positively skewed. Normally distributed data would follow the theoretical line. A non-parametric test such as the Mann-Whitney U-Test is better suited to compare samples from this kind of distributions.
We thus FAIL to reject the null hypothesis. If there really is an effect it might be much smaller than we can detect and is hidden behind confounding factors like student skill.

5.3.2 Use a Class With Comments

Table 5.4: Time in seconds for reading time in the file HTMLPageView.php, both groups have the same code but group B also got helpful comments with usage examples.

<table>
<thead>
<tr>
<th></th>
<th>Read A</th>
<th>Read B</th>
<th>Read A</th>
<th>Read B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2014</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>100.6</td>
<td>106.4</td>
<td>59.8</td>
<td>95.1</td>
</tr>
<tr>
<td>Median</td>
<td>51.0</td>
<td>80.0</td>
<td>33.5</td>
<td>87.0</td>
</tr>
<tr>
<td>Min</td>
<td>6.0</td>
<td>19.0</td>
<td>0.0</td>
<td>8.0</td>
</tr>
<tr>
<td>Max</td>
<td>678.0</td>
<td>415.0</td>
<td>333.0</td>
<td>291.0</td>
</tr>
<tr>
<td>SD</td>
<td>152.4</td>
<td>93.6</td>
<td>77.2</td>
<td>70.0</td>
</tr>
<tr>
<td>N</td>
<td>23.0</td>
<td>19.0</td>
<td>23.0</td>
<td>39.0</td>
</tr>
</tbody>
</table>

In our second experiment we investigated if helpful comments in a class increase the reading time and if it also decreases the programming effort of using that class.

The experiment was conducted in 2014 with 42 students. The division of students into the experimental groups was done by CSQUIZ. Before starting the experiments the students had practiced to use CSQUIZ in nine tasks. The training exercises included how to instantiate objects, how to require files, how to call methods, and how to work with namespaces. To start the experimental task students must first finish all nine previous tasks. The experiment was repeated in 2015 with 62 students. Table 5.4 show that CSQUIZ did assign 23 students to group A and 39 students to group B. The division was surprisingly uneven. CSQUIZ uses the PHP rand-method and we suspect this was just a random fluke. The students in 2015 were prepared for the experiment in the same way as the students from 2014. Except in 2015 the students did two more tasks to prepare for the experiment, including the divided task of Experiment 3 and an extra task on handling errors.

We measure read time in the HTMLPageView.php file that students should extract information from and that was provided in two versions, with (B) and without comments (A). We investigate the distributions for read and write in density-diagrams in Figure 5.6 on the facing page. It shows that the distributions for the observations are positively skewed like in experiment one. For that reason we use one-tailed Mann-Whitney U-Tests for independent samples.

A Mann-Whitney U-Test for the read class times in 2014 indicates that the time to read class version B (Median = 80) was longer than for version A (Median = 51), \( p = 0.059 \) and \( U = 281 \). The p-value \( p > 0.05 \) indicates this might not be significant, and on its own we would reject it. However, the
Figure 5.6: Density graphs show read and write effort in code without comments and with comments (dotted line). Read time for code with comments is significantly longer in 2015. All distributions show positive skew.
result from the second Mann-Whitney U-Test on the observations from 2015 confirms that read time is longer when comments are introduced. Read time in 2015 for version B (Median = 87) was significantly longer than for version A (Median = 33.5), \( p = 0.002 \) and \( U = 668 \). We therefore reject the hypothesis \( \text{ReadH}_0 \), and the alternative hypothesis \( \text{ReadH}_1 \) that the effort to extract information from a class is significantly longer in the presence of comments is strengthened. The effect size median difference is 29 seconds in 2014 and 53 seconds in 2015.

Table 5.5: Time in seconds for effort spent in the file \textit{index.php}. This time includes reading, writing, and debugging.

<table>
<thead>
<tr>
<th></th>
<th>2014</th>
<th></th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write A</td>
<td>Write B</td>
<td>Write A</td>
</tr>
<tr>
<td>Average</td>
<td>270.7</td>
<td>279.7</td>
<td>199.5</td>
</tr>
<tr>
<td>Median</td>
<td>236.0</td>
<td>200.0</td>
<td>141.0</td>
</tr>
<tr>
<td>Min</td>
<td>63.0</td>
<td>80.0</td>
<td>54.0</td>
</tr>
<tr>
<td>Max</td>
<td>921.0</td>
<td>757.0</td>
<td>486.0</td>
</tr>
<tr>
<td>SD</td>
<td>208.4</td>
<td>220.4</td>
<td>122.3</td>
</tr>
<tr>
<td>N</td>
<td>23.0</td>
<td>19.0</td>
<td>23.0</td>
</tr>
</tbody>
</table>

We also test if effort spent in \textit{index.php} that students write their own code in was affected by the presence of helpful comments in the class file. We measure the effort spent in \textit{index.php} and present in Table 5.5. We can see that the median time spent in \textit{index.php} is 36 seconds shorter in 2014 for the group with comments. In 2015 the median time is 94 seconds longer for the group with comments. The write times are not normally distributed, which can be seen in the density diagrams in Figure 5.6 on the previous page.

To test the write hypotheses we use one-tailed Mann-Whitney U-Tests for independent samples and testing if the writing time for the group with comments (B) is less than without comments (A). The test indicate no difference between the writing time in 2014, \( p = 0.93, U = 214.5 \). For the 2015 data a Mann-Whitney U-Tests indicates that that the group A is significantly faster (Median = 141) than the group B (Median = 235) when writing code, \( p = 0.03, U = 621 \). The p-value is relative high which indicate that we might have a false positive. Since we are conducting many significance tests, we use Bonferroni correction [GS14] to adjust the significance levels to compensate. Testing four hypotheses with alpha level of 0.05 results in an adjusted significance level of 0.0125. Since the p-value is much higher and we get a non-significant result in 2014, we FAIL to reject the null hypothesis and conclude that the time for writing code in \textit{index.php} was NOT affected by the presence of comments in \textit{HTMLPageView.php}. The helpful example in the comments was not as useful as we thought after all.
5.3.3 Informed Change

An assignment in 2014 was divided into two smaller assignments in 2015. We compare the total effort of the two assignments to the effort in 2014.

**Table 5.6**: Total effort in seconds to complete 14 assignments that did not change significantly between the years of 2014 and 2015. The difference in effort is small.

<table>
<thead>
<tr>
<th></th>
<th>2014</th>
<th>2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>15453.7</td>
<td>15458.3</td>
</tr>
<tr>
<td>Median</td>
<td>13270.0</td>
<td>13634.0</td>
</tr>
<tr>
<td>Min</td>
<td>2458.0</td>
<td>2328.0</td>
</tr>
<tr>
<td>Max</td>
<td>78808.0</td>
<td>63920.0</td>
</tr>
<tr>
<td>SD</td>
<td>13297.6</td>
<td>11199.8</td>
</tr>
<tr>
<td>N</td>
<td>39.0</td>
<td>59.0</td>
</tr>
</tbody>
</table>

In this experiment we are comparing student groups that were NOT randomly assigned to different versions of the code. Instead the first group contains students from 2014 and the second group took the course in 2015. To compare them we need to find out if they can be considered to have the same skill. Therefore we calculate the total effort that students spend on fourteen assignments that was NOT changed and compare the two classes on them. We included assignments that changed name or was updated with smaller corrections like spelling mistakes. We did not include assignments that changed the amount of work, or was completely replaced. We exclude six students that did not complete all fourteen tasks and calculate the total effort to complete all fourteen assignments for both years. The values are presented in Table 5.6. We see that the effort to solve the fourteen assignments in 2015 was just six minutes longer than in 2014. A box-plot in Figure 5.7 on the next page show that the distribution of effort in the fourteen assignments is (again) positively skewed and that the student groups are very similar.

To test if the same skill is used in the fourteen assignments as in the changed assignment, we analyzed the relationship between the effort to solve the fourteen assignments and the single assignment in 2014, and find a strong correlation, \( R = 0.83, p < 0.0001 \). In 2015 the correlation is moderate between the tasks and the total time of the two tasks, \( R = 0.65, p < 0.0001 \). Figure 5.8 on page 63 shows linear regression lines between the changed assignment and the total effort to complete the fourteen assignments. We can see that students who solved the fourteen assignments fast also solved the changed assignment fast. The residuals increase with time. This indicates that the prediction becomes more unreliable when the time to solve assignments increase. Even so, we consider the student groups comparable since the groups performed similarly on other assignments that did not change.

We measure the time to complete the changed assignment and present it in Table 5.7. The median time to solve the assignment was 2,683 seconds in T0 and the median times to solve the two smaller assignments was 1,458 seconds.
Figure 5.7: Boxplot comparing the total amount effort to complete 14 tasks that did not change between the years. The plots indicate that students had similar skill in both years.
Figure 5.8: Linear regression lines show that the fourteen tasks have a positive relationship to the foreach task. Students who are fast in these tasks tend also to be fast in the foreach task(s). Note the different scales.
Table 5.7: Effort measured as time in seconds in the one task in 2014 that was divided into two tasks 2015. The second column is the sum on individual student level and not a sum of column three and four.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>3224.0</td>
<td>3452.0</td>
<td>1786.0</td>
</tr>
<tr>
<td>Median</td>
<td>2683.0</td>
<td>2846.0</td>
<td>1458.0</td>
</tr>
<tr>
<td>Min</td>
<td>536.9</td>
<td>524.0</td>
<td>145.0</td>
</tr>
<tr>
<td>Max</td>
<td>11360.0</td>
<td>17880.0</td>
<td>13930.0</td>
</tr>
<tr>
<td>SD</td>
<td>2530.8</td>
<td>3135.0</td>
<td>1934.2</td>
</tr>
<tr>
<td>N</td>
<td>43.0</td>
<td>63.0</td>
<td>63.0</td>
</tr>
</tbody>
</table>

$T_1$ and 1,219 seconds for $T_2$. The median time for their combination is based on the individual student times and is 2,846 seconds.

To test if the total time is significantly longer for the two assignments in 2015 compared to the one assignment in 2014 we use two-tailed Mann-Whitney U-Tests for independent samples. The test indicate no significant difference ($p = 0.67, U = 1288$). We are therefore unable to reject the null hypothesis $TimeH0$. The division of the assignment into two did not make it harder or easier, solving the same requirements requires the same amount of effort.

Table 5.8: Number of breaks longer than fifteen minutes in 2014 and in the two tasks $B_1$ and $B_2$ in 2015 as well as the total number of breaks in the fourteen tasks that was not changed.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.72</td>
<td>1.20</td>
<td>5.60</td>
</tr>
<tr>
<td>Median</td>
<td>0.00</td>
<td>0.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Min</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Max</td>
<td>10.00</td>
<td>8.00</td>
<td>27.00</td>
</tr>
<tr>
<td>SD</td>
<td>1.86</td>
<td>1.90</td>
<td>7.40</td>
</tr>
<tr>
<td>N</td>
<td>43.0</td>
<td>63.0</td>
<td>39.00</td>
</tr>
</tbody>
</table>

We measure the number of breaks longer than fifteen minutes the students took in 2014 and 2015. These values are presented in Table 5.8. We did not include breaks between the assignments.

We see that the average number of breaks increased from an average 0.72 breaks per student to 1.2 breaks per student in 2015. Most students does not take a break for this assignment but a Mann-Whitney U-Test for independent samples indicates that the number of breaks longer than fifteen minutes for the two assignments in 2015 was higher than for the one assignment in 2014 ($p = 0.02, U = 1086.5$). The number of breaks has increased overall in 2015. We counted the number of breaks taken on the fourteen assignments and students took on average 5.6 breaks in 2014 and 6.4 in 2015. Since students in 2015 do take more breaks on average, we are unable to attribute the increase in breaks
to the change of the assignment.

The number of forum questions asked on the assignment dropped from four questions in 2014 to zero questions on the two assignments in 2015.
Chapter 6

Discussion

6.1 Dividing Time into Activities

We used CSQUIZ to compute average fractions of programming solving effort spent on different activities. We found that on average only 15% of the programming effort is spent editing the text. Tools that capture events on key stroke granularity thus miss 85% of the total effort on average. By capturing mouse and text caret events we found that on average another 40% of the effort is spent actively using the editor but not editing the text. In an editor that does not fade the input area we would expect the time for Active Use to be lower than our measurements, since students move the mouse to keep the area from fading. Our students spent on average 27% of their time with the editor open but without interacting with it. This Passive Use time is an aggregation of many small micro-breaks that are longer than three seconds from the last interaction to a new interaction, or until the text fades out. The maximum time for reading can thus be up to the total of Active and Passive Use and is on average 66.7% (not including editing). These average fractions are very similar between 2014 and 2015 even if some of the assignments differed. The time spent Out of Tool was on average 19% but the median fraction was only 11.6%. This is explained by that in a quarter of all programming sessions, the students solved the entire assignment without taking any breaks. In many of the assignments the students spent very small amount of time outside of the tool. Students may prefer to take breaks between assignments instead of within to keep their focus. However the time you can spend concentrated on a task is not infinite. We did find a strong correlation between Time in Tool, and the time Out of Tool. This means that students take more breaks on longer assignments. We are a bit concerned that students take more breaks in 2015 than in 2014. In 2013 many of our students reported to be frequently interrupted by both internal and external factors during their programming, the main reason was that they used social media or got interrupted by a friend during programming. The reason for our concern is that we increased the theory in the assignment instructions in 2015 to keep the students working in the tool instead of searching for information online.
We have seen that the fractions of time spent on different activities vary between assignments. The first three assignments are very short and require very small text changes. Students also spend short fractions (6–8%) of their time editing those. Another example is the first experiment. In that task, only two characters needs to be changed to solve it. The average student used on average 10% of their time editing the code. This can be seen as column eight and nine in Table 5.3 on page 54. However, some students tried many different changes before reaching the solution, and others just mindlessly edited their way to a solution. For example, one student used 28% of the total effort writing code in the first experiment. Soloway and Spohrer observed similar behavior in novice programmers, and called those the Extreme Movers [SS88]. Maybe mismatches between expected editing effort and actual effort can be used detect such behavior?

Ko et al. [KMCA06] studied full screen recordings of ten experienced programmers working on five different maintenance tasks. The observed programmers spent 20% of their time editing the code and 45% of their time reading and navigating. These values are a bit higher compared to our averages. It seems reasonable that the more experienced programmers in the study were also more focused and effective in their work than our students. We find it interesting that even if the tasks, tools, programming language, and programmers differed we got quite similar fractions of time as Ko et al. A main difference in method is that we automated the recordings and calculation of effort while Ko et al. computed their values by hand.

There are a number of sources of uncertainty in our measurements. First, we have no control over what the students are doing during their time Out of Tool. We selected fifteen minutes as the longest break a student would take and be working on the task. It is possible that students spend more time on the task outside of the tool, e.g., read documentation. Second, during the Passive Use time we do not get any interactions, so the students may not be working. However shorter periods of Passive Use that does not lead into a break is probable reading time.

Even the time for Active Use may be used for other activities. For example, the student could be moving the mouse over the CSQUIZ editor without reading or working. Even editing of the text can happen by accident but we consider that to be quite rare. Since CSQUIZ can replay the events, we can at least observe such events, and try to determine if the events make sense or not. For example: When watching a recording we sometimes see a passive period followed by a quick jerking mouse movement. We interpret that the student wants to keep the text from fading and may see that as a stronger indication that the Passive Use time before the mouse movement is used for reading. Such heuristics could be used to further divide the Passive Use time.

We could also improve the accuracy of our estimations by adding even more control over the measurement situation. For example, if we captured full screen recordings of the student’s computer, we could differentiate a student’s social media use from reading API-documentation. Eye tracking equipment could be used to accurately determine the reading time, and also what the students read. Observations through camera or in person could tell us more of what
The students are up to when they do not interact with the computer. However, each increment in instrumentation requires more effort, more infrastructures, and more equipment. Increased instrumentation also calls for more severe privacy concerns. There are both technical and ethical obstacles that may not be worth the climb.

Even on code submissions granularity there are privacy issues, students may by accident submit code that they do not want us to see. We may capture edits that were unintentional, or that may contain private information. To mitigate that risk, we have been very open with what events we collect, and we have not seen any private information in the recordings. The students did not mind having their mouse recorded when they understood it was just in the application and not full screen. In 2015 the students told us in the course evaluation that they were happy about using CSQUIZ, but did not like fading the text on inactivity. Fading text is thus a bit too intrusive to use so extensively, and perhaps it should only be used in experiments.

### 6.2 Comparing Granularities

To test the accuracy of the increased granularity we created 20 effort estimation models using simulated lower granularities and varied the thresholds. We found that when the threshold is increased, we are able to produce quite accurate results on average. The trade-off is that the error increases with larger thresholds. Having a too low granularity means that we risk severe overestimation or underestimation in individual cases. With increased granularity comes the ability to make better effort estimation models.

#### Figure 6.1: CSQUIZ timeline showing events in different files, the yellow editing events in *index.php* are followed by periods of green mouse movement events and blue text selections, indicating collection of information in other files.

The Interactions granularity produces very accurate results with only one minute threshold. The reason for this is that the mouse movement events cover the gaps of editing events in the timeline. This can be seen in Figure 6.1 where the green mouse movement events almost always are disjoint from the yellow
key-stroke events except when the student selects text (blue) and paste text
(orANGE). The error in the Interactions granularity comes from being unable
to separate Passive Use from time spent Out of Tool. To do that we need to
fade the text or even more intrusive instrumentation like eye-tracking. The
Compilation granularity is unfairly represented in our models, since its events
are sometimes very spread out and do not connect to others on small thresholds.
We counted single events only as a half a second of time. If we increase the time
for single events we could create a better estimation, but that would probably
also increase the error.

To accurately capture the students’ effort, at least Interaction granularity
should be used with one minute threshold. If many programming sessions are
averaged, and we are not looking at individual efforts, the Key Stroke granu-
larity with five minute threshold makes a good estimation model. The main
reason for errors is that students are interrupted. Fading of text was effective
to differentiate between passive reading and time spent on other activities but
it was too intrusive and our students did not like it. A shorter time for fad-
ing or more intrusive instrumentation may be used in experiments to produce
more accurate reading times. We could also ask the students to give us more
information, for example if they have been inactive we could ask them what
they did when they were away.

6.3 Experiments

6.3.1 Redundant Comments

We searched for an effect of introducing redundant comments into an assign-
ment. In 2014 the effort was higher for the group with comments, and in 2015
the effort was higher for the group without comments. It is important to re-
member that we do not know if there is an effect or not. If there is an effect
it is well hidden in the individual differences between the programmers. The
average time to solve the assignment for all groups was little longer than nine
minutes. With only twelve lines of redundant comments, we estimate the effort
to read them to be small. Some students may ignore the comments altogether
and just read the code, in those cases the comments should have minimal ef-
fect. The median time to solve the assignment was around seven minutes and
we got huge standard deviations. The difference between the shortest time to
solve the assignment (42 seconds) and the longest (72 minutes) is two orders
of magnitude. With such high variation we are unable to detect a difference in
programming effort.

We conclude that for redundant comments in a debug scenario the effect is
small. This indicates that redundant comments do not help, but neither does
any harm. Thus when creating assignments for students we might want to write
redundant comments just to show how to write comments, but not to expect
them to help the students unless we add information that makes them worth
reading. The students participating in the experiment had been programming
for one or more years. The effect on less experienced programmers might be
another; they might prefer English prosaic comments to source code.
6.3.2 Use a Class With Comments

The second experiment allowed us to make more accurately measurements of reading time. We measured the time spent in separate file that students did not change. In 2014, we did find a increase in reading time that was confirmed and significant in 2015. In 2014 the difference was half a minute, and in 2015 it was almost a minute. The standard deviation in read time was much smaller than in the last experiment, which allowed us to find such a small difference. The number of lines with comments was 22, so the increased reading time of having such comments would be 1.36 seconds per line \((30s/22)\). Extrapolating from that, the time for reading twelve lines of comments in the first experiment would be only sixteen seconds. Such small effect makes little difference when programming sessions are long. Tenny [Ten88] found that his students benefited from reading comments in his experiment. He hypothesizes that the comments in his experiments may have saved code that was hard to read. Buse and Weimer [BW10] investigated readability of comments and found they had little impact on readability. Buse and Weimer [BW10] explains this by stating that comments are more common in unreadable code. We never aimed to make the code part less valuable or unreadable which may explain why we did not see a decrease in reading time. In this experiment it might be just a difference in size of files that makes it easier or harder to extract information. If the information is harder to extract from the code like in more complex scenarios then information stored in comments might prove more valuable.

![Figure 6.2: Timeline for a student who spent no time in the HTML-PageView.php class file and guessed the order of arguments and namespace names.](image)

Before conducting the statistical analysis and tests we examined the data manually by playing back recordings of students that took very little time. A student in 2015 completed the assignment without even opening the HTML-PageView.php file. We inspected the recording (See Figure 6.2) and found that the student used the instructions to find out the name of class and methods, and guessed the order of parameters. The theory presented together with the instructions showed how another class was created and that class was in the same namespace as HTMLPageView. This shows that having or not having comments may not matter if they are not read. It also shows that we had more redundancy in the second experiment than we thought we did.

Another student in 2014 spent only 8 seconds in index.php and no time in HTMLPageView.php. That student did copy-paste a solution from outside of
CSQUIZ and was removed from the data set. This shows that in a data set collected in the field and not in a laboratory environment we do get students who cheat, talk to each-other, share solutions, and get help from teaching assistants. Significance tests such as a Mann-Whitney U-Test assume independent samples which we do not have. Increased control over the measurement situation would be preferable to make sure students do not interact with each other. However, since this assignment was short and uncomplicated, we are convinced that most of the measurements are independent.

We have investigated two different scenarios and found that redundant comments had no effect. When we isolate the reading time we see that reading comments takes time.

### 6.3.3 Informed Change

We divided a problematic assignment into two separate assignments and compared the effort of completing the two assignments with the original. We compared the classes of 2014 and 2015 on fourteen tasks that were not changed, and found remarkable little differences. The maximum time and standard deviation was higher in 2014 and in 2015 the fastest student was a bit faster. But the median and average times to complete the assignments were similar. We noted that the distributions were positively skewed with a handful of students that did take extremely much more time than others. In both years, the time to complete the fourteen tasks did predict how much time students would spend on the changed task. We therefore argue that the same skill was used in the fourteen tasks as in the one we changed. From the similarities we argue that the groups are comparable.

When we compare the changed task we find that on average the total effort of completing the two tasks in 2015 is very close to completing a single task in 2014. Since we did not remove any requirement there was the same amount of work to be done. An equal amount of work apparently takes the almost the same amount of time even if we divide it into two different parts. At least the lack of forum posts and questions on the assignment in 2015 do indicate the students needed less support to solve the assignment. We may have hoped to reduce the total effort, by giving the students a stepping stone, but did not. At least we know that we did not significantly increase their effort.

However we found that the number of breaks was significantly higher for the group who got the divided assignment. We are quite surprised to see that since the students got a natural break in between the two shorter assignments in 2015. A possible explanation would be that the group of 2015 has a different break habit. By comparing the number of breaks in the fourteen tasks we found that students in 2015 did take more breaks on average when comparing the fourteen tasks. We are therefore unable to attribute the increased number of breaks as related to the change of task.
6.4 Using CSQUIZ

We started out with a feeling of not understanding how much students worked, a feeling also expressed by other researchers in the field [FNB+09]. We designed and used CSQUIZ to estimate the student effort. Do we understand our students better now? We think that the measurements of student effort, and the ability to observe their programming process, have increased our understanding. First, we now know that some students work for long hours. We can observe their struggles, and when they make slow progress. This has increased our respect for their work, and this makes us want to improve our assignments to better suit their learning need. Other students amaze us with their programming skill and speed. We think that we have failed to meet their needs. They would need more challenging assignments to further develop their skill! While the diversity of skill displayed by the students made experiments more difficult, we were able to measure small effects on reading time in the second experiment. The effect of adding comments was not large but we could measure it; compared to not knowing, that is an improvement.

6.5 Examining Skills

Knowing that a part of the class struggles and spends unreasonable amounts of time is humbling. It makes us interested in why some of the students complete these assignments so much faster than others? In most of the assignments there are outliers taking much more time to complete the assignment than others. What is it that makes an assignment so hard for some of the students?

We have seen in the three experiments that student effort varies a lot. In Figure 6.3 we can compare the effort required to solve all assignments conducted in CSQUIZ in 2014. Each vertical line is an assignment. Each horizontal line is an individual student and the height at which it crosses the vertical line is that students effort in that assignment. In the problematic assignments (the hills) we can see that all assignments have students that spend much more effort
than the main group.

Figure 6.4: An assignment performed before Experiment 1 was subtracted from the effort of the experiment to produce gain values. The gain values had higher variation since it was not the same students that performed well in both assignments.

Figure 6.5: Showing student ranks from the 2014 dataset. Each horizontal line is a student. There are a lot of students who switch ranks over the assignments.

We have tried to remove the variation by using other assignments the students had solved before the experiment. For example we used the effort of another assignment as a pretest and calculated a gain value \( \text{gain} = \text{effort(experiment)} - \text{effort(pretest)} \). A pretest that removes variation should lower the standard deviation. We actually got higher deviations in some cases. In Figure 6.4 we can see that the gain distributions get tails on both ends instead of a positive tail. This means that students did not rank the same in both assignments. Maybe the pretest exercised other skills than was used in the experiment? The use of pretest has been questioned by others. Kleinschmager and Hanenberg [KH11] compared three different ways of rating programming skills: University marks, pretests and self-estimations and found that neither are good predictors of programming effort.

Even so the strong correlation shown as a linear regression line in Figure 5.8 on page 63 seems to indicate that on average one student can be ranked as faster or slower compared to other students, and that we can use this knowledge to predict the student’s future effort. It seems reasonable that lower effort
would indicate a higher skill, a student with more experience should solve an assignment faster than an unskilled student since a student that lacks a skill must learn to be able to solve the assignment. But if the students’ skill decides their effort then students should have similar rankings over many assignments. A fast student in one assignment should also be fast on the next, and so on. While we find that to be true on average, we observe something different when we study students’ rankings (c.f., Figure 6.5). Instead of time we show ranking position with fastest students at the bottom. We find that students swap ranks a lot, illustrated by lines crossing each other. A student that was among the fastest in one assignment is among the slowest in another.

![Figure 6.6](image.png)

**Figure 6.6:** The Views assignment was conducted twice with the same students. Students who took the longest to solve the first time are not necessarily among the slowest the second time. To understand what happened we must look at the programming process.

We have tested to let the student solve the same assignment again. The median effort for students solving the “views” assignment the first time was 67 minutes. When the students did the same (same code, other data) assignment a second time the median time was 23 minutes. We argue that the same skills are required to solve two assignments, so students should rank similarly in both. In Figure 6.6 we see that the slowest students from the first attempt are suddenly in the middle of the group in the second attempt and some of the students that ranked in the middle of the group in the first attempt are suddenly among the slow students. We do not know why this is, but it is clear that we are unable to explain this using only high level statistics.

We pose three hypotheses that may explain the differences. Our first hypothesis is that different students are not ending up in the same solution state. A student that aimed for higher quality might be spending more time on the assignment. Bergersen et al. [BHS+11] argue that the effort must be combined with quality to make sense. For example, a low effort in combination with low quality is not easily compared to low effort with high quality, since
the increased quality may require more effort. Bergersen et al. [BHS+11] find empirical support to use a two dimensional Guttman-structured scoring that combines quality and time into a single score. While we agree this makes sense, we argue that a good measure of quality must be used. Bergersen et al. [BHS+11] use completeness of requirements. In CSQUIZ the effort is measured until all tests pass, so the end state is always 100% complete and fulfills all requirements. We cannot separate students’ solutions using completeness. We consider creating better test suites using static code analysis on architecture level or code metrics to better measure quality.

Our second hypothesis is that different skills are used in different assignments. Students may be good at some skills and need to learn or struggle with others. Learning takes time. CSQUIZ detailed recordings could be used to inspect patterns in students information collection, or when students are unable to make progress. This could be an indicator of learning.

Our third hypothesis is that the students that are among the slowest encountered some unknown problem. For example, they may have introduced a bug and spent a lot of their time debugging or they did not read the instructions and did the something wrong. To explain more of what happened and why it sometimes takes more time, we need to look at the programming process and try to identify activities that can be used to explain the effort.
Chapter 7

Conclusions

RQ1. How can student programming assignment solving effort be automatically measured with high accuracy?

We have seen that a large proportion of the programming time is dedicated to writing code, reading, and navigating through files by tracking keys, mouse, and text caret. On average this is 54% of their total assignment solving time. The remaining time we can separate into time that students may be using for reading code or instructions in the tool (27%) and time they spend out of tool (19%). We have seen that mouse, keys, and text caret must be tracked to accurately measure the programming solving effort with little individual error.

If individual errors do not matter much we could produce accurate averages of the time spent in tool using a coarser granularity by increasing the threshold. We present these as recommended thresholds in Table 7.1.

Table 7.1: Recommended thresholds for time aggregation models of lower granularity. The percentage is the Time in Tool we got with our data for that threshold and the standard deviation.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Threshold</th>
<th>Min</th>
<th>Average</th>
<th>Max</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1. Compilation + Start</td>
<td>15 min</td>
<td>0.08%</td>
<td>109%</td>
<td>683%</td>
<td>41%</td>
</tr>
<tr>
<td>G2. Key-Stroke</td>
<td>5 min</td>
<td>4%</td>
<td>100%</td>
<td>412%</td>
<td>24%</td>
</tr>
<tr>
<td>G3. Interactions</td>
<td>1 min</td>
<td>52%</td>
<td>102%</td>
<td>185%</td>
<td>7%</td>
</tr>
</tbody>
</table>

To detect small differences in reading time, tracking text-visibility is a must since students spend a lot of time reading without interacting. Fading out text was however not a popular feature for our students. To get more accurate measurements than ours, we would need even more intrusive instrumentation. Shorter fading time, full screen recordings, eye-tracking, or direct observations would allow us to account for time that was spent outside of the tool, and when students are not actively interacting. We do not consider those possible to use with our students that are working from home. However, to capture small changes in reading time in an experimental context in the laboratory, increased instrumentation is both possible and needed.
RQ2. Do source code comments in a programming assignment significantly decrease the effort to solve that assignment?

We have seen that redundant source code comments did not significantly affect the effort in the first experiment. However there was a huge deviation in results as the fastest student spent less than one percent of the slowest students time to solve the assignment. Small effect sizes cannot be detected with such a high variation. The second experiment had much less variation. In the second experiment we could see an increase in the time spent in the file that had comments. More text do take more time to read. We estimate the increased reading time to 1.36 seconds per line of comment. We learned that despite our effort to make these comments useful the comments were highly redundant. The same information could be found in code and instructions. We could not detect any benefits from having these comments in the file. We therefore conclude that redundant comments have little effect on the programming effort except increasing the reading time.

RQ3. Does dividing a programming assignment into two smaller assignments decrease the total effort of completing the assignment?

We compared the two classes of 2014 and 2015 and found little differences in effort in a wide range of programming assignments. We therefore conclude that the two classes can be compared.

We used the detailed effort measurements in CSQUIZ to do an informed change. First we were able to detect that one assignment was harder than expected. We used recordings in CSQUIZ to understand why it was hard, divided the assignment into two, and added more instructions. We also used CSQUIZ to find out the effect of the change. We found that the change did not decrease the total effort, in fact we got a small increase of the total effort for 2015. The number of forum questions was reduced to zero in 2015 which indicate that the students became more self-sufficient.
Chapter 8

Future work

We are interested in predicting future effort from recorded programming sessions. In the discussion in Chapter 6 we stated three hypotheses that might explain why students shift ranks. To predict future effort we want to identify under which circumstances one assignment can be used to predict the result in another. A tool such as CSQUIZ will be needed to do this and the detailed recordings contain clues to the variations.

We have watched replays of student programming sessions and found patterns that we believe could be automatically detected. For example the debugging in Figure 3.12 on page 27 position seven. Such patterns could possibly be used to explain why students sometimes spend much more effort. The goal would be to be able to automatically classify different activities, and use that classification to explain differences in effort.

We also want to find and isolate skills that predict effort. We think it is probable that different students excel in different areas. If we can detect missing skills we could suggest complementary assignments to students.

A replay contains clues to how the students worked with feedback from tests and compiler. Watching replays also lets us inspect the strategies deployed by the student. One such strategy is to be very reactive to feedback from the test-suite or compiler instead of reading instructions (Second half of Figure 3.12 on page 27). We think it would be interesting to compare different ways of giving students feedback and see how it affects their choice of strategy.

We find the following goals for future research:

- Predict student future effort from existing programming session data.
- Detect presence of programming sub-skills.
- Divide the programming effort into activities or strategies that explain effort.

CSQUIZ will also need to be adapted. In the future we intend to create support for other programming languages. We also want to be able to run static analysis in the test suites, to give students better feedback, and to restrict their solution space. For example, having tests that can assess that the students used a specific architecture or wrote their code with a specific standard.
Bibliography


