Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Investigating the Relationship between Conscientiousness and the Performance in Solving Coding Challenges

Patrick Lux

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	M.Sc Marvin Wyrich
Commenced:	Febuary 3, 2020
Completed:	September 28, 2020

Abstract

A recent study [WGW19] has provided clues that conscientiousness can have a negative effect on the performance in solving coding challenges. Since coding challenges have become a popular tool to assess the problem solving ability and conscientiousness is widely acknowledged to be a positive influence on work performance this has serious implications.
To study this effect and its consequences we conducted an exploratory study to find differences less and more conscientious developers display while solving coding challenges. Further, we analyze the differences found on their impact on the performance in solving coding challenges.
Our findings indicate that software developers of intermediate and high conscientiousness are more likely to create concepts, think in silence for longer periods of time, start implementing later than less conscientious software developers and provide better code quality. Furthermore, we found that software developers use trial and error approaches regardless of their level of conscientiousness.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Motivation

Technical interviews have become a staple of the hiring process within the field of software development. Most successful companies dealing in such endeavors, especially the well-known software giants Amazon, Apple, Facebook, Google, and Microsoft use technical interviews to assess the problem solving ability of their applicants using algorithmic tasks that are to be solved a given time frame. Such tasks are referred to as coding challenges.
Although the exact process of the technical interview itself differs greatly from company to company and sometimes even from interviewer to interviewer, they all have one thing in common: The candidates have to solve or present knowledge about coding challenges in one way or another. More often than not this is done by introducing a problem to the candidate and observing the candidate during his attempt to solve it. In most cases, access to external sources e.g literature and the World Wide Web are cut out of the process and the only help the candidate may rely on is the interviewer himself.
In the past years, there has been a lot of critique against different forms of technical interviews especially but not exclusively in the sense of bias towards specific age groups, technical interviews having a high false-negative rate, excluding candidates based on their personality types as well as testing for abilities not necessarily needed for the job description [BPB19] [BSBP20].
Recent research into the individual characteristics of coding challengers suggests that there are be significant positive correlations between academic success, programming experience, and the coding challenge performance and significant negative correlations between the affective state of sadness and the performance in coding challenges as well as the personality trait conscientiousness and the performance [WGW19]. While academic success and programming experience are, for the most part, factored in as preconditions to get a chance to participate in a technical interview in the first place and the state of sadness is usually temporary, conscientiousness is something different, which, depending on the job description, should be highly sought after. However, according to the study, conscientious developers appear to be at a disadvantage in the current state of technical interviews[WGW19].
Further research into conscientiousness within the field of software development is necessary. We suggest a qualitative exploratory study to investigate the relationship between conscientiousness and the ability to solve coding challenges well and within the given time frame. The findings of this work could help raise the developers' and companies awareness of this issue and may lead to changes in future hiring practices.

1.2 Research Objectives and Contributions

The objective of our research is to explore the correlation between conscientiousness and the performance in coding challenges. As such we identify differences between software developers and analyze which differences can be attributed to conscientiousness. Further, we investigate what impact those differences have on the performance in coding challenges in the context of technical interviews. The contributions of this work are:

1. Less conscientious software developers are less likely to create concepts. Creating concepts within limited time decreases the probability of successfully solving coding challenges. Additionally, participants of technical interviews receive fewer hints during the time they take to create the concept.
2. There is no correlation between conscientiousness and participants using a trial and error approach. Additionally observed developers incorporating trial and error elements into their solution process have a higher probability of solving coding challenges successfully within technical interviews.
3. Less conscientious software developers spend less time thinking in silence. We found thinking in silence has a negative impact on the performance in coding challenges within technical interviews.
4. Less conscientious software developers start implementing earlier. We found less conscientious developers spending less time to conceptualize. Having more time to fix potentially occurring problems increase the probability of solving coding challenges successfully. Additionally, we observed less conscientious software developers receive more hints and guidance from the interviewer since their solving process is more transparent.
5. Software developers of intermediate and high conscientiousness provide better code quality. This can have a positive impact on the impression they make on the interviewer as well as on their scoring.

1.3 Methodological Approach

To investigate the relationship between conscientiousness and the performance in solving coding challenges, we conducted a qualitative exploratory study with the following research questions.

RQ1: What are the differences between more conscientious and less conscientious software developers?

RQ2: How do those differences impact coding challenge performance?

A total of 12 software developers took part in our study, all of which were currently studying computer science, software engineering, or media computer science or had graduated recently. Over the course of our study participants had to solve three coding challenges while being observed using screen-sharing technology. In between those challenges they were personally interviewed, with special regards towards their approach in solving the challenges. Additionally, the participants had to fill out a personal questionnaire, gathering data about their academic performance, experience, and current mood as well as a personality test.

After the study had been conducted the performance score of the participants was calculated based on correctness, time-complexity, and, for one challenge, in elegance. We then proceeded to apply Grounded Theory towards the qualitative data we gathered during the session recordings and interviews to find differences between the approaches of the participants. As next step, we evaluated the personal questionnaire and the personality test with special regards towards their conscientiousness values. Afterwards, we categorized the participants according to their conscientiousness and investigated which differences can be attributed to conscientiousness.

1.4 Structure of the Work

The following Chapter contains the background and related work of this thesis. The current situation in hiring and technical interviews, the coding challenges in technical interviews and programming competitions, behavioral traits, personality including conscientiousness. In Chapter 3 we describe the design of the study, the participants, the technical details, the coding challenges including various solutions as well as all the components of our data collection and data analysis. The differences between software developers and the attributions towards conscientiousness are presented in Chapter 4. In Chapter 5 we discuss the findings and their implications and in Chapter 6 we present our conclusion.

2 Background and Related Work

The related work chapter contains four sections. The first section provides the background of our study. The second section covers general aspects of the current situation and research of technical interviews. The third section provides knowledge about personalities, measurements of personality, and behavioral traits with a focus on the conscientiousness and its role as a performance predictor. The fourth section focuses on coding challenges and their use within educational environments, coding competitions, and technical interviews.

2.1 Background

In a previous study, published under the name of "*A theory on individual characteristics of successful coding challenge solvers*", several potential performances predicting variables, for example, experience, GPA, and personality types, have been analyzed [WGW19]. With the expectation a low score on *extraversion*, one of the five personality dimensions, according to the Big 5 Inventory, might have a negative impact on the performance in coding challenges, they explored the correlations between the personality dimensions and the coding challenge performance. Their findings show no indication that *extraversion*, *agreeableness*, *neuroticism* and *openness* are in correlation with the performance in coding challenges,however, they found a significant moderate negative relationship for conscientious participants meaning individuals who scored high in *conscientiousness* performed worse in the coding challenges compared to those who scored low.

2.2 Hiring and Technical Interviews

Hiring is the process of attracting potential candidates and evaluating them for a given position within a company. Finding the best suitable person for a given job is of utmost importance for any company and can be a tedious time- and resource-consuming task. For this reason, most modern companies have some sort of structured pipeline to order this very valuable process, striving for impartiality and accurate results.

Proposals for standardization of the hiring practices exist but are rarely taken into account. It is up to the companies whether or not to follow any sets of instructional guidance, which tends to lead to companies making and following their own rules. This leads to the fact that hiring processes are performed wildly differently comparing companies, in some cases even departments and sections within the same company. However, in modern software development, this hiring pipeline usually consists of preliminary screenings, behavioral and technical interviews, as well as an offer and negotiation part.

During the preliminary screenings, potential candidates are assessed based on their applications and scores. If these applicants are, on paper, fit the job description, they'll receive an invitation, usually via phone or email. The interviews can generally be separated into two categories, behavioral and technical interviews.

While behavioral interviews focus on the interpersonal skills and the presentation of the candidate, technical interviews are tailored towards evaluating the problem solving or analytical ability of the candidates as well as giving the interviewer an impression of the candidate's cognitive state and his applied thought process.

Since this thesis concentrates on finding out differences in job performance based on the candidates' personality, there is a strong focus on the technical aspect of the hiring process and thus focuses on the technical interview aspect rather than the personal interview.

To test the problem-solving ability, applicants are tasked to solve programming challenges which typically involve writing code or pseudo-code on paper, a whiteboard, or a specific set up computer environment which limits the access of other sources. While technical interviews are frequently conducted on-site, off-site testing of applicants is nothing unheard of, specifically when the given position is for a remote job. Remote technical interviews typically use screen sharing technologies, a coding platform and the interviewer and interviewee remain in a call for the duration of the interview.

Properly carried out, technical interviews are a benefit to the company, the hired applicants, and thanks to constructed feedback, even to candidates who did not receive an offer. Conflictingly recent scientific research suggests the technical interview process, as currently applied by many companies, has a high potential for flaws, which will be discussed in the following analysis of related research [BSBP20][BPB19].

Bad Practices

To avoid mistakes during the conduction of our study we identified several bad practices in the following literature.

In an empirical investigation, Behroozi et al. [BSBP20] identified poor practices in the hiring process where otherwise qualified candidates are lost due to various reasons and provided guidelines for future improvements. Using the website Glassdoor as a resource of anonymous reviews on companies hiring pipelines, each step of the hiring process has been analyzed for risks, identifying bad practices from the initial contact, preparation and scheduling, interviews, hearing back as well as offer and negotiation. Among the highest impact on bad practices have *inexperienced interviewers*, *bad communications of the hiring criteria* and *ghosting*[1] candidates. [BSBP20].

Another study conducted by Behroozi et al [BPB19], based on the feedback of interviewees taken from Hacker News, a social website for software developers, states that technical interviews are *biased towards younger candidates*, *require practice* to do well in and may even *cause anxiety* and *frustration* as well as having little to *no real-world relevance* [BPB19].

[1] Ghosting, in the professional environment, is the process of *not* contacting and updating the candidates on their application status after the conduction of interviews

In a study conducted by Ford et al.[FBRP17] gathering both, qualitative and quantitative data about technical interviews, they found that software engineer job candidates often do not succeed despite correctly answering most questions as well as solving most challenges presented [FBRP17]. According to the study, this is primarily because many candidates underestimate the value the interviewer places on interpersonal skills and proper communication. The study was conducted using mock-up interviews with verified interviewers from nine different companies, gathering and analyzing qualitative and quantitative data based on the feedback of the interviewers. The primary goal of the study was to determine whether or not there are differences between companies in their interview criteria and how the interviewers interpret these criteria. Determining the success of the interview was based on six criteria, with problem-solving only being one of them. *Body language*, *clear communication*, *concrete examples* to back up statements about themselves, their *enthusiasm* displayed as well as *confidence* in their abilities were just as important. The study concludes that most companies have consistent expectations for candidates and that interviewers care about interpersonal communication just as much as their technical abilities. The difference between what the candidate prepared for and what the interviewer was looking for became apparent, especially considering that the expectation and interpretation from different interviewers varied [FBRP17].

The Media used in Technical Interviews

Apart from the interviewer, the medium provided to the candidate is also important. As typical media, we identified whiteboards, pen&paper, integrated development environments (IDEs) with limited to full access to features like coding assistance, refactoring, and testing tools. External sources e.g, documentation, guides, and internet research are typically cut out of the process. In a pilot study, Behroozi et al. found out that whiteboards can cause excessive stress and cognitive load compared to solving a task on a paper sheet [BLM+18].

2.3 Behavioural Traits and Personality

Research in the past forty years has shown that personality has a major impact on job performance.

2.3.1 Evolution of behavioral software engineering

Over the years, scientific research specifically tailored to software engineering has constantly improved their methodological approach to gather relevant data, gain meaningful insights, and to draw the right conclusions. However, to this day, many different approaches, practices, and personality tests are used and contradictory results in similar studies are no rare sight. Although this issue is widely known, it has not properly been addressed, many scientists have started using personality tests developed from more established theories like the Myers-Briggs Type Indicator or the Five-Factor Model instead of their own constructs, which studies clearly benefit from.

Scientific research does currently not agree on the specifics, but the general consensus is that behavioral traits and personality have a major impact on job performance.

Even though human aspects of software engineering have been acknowledged for some time, and studies on the matter have been conducted since the 1970s, the subject is still considered immature, especially regarding some concepts receiving more attention than others [CSC15] [LFW15]. One of the causes might be the industry, studies of the sort rarely yield tangible results. Focusing on short-term profits to satisfy investors seems to be more important than a long-term improvement in their development process. in the recent past a new term, Behavioral Software Engineering (BSE), has surfaced, describing research aimed at exploring cognitive, behavioral, and social aspects of software engineering performed by individuals, groups, or organizations. The distinction between Human-Computer Interaction (HCI) and BSE is the object of focus, BSE focuses on the engineers during the development process, while HCI focuses on the computer system and software from a user perspective [LFT+17]. Also, multiple studies reviewed past research and identified knowledge gaps and the need for collaboration between social science and software engineering to yield adequate results [LFW15] [CSC15].

Lenberg et al. [LFW15] performed a professional search on the ISI Web of Science on the number of articles published on certain topics in software engineering and found out that a staggering 70% of the research was about technological advancements or process-related topics while less than 5% were of human-related themes. On the positive side, the amount of human-related topics has been steadily increasing for the past decades. From 1997 to 2015, more than 250 BSE related research topics have been published [LFW15].

2.3.2 Methods of measuring behavioural traits and personality

Motivated by wild inconsistencies within the results of scientific research on behavioral software engineering, Cruz et al. [CSC15] performed a systematic review on 19000 studies published between 1970 and 2010, analyzing approaches and comparing facts based on the studies results. To do so 90 articles have been, in part automatically, in part manual labor, selected to give a deeper insight in a very broad set of context, while maintaining a high representativity. Analyzing relevant quantitative data (e.g temporal view, number of pages) as well as qualitative data, retrieved from the content of the articles, they manage to show recent trends in both, theoretical and empirical research. Cruz et al. [CSC15] identified that the majority (57%) of researchers chose personality tests based on Jung's Personality Types Theory, the Myers-Briggs Type Indicator (48%), or the Kersey Temperament Sorter (9%), while 19% used the Big Five (BF) or Five-Factor Model (FFM) personality tests like the NEO-PI, increasing in popularity recently. Other personality test included the Rotter Internal-External Control Scale (Rotter I-E), Rathus Assertiveness Schedule (RAS), Thurstone Temperament Schedule (TTS), Rosenberg's Self-Esteem Scale (RSES), Judge's Generalized Self-Efficacy Scale (JGSE), Levenson's Locus of Control Scale (LLC), Personality Type A/B, Self-Monitoring of Expressive Behaviour (SMEB), Hostility Inventory (HI), Type A Behaviour, Minnesota Multiphasic Personality Inventory (MMPI), Personal Resilience Questionaire (PRQ) and Personality Research Form (PRF) [CSC15].

The Myers-Briggs Type Indicator (MBTI) is, as previously discussed, one of the most frequently used tools to assess personality types to this day, especially in the field of consultancy and training world [Fur96]. Developed by Kathrine Cook Briggs and Isabel Briggs Myers based on Jung's Personality Types Theory, the test not only provides the type of personality but also gives plentiful insight into the meaning behind the result. Technically the MBTI splits personality into four dimensions, Extroversion(E) and Introversion(I), Sensing(S) and Intuition(N), Thinking(T) and

Feeling(F), Judging(J) and Perceiving(P), which leads to sixteen possible combinations of distinct personalities. The personality test itself has different forms, however, it is usually a questionnaire consisting of 50 to 150 questions answered on a five- or seven-point Likert scale. The concept behind the MBTI has proven to be reliable, robust, easy to use, providing accurate results and has thus been used in various studies, some of them related to software engineering [Mye] [Mye62] [CA10a] [CA10b] [PY18] [YO12] [Que09]. A handbook for the MBTI has been developed as early as 1944 and has since then steadily improved and is, to this day, one of the most used approaches to analyze the personality traits of a given person. The most recent addition to the MBTI has been Step III from 2009. [Mye62] [Que09]

Another popular approach to assess the personality type of a person is the **Five Factor Model (FFM)**, sometimes also referred to as the **Big Five Inventory**, which, similar to the MBTI, splits the personality into different dimensions. These are Neuroticism (N), Extraversion (E), Openness to Experience (O), Agreeableness (A) and Conscientiousness (C) [RMSA12] [Fur96].

During the 70's many psychologists experimented with three or four dimensions to display an entire personality, however, those models were not able to display the full range of personality traits, which at the time was believed by many psychologists to be impossible [CM08]. "*Language and individual differences: The search for universals in personality lexicons* written by Lewis R. Goldberg [Gol81] was the first to publish an article connecting five individually accepted dimensions, forming what he believed to be a full range of psychological characteristics, which, at the time, received plenty of critique [CM08]. Supported by evidence of several researchers, the construct has stood its test of time and is deemed as one of the more reliable and accurate methods of getting insight into personality today [Chr] [MJ92] [**big**] [BM91].

In 1991, John O.P et al. [JDK91] compiled and simplified the Big Five Inventory, listing the five dimensions, their facets, and correlated trait adjectives and provided a simple 44-item test for their evaluation.

In 2012, Deborah A. Cobb-Clark and Stefanie Schurer [CS12] analyzed the stability of the big five personality traits, and according to their data provided by the Household, Income and Labour Dynamics in Australia over a four year period (2005-2009), the traits do appear to be stable in the working-age group (25-65).

Another, shorter, 15 item version of the BFI, the BFI-S [HGS12] has been developed by Elisabeth Hahn, Juliana Gottschling, and Frank M. Spinath in 2012. It has been evaluated and yielded good results and acceptable levels of all dimensions but the Agreeableness score.

Quite recently, in 2017, Soto and John proposed a second iteration of the BFI, the BFI-2, to further enhance bandwidth, fidelity, and predictive power, however, most current research is still being done using the original BFI since it has been evaluated and proved to be robust for decades [SJ17].

Another highly popular test, incorporating the big five inventory, the **NEO-PI** has been developed during the '80s by Robert R. McCrae and Paul T. Costa [CM08] and was later refined into the **NEO-PI-R** test during 1992, improving its potency and accuracy by introducing scales to agreeableness and conscientiousness.

The NEO-PI-R test has become a tool widely used, specifically in academic research, to assess personality types due to its ease of use, robustness and comprehensive descriptions [CM08] [Fur96]. The only major problem identified with the accuracy and validity of the NEO-PI-R was that children

and teenagers had difficulties in some of the terminology used, which lead to incomplete personality assessments. This had been addressed in 2005 when Robert R. McCrae and Paul T. Costa further improved and simplified the test, which resulted in the **NEO-PI-3** [MCM05].

For the NEO-PI-R, two different ways of administration are provided, Form S, which is used for self-reports, and Form R, which is designed for a third-person observer. Both versions contain 240 items, responded to by a 5-point Likert-Scale, and take 30-40 minutes to complete, which is a major drawback [CM92]. The validity of the test and underlying model was determined by a variety of studies, performed by McCrae and Costa themselves and by various other researchers [CM08].

In the mid-'90s Adrian Furnham performed a direct comparison between the MBTI and the NEO-PI five-factor model using 160 adult participants for his study. Apart from neuroticism, which had a minimal and inconsistent reflection in the MBTI, all remaining four dimensions of the NEO-PI FFM had at one or multiple correlations to MBTI values [Fur96].

A method of getting insight into emotions, focusing on the frequency of thereof, is the **Scale of Positive and Negative Experiences (SPANE)** [Diea]. The original SPANE, developed by Diener et al. in 2009 [DWB+09], is a Questionnaire consisting of twelve items, six of them regard positive feelings while the other six regard negative feelings. Answered on a five-point Likert scale, then aggregated into SPANE-P(ositive) and SPANE-N(egative) values which are used to calculate the Affect Balance (SPANE-B) which in turn gives insight into the users' current state of happiness. Spane has been successfully used in studies both, inside and outside, the scope of software engineering and is regarded as a robust and efficient tool by various sources [Diea][Dieb][RHS17a][RHS17b] [WGW19][GWA14].

2.3.3 Conscientiousness

While The Cambridge Dictionary[2] defines conscientiousness simply as "*the quality of working hard and being careful*"psychologists and social scientists prefer to define it by its specification of behavioral traits, often referred to as facets. Common synonyms for conscientiousness are dependability, will to achieve, self-control, prudence, and constraint, while it's opposite is defined as lack of direction [JDK91] [CM98].

Conscientiousness and the General Mental Ability (GMA) have proven to play a major role in predicting the performance in all kinds of professions [MBS99] [BM91] [Hun80] [Hun86]. The joint relationship between the two has been researched and discussed for the better half of a century. Since the late 1950s, researchers try to create mathematical formulas to describe this effect. Some suggest there might be an interactive formula between motivation and GMA, while others believe they add up additively [MBS99] [Dig90]. The only matter on which scientific research seems to agree on is that conscientiousness has little or no effect on the GMA [MBS99] [Dig90]. Controversially conscientiousness has also been seen to have a negative correlation with Intelligence, one of the many synonyms used for GMA, which further complicates the matter [MFP04]. However, the significance of this relationship is still in a debate, and there is also a debate about whether personal performance on intelligence tests reflects actual intelligence [MFP04].

[2] https://dictionary.cambridge.org/de/worterbuch/englisch/conscientiousness

A detailed investigation into Conscientiousness shows its effect on performance is primarily in two ways. Conscientious individuals are more likely to set goals and plan for future success while also actively and deliberately avoid counterproductive behaviors [MBS99] [BMS93] [BWPO91].

While there is no consensus on a definitive set of behavioral traits for conscientiousness in current research, the BFI and NEO-PI-R FFM indicate that Competence, Order, Dutifulness, Achievement striving, Self-Discipline, and Deliberation are the driving facets of conscientiousness [CM98].

Recent research about which behaviors conscientious persons show conducted by Jackson et al. [JWB+10] display confirms the stereotypes many of us had in mind. "*Individuals are clean and tidy, work hard, follow the rules of society and social decorum, think before acting, and are organized. For example, conscientious people tend to write down important dates, comb their hair, polish their shoes, stand up straight, and scrub floors. Less conscientious people exceed their credit limit, watch more television, cancel plans, curse, oversleep, and break promises*"[JWB+10, p. 507]. However, they also determine that individuals with the "*same latent trait level of conscientiousness .. differ in frequency and type of their behavior.*"[JWB+10, p. 507].

As problem solvers, conscientious people are rumored to be narrow-minded and stick to predetermined processes, while possibly missing solutions outside of the box.[Leh]. However, we did not find a scientific source to confirm this.

Scientific research on the relationship conscientiousness and the performance in tasks relevant to software engineering are scarce. We found one study by Acuña et al. [AGJ09], which found a correlation between job satisfaction, conscientiousness, and agreeableness when working in teams.

2.4 Coding Challenges

Coding challenges, also known as programming challenges, are programming tasks designed to assess the problem-solving ability of a person. Creating a good coding challenge is not an easy task and depends highly on the area of application. However, usually, they have one thing in common, multiple possible solutions in different complexity and run-time classes.

The main areas of application are education and assessment of problem-solving ability. In teaching, those challenges can stimulate and engage students into programming, while also being a possible method of testing and assessing the students. In a regular job, interview assessment is the key, and herein lies the focus.

Major Programming contests such as the International Collegiate Programming Contest (ICPC)[3], the International Olympiad in Informatics (IOI)[4], Googles *Hash Code*[5] and *Code Jam*[6], as well as online competition websites such as *Coderbyte*[7], *HackerRank*[8] and *Edabit*[9] are based on coding challenges to test and challenge their subjects.

The purpose of those competitions, as well as their tasks and difficulties, vary greatly. The IOI, for example, uses different levels of difficulty in their set of challenges with the primary goal to score participants according to their skill level in a highly competitive environment and over a small period of time, currently five days. Other organizations, e.g Edabit, have their primary focus on long-term education and the enhancement of their users' skill levels and experience in different programming languages, various difficulties with a more neighborhood-friendly, however still competitive, environment.

2.4.1 Training and Preparation

Motivated by various statements that programming competitions become progressively harder each year, Michal Forisek [For10] analyzed tasks of major programming competitions quantitatively and qualitatively. The qualitative data gathered were used to find out if the subjective task difficulty rating of contestants while the quantitative data based on the results of the contest were evaluated using Item Response Theory. According to the results, the yearly increase in difficulty is mainly based on the growing popularity of the contests and the continually increasing preparation [For10].

Many detailed guides and books have been written in the recent past to improve the readers' abilities in solving puzzle-like questions currently used in contests as well as in technical interviews of many major companies. Among them are the "*Art of Programming Contest*"[Are06], "*Elements of Programming Interviews*"[ALP12],"*Programming Interviews Exposed*"[MKG12] and "*Cracking the Coding Interview*"[McD19]. While the specifics vary within these books, they all provide plentiful sources of different programming tasks, explain how to tackle them and provide hints where necessary.

2.4.2 Designing Coding Challenges

Burton et al.[BH08] investigated in what they called "the Black Art of Olympiad tasks and hence the hardship coming with designing and creating tasks worthy for the competitive contest environment. They provide a detailed plan on how to tackle this problem and offer extended guidelines on how to design proper challenges for programming contests[BH08].

[3] https://icpc.global/
[4] https://ioinformatics.org/
[5] https://codingcompetitions.withgoogle.com/hashcode/
[6] https://codingcompetitions.withgoogle.com/codejam
[7] https://coderbyte.com/
[8] https://www.hackerrank.com/
[9] https://edabit.com/

With the intent to assess the problem-solving ability accurately using programming contests, Coles et al.[CJW11] provide a deep analysis of the topic, define problem-solving skills, and design programming exercises accordingly. Their contest approach has been evaluated in a 15 participant study with a high degree of variation among the results of the students, revealing that maintaining the participants' motivation was a major issue and needs some sort of incentive. They remain confident the contest approach they have taken can give very accurate results on the general problem-solving ability of participants, however, they acknowledge that the difficulty and style of the tasks, as well as the students' motivation, have to be addressed [CJW11].

2.4.3 Coding Challenges and Education

Coding challenges within the educational environment have been used with great success to attract and motivate students as well as to evaluate and test them. Short summaries are provided in the following paragraphs.

Motivated by lowering the high dropout rates of students in computer science degrees, which according to their research happened mainly due to the complexity of the matter and lack of motivation, García-Mateos et al.[GF09] changed the methodology and structure of their courses, replacing the final exam with a continuous stream of tasks making those tasks, comparable to online competitions, more appealing to students. Automatically evaluating the results using a bias-free online judging tool Mooshak. Based on the qualitative and quantitative analysis in they succeeded, reducing drop-out rates from 72% down to 45% and receiving positive feedback from the majority of students [GF09].

Brad Alexander and Cruz Izu [AI10] admit that post-graduation students usually have very good theoretical knowledge while, for the most part, lacking in the practical application. Their primary concern was to challenge the stronger students while also helping and motivating the weaker ones. They had success, determined by quantitative data gathered on their exams as well as the qualitative feedback given by students, by altering their course structure to offer optional paths to take especially incorporating non-enforced cooperative learning [AI10].

Carbone et al.[CHMG00] studied and analyzed the learning behavior first year programming students to improve the design of programming exercises. Baird identified poor learning tendencies, namely superficial attention, impulsive attention, and staying stuck, which were then analyzed, and evidence was provided. Carbone at al. then improved the tasks by minimizing the risk students fall into those poor learning approaches [CHMG00].

Dagiene et al.[DS04] analyzed major programming competitions and argue that one of the best methods of developing problem-solving abilities is through coding competitions. They have the opinion that programming in schools should be brought back alive since it allows the development of critical and creative thinking to solve problems [DS04]. In further investigation, they analyzed the current decline in school-informatics, which was once very pronounced in the '80s and '90s, and the role competitive programming contests could play general education, especially in attracting, motivating, and inspiring students [Dag10].

Unhappy with the shifting focus from algorithmic problem solving to object orientation, event-driven programming, GUIs, and design patterns, Owen L. Astrachan [Ast04] implemented a non-competitive web-based submission system, based on just-in-time teaching, to solidify students

algorithmic problem solving using small one-method/one-class weekly tasks evaluated by a small number of tests. These tasks usually take about two to three hours per week to complete and provide APIs where necessary, to keep the assignments short. According to him, this method is successful in creating better problem solvers, and programmers and the added experience has a beneficial effect[Ast04]. After investigating different puzzles and analyzing algorithms in regards to their efficiency, Anany Levitin [Lev05] concludes that puzzles are very suitable for teaching algorithmics, improving general solving techniques, and are a valuable tool for algorithm analysis [Lev05].

2.4.4 Automatic scoring of Coding Challenges

Since human scoring of assignments is slow, tedious and error-prone many coding challenge providers have implemented automatic grading systems. Short summaries are provided in the following paragraphs.

Handling larger programming contests manually can be a logistical nightmare, requires a lot of staff to judge solutions, and without clear guidance, the marking accuracy can suffer and is potentially biased. Mooshak, developed by José Paulo Leal and Fernando Silva [LS03], is a web-based system to manage and judge programming contests in a highly scalable fashion. Code evaluation is done simply by testing for input and output and then marked after the all or nothing principle. [LS03].

Another take on alternative methodologies in introductory computer science courses is the Code Mangler. A fictional character used to modify well-written code, rearranging lines, messing up the indentation, removing comments, adding bugs, and more as an alternative to tasks simply asking to generate code or pseudocode. Cheng et al. investigated the marking speed, the confidence of teaching assistants, and how accurately marks resemble the students' abilities between different solutions and found code written from scratch as well as mangled code to be the question types superior to others in regards to the criteria [CH17].

Cheang et al. propose a system, an online judge, which evaluates electronically submitted assignments automatically. They analyzed human grading behavior and identified issues and hardships to use this information designing and improving their automatic grading system. The chosen criteria enveloped correctness, efficiency, and maintainability. While grading correctness and efficiency were fully automated the grading on maintainability stayed in the hands of the lecturers, since this topic is highly subjective and hard to translate into quantitative, measurable data. The judge was then evaluated in three programming courses, from beginner to advanced training, showing that differences in difficulty pose no issues. A positive aspect of this system is that students can complete shorter tasks regularly, compared to a few rather large projects throughout their courses, alleviating stress on both, lecturers and students [CKLO03].

Investing in objective scoring for computation competition tasks, Kemkes et al.[KVC06] debate whether the current scoring system is the right approach for the IOI. Solutions within the IOI are automatically evaluated using tests regarding time and memory limits. Within the current scoring system, failed tests lead to a solution receiving a partial score, however at the time of programming the contestants have no clue how high that might be. This issue was addressed in 2004 by implementing a 50% rule stating criteria with when fulfilled, the user receives a score minimum of 50%. Kemkes et al. [KVC06] use Item Response Theory to investigate the difficulty and discrimination of the IOI tasks and evaluate them using different automated scoring schemes. The motivation behind this is the huge gap between contestants, with many receiving an unpredictable

partial scoring for any of their solutions. They propose a combined scoring scheme which separates the tests into batches according to their difficulty, and awards points for each batch successfully run. This allows for students of all ability levels to predict their potential score and plan accordingly [KVC06].

2.4.5 Parsons Programming Puzzles

Parsons programming puzzles and other similar learning environments were originally introduced to the programming community as an alternative to writing code specifically designed to appeal first-year students in introductory programming courses. Using predetermined blocks of code, which then have to be moved around by the user to form a program that is capable of solving the question. They are powerful tools allowing for an engaging environment and interesting problems and immediate feedback. The process of solving algorithmic tasks is usually of great interest, however, invisible to the lecturer. Using Parsons Puzzles, Helminen et al. propose a method to gain insight into their students' thought processes by analyzing a recorded trace of the solving process. They focused on and categorized how their students arrived at a solution, using various tools, detecting positive and negative solving patterns, analyzing how often students made use of the immediate feedback provided. Also, a valuable visualization approach for the solving process is provided. The intent behind this work was to provide better feedback in the immediate and automated feedback feature, not only on the current state but on the entire process, enabling students to improve their thinking process and avoid negative solving patterns [HIKM12].

2.4.6 Good Coding Challenges

As a result of our literature research, we have identified several key components and features, good coding challenges must have. Firstly, the challenges have to have an easy-to-read, easy-to-understand, non-ambiguous task description. Secondly, a good coding challenge has to have several different solutions, especially for different run-time classes. Thirdly, in a technical interview environment, the solutions must have straightforward implementations. For our evaluation, we decided against incorporating any automatic evaluation tools. However, we plan on using predetermined test-cases to evaluate the solutions our participants provide us with.

3 Methodology

The objective of this thesis to explore the relationship between conscientiousness and performance in coding challenges. In order to achieve this, we designed a study of qualitative and exploratory nature which we propose in the following section. In Section 3.2 we discuss the participants and in Section 3.3 we describe the technical details of the study. In addition, we describe the coding challenges and provide possible solutions in Section 3.4. Furthermore, we describe our data collection in Section 3.5 and the analysis of our data in Section 3.6.

3.1 Study Design

To answer the research questions we designed an exploratory qualitative study, incorporating three coding challenges which had to be solved, three interviews, two in between the challenges and one after all the challenges had been completed, a personal questionnaire gathering data such as experience levels, their subject of study, their current grades, their frequency of positive and negative emotions and a Big Five personality test. The study was conducted online, recording both, the solving process of the coding challenges and the intermediate and final interviews. The questionnaires were filled out in private after the final interview had been conducted. As typical for this type of study, we do not have any well-defined hypothesis which we could have tested for, but rather a set of guidelines and ideas to gather qualitative rich data and derive hypotheses from these data [Max08].

Since this study is strongly related to technical interviews used in the application process of well-known companies, we restricted the use of the internet during the interview solving process accordingly, not allowing the participants to do any online research. The only two sources of additional information and help during the interview process were hints provided by the interviewer and all features, specifically the Java documentation, offered by their integrated development environment (IDE). The Java documentation in particular was allowed to use for two reasons. Firstly, in any real technical interview participants usually come prepared, for a study, however, this is not the case. Secondly, students come across many programming languages over the course of their education, which in turn means they often have trouble recalling the exact syntax for a specific language.

To motivate potential candidates to take part in this study, we explained to them how strongly related this study is to the application process, especially technical interviews, of many major software developing companies and thus the experience might be worthwhile their time. We also told them the study would take about one and a half hours of their time, involves solving three coding challenges in a given time frame, and requires a laptop or pc, a microphone, and a stable internet connection. After the brief introduction, we asked interested candidates to enter an online poll to choose their preferred day and time using their names and email addresses as credentials. For simplicity, we

used predetermined time slots, with the option of customization by contacting the interviewer. After entering the online poll, a follow-up email was sent, with additional information on the technical details of the interview, such as the requirement to download Zoom[1] and either IntelliJ IDEA[2] or Eclipse[3] as IDE, which could be downloaded prior or during the study. Further, we sent them a study introduction script, which can be found in the appendix, describing the structure of the study, the rules, the scoring of their solutions, and a consent form. To ensure every participant had the same information we used the same invitation and follow-up email as well as an introduction script for each participant.

The study was conducted one interview at a time with one researcher and a single participant. This ensured the consistency of the interviews as well as the attention and possible assistance the individual participants received.

Before the recording of the interview, the participant received a folder containing the coding challenges, a second copy of the introduction script which was sent by email before, and several links including the download links for all necessary software. The introduction script was then traversed through by the interviewer and participant together, encouraging the participants to ask questions where needed.
Within the introduction script, we provided the participant with the scoring method, how his or her solution will be judged after completion. The criteria for the algorithmic solutions were, descending in importance, correctness, time-complexity, and elegance, explicitly excluding space-complexity. The participants were also told that an early submission will not result in any advantage and focusing on potential improvements to the three previously stated criteria might be more beneficial in their remaining time. Additionally, the participants were enlightened, in case they provided multiple solutions they had to choose which one to hand in, commenting out the others.

At the beginning of the recording, the participant was asked six relevant questions. If they had any additional questions, what their subject of study is or was if they had any experience with Java and the IDE in use (IntelliJ IDEA or Eclipse) if they had any questions about the consent form and finally they were asked to give their verbal consent.

After the formal consent had been given, the interviewer and participant proceeded to import the first challenge to the chosen IDE explaining the structure of the challenges. Every challenge was comprised of at least a runnable Java environment, the main method, and an explanatory comment. The comment described the problem the participant had to solve in detail and stated the time frame in which they had to do so. The first challenge did not contain anything further since everything needed to solve the problem was possible within the main method, without any parameters or return values, to give the participant a high degree of freedom in design.

Since challenges 2 and 3 were a little more complex, we provided a little framework, an additional method, specifying the parameters and return values which the participant was not allowed to change. We also provided the user with a little Input/Output example within the descriptive comment. Further information about the challenges can be found in section 3.4.

[1] https://zoom.us/
[2] https://www.jetbrains.com/de-de/idea/
[3] https://www.eclipse.org/

After a brief discussion between the participant and interviewer, making sure the participant understood the challenge correctly, the timer was set accordingly and started. As previously mentioned the participant was not allowed to change the given signatures, the parameters, and return values. They were allowed to create additional methods and encouraged to use alternative data structures if they made sure to translate those back into the originally intended and provided return values. This was primarily done to not give unintended hints about possible optimizations to the runtime complexity of their code while still providing some kind of helping framework. The participants were also allowed to run the code whenever they chose to, to take notes, in paper or digitally, to support their thought process, and to ask questions.

An image describing the structure the study with all of its components can be found in Figure 3.1

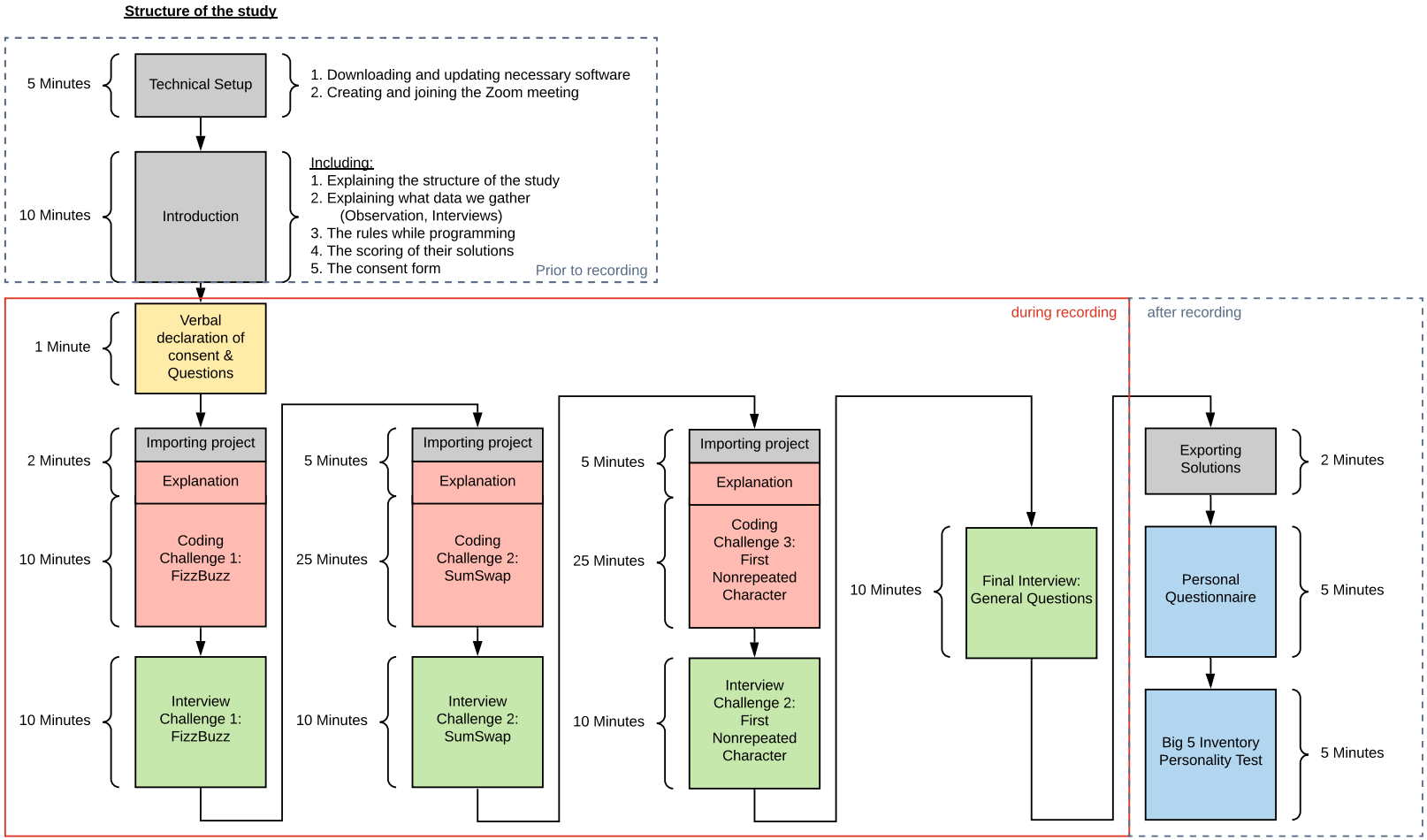


Figure 3.1: Structure of the Study

3.2 Participants

As potential candidates, we preferred soon-to-be graduated students and students who had graduated already, with the focus that they benefit from the experience the most as well ensuring they were experienced enough. The minimum requirement to participate in the study was that the candidates had at least visited lectures on data structures, algorithms, and time-complexity. We settled with 12 participants, one post-grad computer scientist, two students of media computer science, and the remaining nine being software engineering students currently working on their final thesis or who already had graduated in the recent past. None of the participants were forced to participate to receive any credit nor did they receive any compensation other than the experience gathered during the interview. All of the students had been invited personally by email before the conduction of the first interview.

3.3 Technical Details

For the recording process, different screen-sharing programs were evaluated. We considered Skype[4] and Discord[5] for the sessions and screen-sharing, having in mind their popularity and ease of use. The issue we encountered was the missing feature of recording such a session in a way that fits our requirements. This could have been solved by using a recording software such as Open Broadcaster Software (OBS)[6], however, maneuvering through different programs while conducting an interview seemed to be unnecessarily stressful and prone to error. After having gone through other software products we finally settled for Zoom™[7] since it had the capability of recording entire screen-sharing sessions using very little resources, only required a free account to operate and the download was small in size.

To further reduce the technical setup time for the participants we chose two for Java commonly used IDEs, namely Eclipse[8] developed by the Eclipse Foundation®and IntelliJ IDEA[9] from JetBrains. We built our Coding Challenge projects for those two IDEs, letting participants choose which of them they want to work in, handing out the proper projects.

3.4 Coding Challenges

To gain insight into the participants' solving process we chose three different coding challenges. As previously discussed in 3.1 the participants had to solve one challenge at a time using either of the two suggested IDEs within a given time frame. Our decision in which coding challenges to choose was based on our research in Section 2.4. First and foremost the challenges had to be easily understandable while still enabling the participants to provide different solutions in either elegance,

[4] https://www.skype.com/de/
[5] https://discord.com/
[6] https://obsproject.com/de
[7] https://zoom.us/
[8] https://www.eclipse.org/
[9] https://www.jetbrains.com/de-de/idea/

time-complexity, or both. Secondly, we put our emphasis on challenges that had a straightforward implementation for most of its concepts, especially regarding time-complexity. Thirdly, we made sure that most of the solutions could be further improved upon in elegance and/or time-complexity without the need of starting over or changing too much in the already written code. Fourthly, based on the context of the study, we didn't want to deviate too far from actual technical interviews and we made sure to choose coding challenges that are used in practice. And last but not least, since the focus of this thesis lies in how coding challenges are solved and not if, we chose challenges with appropriate difficulties.

To make sure the participants were able to solve the challenges within the restricted time frame, we piloted the study twofold. In the first iteration the interviewer solved the challenges himself and well within time, however, since he was part of the selection process of these coding challenges, he knew about the possible ways to solve them. In the second iteration we had a student from the targeted candidates, who did not have trouble solving the three challenges in time as well.

The following sections describe each individual coding challenge and provide different concepts and solutions.

3.4.1 Coding Challenge 1: FizzBuzz

For our first coding challenge we chose the FizzBuzz challenge, which is currently the most widely used challenge in the evaluation of software developers during their application process. The challenge is easy to understand and has many straight forward implementations which can be further improved. We set the time limit for this challenge to 10 minutes, which left plenty of time for potential improvements after the initial implementation. FizzBuzz itself is a game about division. As shown in Listing 3.1 one counts up the numbers from 1 to a predetermined value, in our case 100. Each number divisible by predetermined number a in our example 3 is replaced by the word *Fizz* and each number divisible by second predetermined number b in our example 5, is replaced by the word *Buzz*. if a the current number is divisible by both, a and b, the output is replaced by the word *FizzBuzz*.

Our focus on possible optimizations for FizzBuzz was less on time-complexity, which is set to $O(n)$, but rather on elegance, for example, reducing the amount if-statements, or to outsource predicates into their own methods with proper names to improve on expandability and/or maintainability.

```java
public class Main {

    /**
     * Challenge 1: FizzBuzz
     *
     * Write a method to print the numbers 1 to 100.
     * For each multiple of 3 print "Fizz" instead of the number.
     * For each multiple of 5 print "Buzz" instead of the number.
     * For each multiple of both, 3 and 5, instead print "FizzBuzz".
     *
     * Timelimit: 10 minutes.
     */

    public static void main(String[] args) {
        //TODO
    }
}
```

Listing 3.1: Coding Challenge 1: FizzBuzz

The straightforward implementation, described in 3.2, is a simple for-loop generating the numbers 1 to 100 filtered by an if-else statement using the modulo operator. In this solution, the first statement has already been optimized to $(i\%15 == 0)$ which is the smallest common divisor of 3 and 5 to check for one argument rather than two.

```java
    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++){
            if (i % 15 == 0){
                System.out.println("FizzBuzz");
            } else if (i % 3 == 0) {
                System.out.println("Fizz");
            } else if (i % 5 == 0) {
                System.out.println("Buzz");
            } else {
                System.out.println(i);
            }
        }
    }
```

Listing 3.2: FizzBuzz Solution 1: Simple loop in $O(n)$

The second solution, described in 3.3, optimizes the previous solution by extracting and outsourcing the predicates into methods, improving readability. More importantly, it improves maintainability by reducing the places the variables, which might be subject to change in the future, from multiple to one.

```
    private static boolean fizz(int i) {
        return i % 3 == 0;
    }

    private static boolean buzz(int i) {
        return i % 5 == 0;
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 100; i++){
            if (fizz(i) && buzz(i)){
                System.out.println("FizzBuzz");
            } else if (fizz(i)) {
                System.out.println("Fizz");
            } else if (buzz(i)) {
                System.out.println("Buzz");
            } else {
                System.out.println(i);
            }
        }
    }
```

Listing 3.3: FizzBuzz Solution 2: Outsourced predicates in $O(n)$

Another version of this maintainability improvement can be seen in 3.4, by declaring and appending to an output string rather than using simple *System.out.println()*. Further, the String version leads to a reduction in if-statements which increases the performance marginally.

```
    public static void main(String[] args){
        for (int i = 1; i <= 100; i++){
            String output= "";
            if  (i % 3 == 0){
                output += "Fizz";
            }
            if (i % 5 == 0) {
                output += "Buzz";
            }
            if (output ==""){
                output = String.valueOf(i);
            }
            System.out.println(output);
        }
    }
```

Listing 3.4: FizzBuzz Solution 3: String appendage in $O(n)$

As alternative to a *for-loop* and multiple *if-statements* a *Stream* can be used as shown in 3.5. The resulting code is a lot shorter in size, however, an argument can be made that it impairs readability, especially towards less experienced programmers. Additionally creating an *Stream* for such a small task can lead to a loss in performance.

```
1  public static void main(String[] args) {
2      IntStream.rangeClosed(1, 100)
3              .mapToObj(i -> i % 3 == 0 ? (i % 5 == 0 ? "FizzBuzz" : "Fizz")
4                      : (i % 5 == 0 ? "Buzz" : i))
5              .forEach(System.out::println);
6   }
```

Listing 3.5: FizzBuzz Solution 4: Stream in $O(n)$

3.4.2 Coding Challenge 2: SumSwap

For the second challenge, we chose the Integer Sum Swap challenge which is featured in "*Cracking the Coding Interview*"[McD19] and is of moderate difficulty. After a testing phase, we decided a time limit of 25 minutes is appropriate. This guarantees the majority of the participants have enough time to complete the challenge and potentially improve their solution.

The challenge is about a mathematical problem, given two arrays, the participant is supposed to implement a program that finds a pair of values, one from each array, so both arrays have the same sum if the values were to be swapped. The participant did not have to swap the values within the arrays, just to find and return them. The result was returned as an array, with the first value being a specific value to swap from *array1* and the second value being a specific value to swap from *array2*. We provided the input and output arrays and the fact that only positive integer values would occur within the arrays. This was done to simplify the solving process in the context of handling edge cases. We also provided a framework to determine the Parameters, in this case, a result array, as well as the output, and provided a simple for-loop to print the resulting array. The description provided to the participants is shown in 3.6

```java
public class Main {

    /**
     * Challenge 2: Sum Swap
     * Implement a program that given two arrays filled with positive integer values finds a pair
     * of values (one from each array) and swaps them so both arrays have the same sum.
     *
     * E.g Input: {1,1,5,2,1} and {5,2,5}
     *     Output: {1,2}
     *
     * Timelimit: 25 minutes.
     */

    public static void main(String[] args) {
        int[] array1 = {1, 1, 5, 2, 1};
        int[] array2 = {5, 2, 5};
        int[] result = sumswap(array1,array2);
        for (int i = 0; i < result.length; i++) {
            System.out.println(result[i]);
        }
    }

    public static int[] sumSwap(int[] array1, int[] array2) {
        int[] result = {0,0};
        //TODO
        return result;
    }
}
```

Listing 3.6: Coding Challenge 2: Sum Swap

In order to solve this challenge, the participant had to find and solve the rather simple but crucial mathematical formula behind it. The sum of both values post swapping values calculates as following $postSum1 = sum1 - val1 + val2$ and respectively $postSum2 = sum2 + val1 - val2$. Since the sum of the arrays was supposed to be the same after swapping the pair the formula resulting was sum1 - val1 + val2 = sum2 + val1 - val2, which could be further reduced to $val1 - val2 = \frac{sum1 - sum2}{2}$. Possible solutions exist for three different time-complexity classes. Two versions of simple brute force Algorithms, described in Listing 3.7 and Listing 3.8 are within the time-complexity class of $O(n \cdot m)$ Both require a nested for-loop for each array, where every possible pair is checked for its validity. The only relevant difference between Listing 3.7 and Listing 3.8 is the outsourcing of the targeting method which makes future improvements easier.

```java
    public static void main(String[] args) {
        int[] array1 = {1, 1, 5, 2, 1};
        int[] array2 = {5, 2, 5};
        int[] result = sumswap(array1,array2);
        //print results
        for (int i = 0; i < result.length; i++) {
            System.out.println(result[i]);
        }
    }

    public static int[] sumswap(int[] array1, int[] array2) {
        int[] result = {0,0};
        int sum1 = IntStream.of(array1).sum();
        int sum2 = IntStream.of(array2).sum();

        for (int i = 0; i < array1.length; i++) {
            for (int j = 0; j < array2.length; j++){
                int newSum1 = sum1 - array1[i] + array2[j];
                int newSum2 = sum2 + array1[i] - array2[j];
                if (newSum1 == newSum2){
                    result = new int[] {array1[i], array2[j]};
                }
            }
        }
        return result;
    }
```

Listing 3.7: Sum Swap Solution 1: Blind Brute Force in $O(n^2)$

```java
public static void main(String[] args) {
    int[] array1 = {1, 1, 5, 2, 1};
    int[] array2 = {5, 2, 5};
    int[] result = sumswap(array1,array2);
    //print results
    for (int i = 0; i < result.length; i++) {
        System.out.println(result[i]);
    }
}

public static int[] sumswap(int[] array1, int[] array2) {
    int[] result = {0,0};

    //Target Brute Force
    Integer target = getTarget(array1, array2);
     if (target == null){
     return result;
     }
    for (int i = 0; i < array1.length; i++){
        for (int j = 0; j < array2.length; j++){
            if (array1[i] - array2[j] == target){
                return new int[] {array1[i], array2[j]};
            }
        }
    }

static Integer getTarget(int[] array1, int[] array2) {
    int sum1 = IntStream.of(array1).sum();
    int sum2 = IntStream.of(array2).sum();

    if ((sum1 - sum2) % 2 != 0) {
        return null;
    } else {
        return (sum1 - sum2) / 2;
    }
```

Listing 3.8: Sum Swap Solution 2: Targeted Brute Force in $O(n^2)$

Another solution incorporates sorting both arrays and traversing through them according to the difference of the current values as shown in Listing 3.9. In a *while-loop* both arrays are traversed through at once, this can be pictured as a modified Turing machine with two input tapes, which are the two arrays. If the difference is currently too small for a valid pair, move the upper tape position one to the right to find a bigger difference. If the difference is too big move the lower tape position to the right to find a smaller difference. If both arrays were to be provided in a sorted manner such a solution would be in linear time $O(n)$, if not the individual arrays have to be sorted in advance resulting into a solution of linearithmic time-complexity $O(n \log n + m \log m)$. This is a significant improvement over the two previously discussed solutions.

```java
public static int[] sumswap(int[] array1, int[] array2) {
    int[] result = {0,0};
    Integer target = getTarget(array1, array2);
    if (target == null){
        return null;
    }
    result = findDifferenceSorted(array1, array2, target);

    return result;
}

static Integer getTarget(int[] array1, int[] array2) {
    int sum1 = IntStream.of(array1).sum();
    int sum2 = IntStream.of(array2).sum();

    if ((sum1 - sum2) % 2 != 0) {
        return null;
    } else {
        return (sum1 - sum2) / 2;
    }
}

static int[] findDifferenceSorted(int[] array1, int[] array2, int target){
    int a = 0;
    int b = 0;
    Arrays.sort(array1);
    Arrays.sort(array2);

    while (a < array1.length && b < array2.length){
        int difference = array1[a] - array2[b];
        /*  Compare difference to target. If difference is too small, then make it bigger
            by moving a to a bigger value, if difference is too large, make it smaller
            by moving b to a bigger value. if it's just right, return the two values.
         */
        if (difference == target){
            int[] result = {array1[a], array2[b]};
            return result;
        } else if (difference < target) {
            a++;
        } else if (difference > target) {
            b++;
        }
    }
    return null;
}
```

Listing 3.9: Sum Swap Solution 3: Targeted and sorted in $O(n \log n)$

The most optimal solution for the Integer Sum Swap challenge is the use of alternative data structures, in this case, a *Hashset* since it offers a lookup time of amortized $O(1)$. The first logical step is to write all values of either array2 (alternatively array1) into the *Hashset*. Secondly we replace the *for-loop* of the chosen array which we translated to a *Hashset* and rearrange the formula from $val1 - val2 == target$ to $val2 = val1 - target$. If the *Hashset* and thus our previous array 2

contains the value we are looking for we found a valid pair and can abort the process. This solution falls into the linear time-complexity class $O(n+m)$ and since we have to touch each element at least once it is the Best Conceivable Runtime (BCR) [McD19].

```java
public static int[] sumswap(int[] array1, int[] array2) {
    int[] result = {0,0};
    Integer target = getTarget(array1, array2);
    if (target == null){
        return result;
    }
    result = findDifference(array1, array2, target);
    return result;
}

static Integer getTarget(int[] array1, int[] array2) {
    int sum1 = IntStream.of(array1).sum();
    int sum2 = IntStream.of(array2).sum();

    if ((sum1 - sum2) % 2 != 0) {
        return null;
    } else {
        return (sum1 - sum2) / 2;
    }
}

static int[] findDifference(int[] array1, int[] array2, int target){
    HashSet<Integer> contentsOfArray2 = getContents(array2);
    for (int i = 0; i < array1.length;i++){
        int two = array1[i] - target;
        if (contentsOfArray2.contains(two)){
            int[] result ={array1[1], two};
            return result;
        }
    }
    return null;
}

static HashSet<Integer> getContents(int[] array){
    HashSet<Integer> set = new HashSet<Integer>();
    for (int i = 0; i < array.length; i++){
        set.add(array[i]);
    }
    return set;
}
```

Listing 3.10: Sum Swap Solution 4: Hashset in $O(n)$

3.4.3 Coding Challenge 3: First Nonrepeated Character

As third challenge we chose a *String* based challenge featured in "*Programming Interviews Exposed*"[MKG12]. Since the source [MKG12], did not specify a difficulty, we performed some testing and categorized it as moderately difficult. We considered using challenges of harder difficulties which is frequently done in similar studies as well as real technical interviews, however, since this

study is focused on the observation of the *solving process* of the participants our intention was to ensure participants had a reasonable chance at solving the challenge. Additionally, we did not want to exceed our pre-selected time limit of one hour for all three challenges which left us only 25 minutes to work with. We could have extended the time for the study in general but we figured the worst-case scenario of two hours was already stretching the limits of the participants. The *First Nonrepeated Character* challenge which alternatively is called *First Unique Character* has been presented to the participants as displayed in Listing 3.11

```java
public class Main {

    /**
     * Challenge 3: First Nonrepeated Character
     * Write a method to find the first nonrepeated character in a given string.
     *
     * E.g Input: dadooriato
     *     Output: r
     *
     * Timelimit: 25 minutes.
     */
    public static void main(String[] args) {
      String sample = "dadooriato";
        System.out.println(firstUniqueChar(sample));
    }

    public static String firstUniqueChar (String sample){
        String result = "";
        //TODO
        return result;
    }
}
```

Listing 3.11: Coding Challenge 3: First Unique Character

The most simple way of solving the *First Nonrepeated Character* challenge is to simply iterate twice over the given String $O(n^2)$ checking for duplicates with a boolean variable, which can be seen in Listing 3.12.

```java
public static void main(String[] args) {
    String sample = "dadooriato";
      System.out.println(firstUniqueChar(sample));
  }

  public static String firstUniqueChar (String sample){
      String result = "";


      for (int i = 0; i < sample.length(); i++){
          boolean duplicate = false;
          for (int j = 0; j < sample.length(); j++){
              if (sample.charAt(i) == sample.charAt(j) && (i != j)){
                  duplicate = true;
                  break;
              }
          }
          if (!duplicate){
              return String.valueOf(sample.charAt(i));
          }
      }
      return result;
  }
```

Listing 3.12: First Unique Character Solution 1: Brute Force in $O(n^2)$

A better solution can be achieved using alternative data structures such as a *Hashmap* as described in Listing 3.13. First, all the values of the String have to be converted into said *Hashmap* using a simple *for-loop*. The *Hashmap* itself can be implemented in multiple ways. We chose to simply to count the letters of the *String*. Then a second iteration over the *String* has to be initiated, checking each letter in its correct order, if the current letter is found within the *Hashmap* with an occurrence of exactly one, we found a solution and can return the letter. Since each element has to be looked at at least once this solution is within the best conceivable runtime (BCR) of $O(n)$ and is thus an optimal solution. There are other approaches, for example to only count the duplicates within the *Hashmap*, however, this only reduces a few write-to operations which are amortized to be within $O(1)$. There are a variety of different *Hashmap* solutions for this problem which all have their upsides and downsides, such as possibly worse readability without the use of comments, worse or better maintainability, and the reduction of operations.

```java
public static void main(String[] args) {
  String sample = "dadooriato";
    System.out.println(firstUniqueChar(sample));
}

public static String firstUniqueChar (String sample){
    String result = "";

    HashMap<Character, Integer> char_counts = new HashMap<>();
    //fill hashmap
    for (int i=0; i < sample.length(); i++){
        char c = sample.charAt(i);
        if (char_counts.containsKey(c)){
            char_counts.put(c, char_counts.get(c) + 1);
        } else {
            char_counts.put(c, 1);
        }
    }


     //iterate through string to find occurences
    for (int i = 0; i<sample.length(); i++) {
        char c = sample.charAt(i);
        if (char_counts.get(c) == 1) {
            result = String.valueOf(c);
            return result;
        }
    }
    return result;
}
```

Listing 3.13: First Unique Character Solution 2: Hashmap in $O(n)$

In Listing 3.14 we present a second optimal solution. This solution does not require the incorporation of alternative data structures, instead, an array, which is often referred to as alphabet array, is used. This approach requires the letters to be converted to numbers, which can be natively done by using the ASCII values of each respective letter and subtract the ASCII value of a from it. Alternatively one could also map the letters to arbitrary numbers in ascending order, which would make another iteration over the array necessary. This has no impact on time-complexity. The array itself counts the occurrence of the letters, comparable to the Hashmap seen in Listing 3.13. Another iteration is then needed to again traverse the original *String* to find the first letter which occurs exactly once in the alphabet array.

```
1   public static void main(String[] args) {
2     String sample = "dadooriato";
3       System.out.println(firstUniqueChar(sample));
4   }
5
6   public static String firstUniqueChar (String sample){
7       String result = "";
8       int[] char_counts = new int[26];
9       for (char c : sample.toCharArray()) char_counts [c - 'a']++;
10
11      for (char c : sample.toCharArray()) {
12          if (char_counts[c - 'a'] == 1) {
13              return String.valueOf(c);
14          }
15      }
16      return result;
17  }
```

Listing 3.14: First Unique Character Solution 3: Alphabet Array in in $O(n)$

Another approach we want to present is shown in Listing 3.15. The solution consists of only four lines of code. The solution might be short and the readability high, but the *indexOf()* as well as the *lastIndexOf()* both settle in the time-complexity class $O(n^2)$ and thus is no optimal solution

```
1   public static void main(String[] args) {
2     String sample = "dadooriato";
3       System.out.println(firstUniqueChar(sample));
4   }
5
6   public static String firstUniqueChar (String sample){
7       String result = "";
8       for (int i=0; i<sample.length(); i++){
9           if (sample.indexOf(sample.charAt(i)) == sample.lastIndexOf(sample.charAt(i))){
10              return String.valueOf(sample.charAt(i));
11          }
12      }
13      return result;
14  }
```

Listing 3.15: First Unique Character Solution 4: indexOf & lastIndexOf in $O(n^2)$

Since the participants were allowed to ask as many questions as they wanted during the study, we purposely left out some of the edge cases of the requirements and encouraged the participants to ask. An example for this is the input *String* and its constraints in Challenge 3: *First Nonrepeated Character*. We purposely left out the fact that the *String* will only ever contain lowercase English letters, no special letters, no numbers, no signs, and no upper case letters and will not be empty. The intention behind this was, to not give unwanted hints and additional information that might steer the participants into implementing a solution they would not usually consider themselves. Having this information can lead to the participant to specifically think about those restricted constraints rather than a more general approach which might not depend on those constraints at all. However these constraints do open up paths to alternative solutions and if a participant asks, according to our interpretation, he is already considering going that direction.

3.5 Data Collection

To answer the research questions *if there are differences between conscientious and non-conscientious software developers while solving coding challenges* and *how those differences impact the coding challenge performance*, without knowing what those differences could be, we decided to gather a set of rich qualitative data which would enable us to find phenomena in regards to this effect.
For this purpose, we designed the study with two components in mind which would be set up in iterative stages, the coding challenges, followed by semi-structured interviews.
Our theory was the observational part would later help us find qualitative data based on our perception, while the interviews would help us understand the thinking process and approaches of the participants, based on their perception.

To enhance our results and conclusions we decided on gathering additional quantitative data in a follow-up questionnaire. This involved demographic data such as the names and email addresses to provide us with their contact information for potential future questions. Additionally, we gathered data of their age, gender, subject of study, how many semesters they had been studying or had studied, their academic performance as well as experience in commercial software engineering, whether they had previously worked on open source projects or possess a Stack Overflow[10] account. We also assessed their current mood using the Scale of Positive and Negative Experience (SPANE) [Diea][RHS17b][Dieb]. We decided that the current mood, especially happiness, is of great importance for software development, and recent research on happiness and unhappiness of software developers by Graziotin et al. tends to agree [GWA14] [GWA13] [GFWA18]. We are aware, some researchers e.g Rahm et al. [RHS17a] who examined the German version of SPANE, especially its validity compared to other measurements, and criticize particularly the SPANE-B value for the reduction of complexity since there is no scientific evidence positive and negative feelings have a canceling effect on each other. They also raise awareness about measuring frequency, disregarding intensity, and being self-reported. However, they also admit the two-factor structure of SPANE showed good results on psychometric properties and convergent validity [RHS17a].

Research suggests, measuring the experience of programmers in years and education can be misleading and wrong. Thus we extended the questionnaire to additionally ask about professional experience and incorporated two questions where the participants had to evaluate their experience based on comparison to (a) their classmates and (b) their professional colleagues since those questions have been evaluated and recommended by one of our primary sources on the topic written by Siegmund. et al. [SKL+14] who criticized other researchers' methods of measuring experience. The main purpose of the questionnaire was to filter out certain phenomena which were with a high probability related to factors other than conscientiousness, such as for example experience.

The last part of the data collection consisted of a Big 5 Inventory personality test assessing the participants' personality in five dimensions. Those five dimensions are: *extraversion*, *agreeableness*, *conscientiousness*, neuroticism and *openness* to experience. As presented in section 2.3.2 we had the option of three differently modeled personality tests, the Myers-Briggs Type Indicator (MBTI), the Big Five Inventory (BFI), and NEO-PI and its different versions. We chose the Big Five Inventory over its competitors for two reasons. Firstly opposed to the MBTI the BFI grants direct access to

[10] https://stackoverflow.com/

conscientiousness instead of a coded personality. Secondly, the different versions of the NEO-PI e.g the NEO-PI-R have a tremendous amount of up to 240 items. For our participants, this would have meant quite some time spend on the personality test. Since the Big 5 Inventory test only features 44 items which can be answered in a few minutes while still yielding scientifically valuable results [Chr] [MJ92] [**big**] [BM91], we decided to incorporate the Big 5 Inventory test.

We consciously placed the personality test and the personal questionnaire posterior to the conduction of the study, since filling out the questionnaire at an earlier stage might have caused the participant to have a certain perception of what data we were after, which, according to our perception, could cause the participant to act biased or derive from his usual path of solving coding challenges. Additionally, both, the personal questionnaire as well as the personality test were filled out in private.

3.5.1 The interviews

To gather the rich qualitative data, we required to answer the research questions we considered different types of interviews which are discussed in the following.

The choice of the Interview Structure

Firstly we had to decide which kind of interview would yield the best results over the course of our study regarding our inexperience. As a starting point, we used several short guides aimed towards different types of interviews provided by the Robert Wood Johnson Foundation, which is also available on their website [CC08]. We also confirmed their statements using other sources [CC08] [Bla13] [KPJK16]. The different types of interviews we considered were *structured interviews*, *semi-structured interviews*, *unstructured interviews* as well as *informal interviews*.

Structured Interviews are typically used when the topic to be researched has a clear focus and a well-developed understanding of, since neither of those factors fit our scenario we decided to dismiss this type of interview [CC08].

Using *Unstructured interviews*, the interviewer has to have a clear agenda towards the focus and goal of the interview. Since our goal was to find out if there are differences within the solving process of coding challenges between conscientious and non-conscientious individuals, we did not know where this would lead us and hence dismissed this type of interview as well [CC08].

Although *informal interviews* are frequently used to accompany participant observations, we parted from the idea of using them due to the fact our researcher had little experience in conducting interviews and the thought of conducting one without any preparation aimed towards what questions to ask seemed to be an unnecessary risk towards the quality of our data [CC08].

Semi-structured interviews are a balanced combination of some components of the other types of interviews mentioned previously. A clear guide and a small set of broad questions keep the interviewer on track while any interesting information given by the interviewee can be followed up on with additional spontaneous questions. They are frequently used when the researcher only gets one chance to interview the participants. Well executed *semi-structured interviews* can result in comparable, reliable qualitative data, and since a comparison between participants differing in conscientiousness is a crucial part of our analysis we decided on using and preparing for *semi-structured interviews* [CC08].

Preparation for the Interview

After having decided which type of interview to use, we entered the planning stage. Firstly we had to determine which questions to use as a guide for the interviewer and secondly, we had to prepare the interviewer himself with the awareness in mind that the quality of the data we gathered strongly correlates with how the interviews were conducted and prepared for [HA05].

According to our sources [HA05] [Pat90] there are six types of questions commonly used in interviews. (1) Questions based on behavior and experience which yield into a description of the interviewees' experiences, behavior, or actions. (2) Questions based on the opinions or values of the participants that reveal information on how the interviewee thinks or feels towards a certain topic. (3) Feeling based questions which aim towards an emotional response. (4) Knowledge questions asking for facts. (5) Sensory questions that draw information from what the interviewee sees, hears, etc. (6) Questions about the background or demographic used to identify the characteristics of the interviewee [HA05] [Pat90].

We determined that we did not need to prepare questions from categories 4 and 5, since neither knowledge-based questions nor sensory questions would offer us any insight into the participants' approach of implementing coding challenges. Additionally, we figured they wouldn't help to guide the interviewer through the interview. Further, we decided to refrain from background or demographic questions since this is what we used the personal questionnaire for.

Since we were lacking experience in conducting interviews we based our selection of questions, in addition to the previously stated categories, on the opinions and experiences of more advanced interviewers. Especially regarding which questions usually work well and yield good qualitative and rich data, such as letting the user explain how he approached a certain aspect or how he did something in particular, or which types questions to avoid, such as very detailed questions where the user may have trouble recalling or questions assuming the user to have completed a task which might not be the case. Additionally, reflexive questions are deemed to provide good information for example what he could have done differently [HA05].

We also made use of a concept called *Grand Tour Questions* mentioned in "Asking Questions: Techniques for Semi-structured Interviews"[Lee02]. A Grand Tour Question usually asks for a routine of the interviewee such as "*Could you describe a typical day on your weekend?*", something the interviewee most certainly knows and has a description for and which gets them to talk. We were also aware that questions within interviews should always be asked in ascending order towards their threat level while keeping it as low as possible [Lee02].

With this information, we set out to create questions we deemed to be worthy of being asked in each interview, which provide rich information and at the same time guide the interviewer through the process.

In order to keep the consistency needed to gather the valuable qualitative data, we prepared six questions for the three interviews tailored to the coding challenges and four general questions the participants would respond to in the final interview. The questions were, as previously discussed and typical for semi-structured interviews, meant to be an open-minded thread, a starting point, and as that we would generally go into the direction the participants lead us with their responses or to a direction which the participant might have omitted, purposely or not.

We designed our first question as something easy to answer to, something that would put the interviewee at ease, not requiring much, however still providing good information so we decided on "*What was the first thing you thought of when you read the assignment?* "which we believed to be a descriptive question, of category 1, however, we later found out many participants reacted with an emotional response.
As the second question, we thought the user had been warmed up enough and we could now be asked the most important question: "*Can you explain your approach towards the Challenge?*". To ensure this question was answered to a satisfying degree and properly followed up the researcher was prompted to put more emphasis on this question.
Since both previous questions had been of explanatory nature and thus category 1 we decided to change the pace and ask a question of category 3, a feeling-based question "*How did you feel during the Challenge?*". This was done because we believed the emotional state could very well have an effect on the approach the participant chose during the solving process.
As the fourth question we used one of reflective and descriptive nature "*What do you think you've been spending most of your time on?*". This lets the user reflect on what he had done and what he may improve on in the future.
This was followed up with another feeling-based question "*Are you satisfied with your result?*"which gave us the insight of their level of satisfaction. Finally, we asked another descriptive and the probably most threatening question: "*Can you think of a more elegant solution?*"which required the user to find improvements to his own solution and thus criticizing himself. We also left the term elegance without explanation since their interpretation of the term could be of importance.
If they managed to find a more *elegant* solution which might have been smaller or larger improvements in performance, readability, and/or maintainability we figured it was interesting why they did not use that approach or solution in the first place and asked: "*(If yes) Why didn't you attempt its implementation?*".

For the final interview we decided on asking two more descriptive questions about their approach which involved their reflection and comparison to their usual approach outside of the interviews in a *Grand Tour*-Esque manner: "*Is that how you usually approach programming problems?*"and "*Were there any differences?*". Both questions we hoped would further increase our understanding of the participants' solving process, especially what differed during the interviews. Additionally, we asked *What is your opinion on Coding Challenges within the application process?*, an opinion based question and finally we asked them "*Assuming this was a real interview, what do you think your chances for the job would be?*"which again like the very first question could be responded to in an emotional approach or very descriptive. The last two questions were primarily for the participants, to enhance their experience from our interview rather than us gathering useful data from.

After having decided which questions to use, we started acquiring information to prepare the interviewer.

Searching through the plentiful literature on the topic, we focused primarily on sources related to software engineering and qualitative research thereof. Some of our sources [HA05] [Lee02] [Bla13] offer guidelines and explain qualities a good interviewer must have . Based on them, we decided on compiling a list of rules which the interviewer had to follow, a code of conduct to always be present during the interviews. This, so we hoped, would increase the quality of the interviews and thus potentially have a major impact on them. In addition to its persistent presence over the course of the

interviews, we made sure the interviewer would read it at least once before every session. The code of conduct has been translated from German to English and is presented as follows, in no particular order:

The code of conduct:

- Build an atmosphere of trust.
- Ask non-threatening questions.
- Assure anonymity.
- Encourage the interviewees to talk freely.
- Ask relevant and insightful questions.
- Follow up / Explore interesting topics.
- Don't openly disagree with interviewees.
- Don't express dismay.
- Don't interrupt.
- Keep it positive!
- Try to enjoy the interview!
- Thank the interviewees for their participation.

In addition to the code of conduct, we used a primitive reflexive journal where the interviewer had to write a short status report on describing anything he went through emotionally as well as his physical status before the interview. We first came across the idea of a reflexive journal reading "*Behavioural software engineering - guidelines for qualitative studies*"[LFT+17]. Since the interviewer was acquainted with all the participants, we also made sure he put any potentially perceived bias into written form.

Some sources [HA05] suggest using two interviewers instead of one, to increase the quality of the data gathered and to reduce the cognitive load of the interviewers through sharing responsibilities. Another benefit of working with two interviewers is that taking notes becomes possible without interrupting the flow of the interview. To ensure the interviewer can fully concentrate on the questions to ask and the responses thereof, we refrained from taking notes frequently, this was possible due to the fact we recorded the entire study. Additionally, studies about interviews [HA05] have proven that interviews conducted by two interviewers generally have a significant increase in time. We were already stretching the time for the participants before this decision, which was another factor of our decision to remain with one interviewer.

Finally, we also ensured the interviewer would use prompts whenever he felt like the answer he received was too little or too general or upon discovery of something interesting to follow up on [Lee02].

3.6 Data Analysis

To process our data which we generated throughout the study, we split the analysis of the data we gathered in four parts, the solutions provided by the participants, the recordings of the sessions, the personality test and the personal questionnaire. Firstly calculated the performance scores of the individual participants in regards to their results and our criteria which is described in the following subsection 3.6.1. Secondly applied Grounded Theory on the qualitative data from the sessions which is described in Subsection 3.6.2.

3.6.1 Scoring the Coding Challenges

The criteria, to evaluate the solutions with, were correctness, time-efficiency and elegance disregarding space-complexity and were based on our needs and experiences from the closely related research described in section 2.4. We figured elegance is a highly subjective matter and only applied this criteria to the results of the first coding challenge. Our first challenge, the *FizzBuzz* challenge did not offer implementations in differing time-complexity classes but a rather high potential for improvements towards its elegance, more precisely its maintainability, the lines of code, combination of predicates, outsourcing of the predicates. which can be seen in section 3.4.
To evaluate the correctness of *FizzBuzz* we did not see the necessity to create any test cases since many solutions had no input parameters and the output was printed to the console so checking the correctness did not require any additional steps but scanning the printed output. For FizzBuzz we decided a running solution will always yield one point. Additionally bonus points could be earned through improvements

- Combining the two arguments for the FizzBuzz case resulted in 0.25 bonus points.
- Outsourcing the predicates in separate methods and giving them proper names resulted into 0.25 bonus points.
- For a solution with 10 lines of code (loc) or less we would 0.25 bonus points.
- Decreasing the number of places a variable had to be changed at if the requirements changed resulted in 0.25 bonus points.

We also added the possibility of giving bonus points towards improvements we did not consider before the study, in case a participant came up with other well designed improvements. Since our approach towards scoring the FizzBuzz Challenge was open minded and based on improvements, we do not provide a maximum score for FizzBuzz.

For the other two coding challenges, *Sum Swap* and *First Nonrepeated Character*, we decided to stay close to what other expert researchers believe, and that is that objectivity is crucial for any programming competition and technical interview and dismissed including elegance to our scoring scheme. To evaluate those two solutions started with the best run-time efficiency possible and score that solution with 2 points.
As described in Section 3.4, the best conceivable runtime (BCR) for both *Sum Swap* and *First Nonrepeated Character* is within the time-complexity class of $O(n)$ since each element has to be touched at least once [McD19][MKG12]. For Solutions in linear runtime $O(n)$, we would award 2 points per challenge. For solutions in linearithmic time $O(n \log n)$, we would deduct half a point. For solutions in quadratic time $O(n^2)$, we would deduct one point.

Challenge 1		Challenge 2		Challenge 3	
Complexity	Score	Complexity	Score	Complexity	Score
$O(n)$	1.0	$O(n+m)$	2.0	$O(n)$	2.0
Improvements	+ 0.25	$O(n \log n + m \log m)$	1.5	$O(n \log n)$	1.5
		$O(n \cdot m)$	1.0	$O(n^2)$	1.0

Table 3.1: The scoring scheme and the bonus points awarded for the first challenge for each improvement

For the evaluation of *Sum Swap* and *First Nonrepeated Character*, we used a specific set of tests, testing only for edge cases we did not previously exclude. The excluded test cases only affect the evaluation of the *First Nonrepeated Character* challenge and are: empty *Strings*, signs, upper case letters, and letters not used in the English language. Other than in programming competitions and possibly technical interviews, we refrained from using a weighting towards the difficulties of the challenges since they were revealing different qualities of the participants.
All criteria were clearly communicated within the study introduction, and have been repeatedly stated throughout the interview when necessary or when asked for.
As elegance is a special case we prompted each participant after having completed a solution to use the remaining time for improvements and we also explicitly mentioned maintainability, readability, code size, and potential outsourcing.

To create a participant's final score we summed up their scores for the individual challenges.

3.6.2 Analysis procedure of the sessions and interviews

The qualitative data recordings we gathered throughout the course of our study were sessions consisting of an iteration between three observational parts, while the participants solved coding challenges, and three interview parts where the participant offered insight into his thought process as described in Figure 3.1.
In accordance with our research objectives, we chose Grounded Theory as our analysis approach. Grounded Theory (GT) is a well-known methodology within the field of qualitative research to analyze qualitative data typically derived from interviews and observations. GT describes an iterative process of constant comparison to code and conceptualize data in order to find things of interest, a situation, an event, an activity, which are referred to as *phenomena*.
Based on the phenomena, hypothesis are found and a theory can be formed aimed at explaining relationships between those phenomena [SC90] [SPP08] [Cha14]. The Methodology was first described by Barney Glaser and Anselm Strauss in 1967 [GS67] and has since been applied, suggested, and modified in numerous ways [RP10].
As far as we are aware of there are currently four officially recognized approaches towards Grounded Theory and its practical application [SCEB11].
The first two originate from the original authors, Barney Glaser's "*Classic Grounded Theory*"and Anselm Strauss and Juliet Corbin's "*Basics of Qualitative Research*"[SCEB11][Gla92][SC90]. Further there is "*Constructing grounded theory: A practical guide through qualitative analysis*"written by Kathy Charmaz and Adele. E. Clarks "*Situational analysis*"[Cha06] [Cla07].

Upon inspecting our data, we found a transcription of the observational parts impractical. The subtleties of e.g, mouse movement and keystrokes together with their respective timings were nearly impossible to be transcribed accurately.
We thought about more coarsely-grained approaches. However, they all yielded in a loss of potentially important data. Since we did not know what we were looking for in the data, as suggested by Grounding Theory, we did not want to take that risk.
As an alternative approach, we used something similar to what Salinger et al. suggested in "*A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming*"[SPP08]. They had previously encountered a similar problem of transcribing video and audio data from screen-sharing and filming. Since tools exist to code the raw video and audio data, e.g ATLAS.ti[11], Salinger et al. suggest annotating the codes directly onto the video/audio data. They also provide a more practical approach of coding the data within ATLAS.ti, since the tool and the version of GT suggested by Strauss and Corbin in "*Basics of Qualitative Research: Grounded Theory Procedures and Technique*", had trouble translating into each other [SC90][SC97][SPP08].

For the interview stages, we decided on a more coarsely-grained approach of transcribing since there was not much going on on the screens except for some rare occasions where the participants pointed at their code to make a statement, which could be easily referenced in the transcript and traced back to the respective recordings.
Another factor for our decision to transcribe the interviews was that transcripts are easier to work with, traverse through, and code. As of technical limitations, working on the video and audio data directly includes many hiccups and do-overs, and searching for references to connect the dots can be tedious and prone to error.

Further, we decided on looking for different types of phenomena within the interviews, compared to what we were looking for in the observation parts. We did not specify any, until later in the process, to remain open-minded towards the findings, but we found ourselves tending to pay closer attention to the frequency of occurrences of certain phenomena within the observational data while focusing on the detailed thought processes and chosen approaches within the interview transcriptions. Which might be retraced to the process of iterating over both data sets various times, gathering more information each time, concretizing our ideas we developed in the process.

Due to similarities in the nature of our study compared to the study provided by Salinger et al. [SPP08] we decided to benefit from their experiences and build our approach towards GT accordingly. For simplicity reasons we decided on using the same coding schemes for both, the transcriptions and the session recordings and used ATLAS.ti to apply any form of coding.

3.6.3 Analysis of the Personality Test and Personal Questionnaire

As final step of our analysis we proceeded to evaluate the personality test and personal questionnaire. We consciously placed the evaluation of the personality test and personal questionnaire after we had completed the coding process to minimize potential bias towards the coding procedure and ensure a higher level of credibility which we will further discuss in Section 5.2.

[11] https://atlasti.com/

Firstly, we calculated the data from the Big 5 Personality Test using the scoring scheme provided in "*The Big Five inventory (BFI)*"[Joh] which is based on "*The Big-Five trait taxonomy: History, measurement, and theoretical perspectives*"written by Oliver P. John and Sanjay Srivastava [JDK91] to gain the participants values for *Extraversion*, *Agreeableness*, *Conscientiousness*, *Neuroticism* and *Openness*.

Secondly, we extracted the personal information such as names, email adresses, academic performance, experience and SPANE-values provided within the personal questionnaire and calculated the corresponding values for the SPANE-N, SPANE-P and SPANE-B according to the scoring scheme in [Mea].

4 Results

Our sample consisted of 12 participants which were in the final phase of their educational process or had already graduated. All of our participants identify as male. The average age of our participants was 29.17 years with a standard deviation of 2.41.

The study was designed to find differences displayed by more and less conscientious developers in their respective approaches towards solving coding challenges and their impact on the performance. In order to reference a particular participant throughout the analysis while maintaining anonymity we used a simple coding scheme. Each participant received an Identifier based on the scheduling order of the interviews. Information on the scheduling order is only available to third parties if the participant willingly shared this information.

4.0.1 Performance Scores

We calculated the performance scores of each participant according to the criteria described in Section 3.6.1 and shown in Table 3.1, and report them in accordance with their performance in descending order as can be seen in Table 4.1. Although challenges two and three might be considered to be significantly harder to solve, we decided the elegance factor of challenge one was of equal importance to us.
Leaving open to award more bonus points than specified in the preset list towards the *FizzBuzz* challenge meant it had no real maximum. However, we estimated the maximum to be 2.0 which leads to a maximum score for all challenges of 6.0 points. On average the participants had 2.17 correct solutions and a score of 2.92. In Table 4.1 we can see that five out of twelve and thus 41,7% of the participants achieved a correct solution for all three challenges and not a single participant had not at least solved one challenge correctly. Neither of our par

For the ***FizzBuzz*** challenge, only P10 failed to provide a correct solution due to a negotiation error he quickly fixed in the following interview. However, just five participants, P11, P04, P01, P08, and P03 managed to receive any bonus points for their improvements on elegance in spite of the fact we prompted everyone to try and provided hints towards possible directions.
The ***SumSwap*** challenge was correctly implemented by seven of the participants while only participant P11 achieved the Best Conceivable Runtime (BCR). P12 ran out of time implementing his previously created concept, P02 got stuck on the logic of his solution and ran out of time trying to figure out a way back in, P09 ran out of time debugging his *Hashmap* solution which would have achieved the BCR had he completed it, P03 and P10 forgot to handle some edge cases in a brute force approach.
The ***First Nonrepeated Character*** challenge was correctly solved by eight participants, and 6 of which found a solution within the BCR. P08 first implemented a solution in $O(n^2)$ and ran out of time trying to improve it. P09 provided a solution in $O(n^2)$ using *Substrings* and preferred testing his solution extraordinarily over the implementation of improvements to achieve a better runtime. P12

Participant	Challenge 1	Challenge 2	Challenge 3	Score
P11	$O(n) + 0.5$	$O(n+m)$	$O(n)$	5.5
P04	$O(n) + 0.75$	$O(n \cdot m)$	$O(n)$	4.75
P01	$O(n) + 0.25$	$O(n \cdot m)$	$O(n)$	4.25
P05	$O(n)$	$O(n \cdot m)$	$O(n)$	4.0
P08	$O(n) + 0.25$	$O(n \cdot m)$	$O(n^2)$	3.25
P12	$O(n)$	-	$O(n)$	3.0
P07	$O(n)$	$O(n \cdot m)$	-	2.0
P06	$O(n)$	$O(n \cdot m)$	-	2.0
P10	-	-	$O(n)$	2.0
P09	$O(n)$	-	$O(n^2)$	2.0
P03	$O(n) + 0.25$	-	-	1.25
P02	$O(n)$	-	-	1.0

Table 4.1: A description of the individual performance of the participants, the run-time of the solutions, the bonus points for the improvements of the *FizzBuzz* challenge and their final score

was the only participant solving the challenge correctly using an alphabet array, the other successful participants and as such P11, P04, P01, P05, and P10 implemented a version using alternative data structures such as a *Hashmap*. Both data structures, the *Hashmap* and the *alphabet array*, lead to a solution within the BCR $O(n)$. P07 implemented his solution using *regex* expressions and a *Hashmap* in $O(n)$, failing implement some of the edge cases which. P03 failed to debug his alphabet array approach, which, even if he had completed the challenge, would have resulted in a $O(n^2)$ run time. P06 had no success at debugging his rather unconventional concept of sorting the *character string* and iterating twice over it using a *nested for-loop*, which would also have resulted in a $O(n^2)$ runtime, and P02 ran out of time after changing his approach several times ending up with a *Substring* based version which did not quite work yet.

4.0.2 Grounded Theory Application

After having calculated the performance scores, we started with the open coding process according to our description in Section 3.6.2 for the transcribed interview data in a *line-by-line* manner, which resulted into very descriptive codes.
At this point, we had not calculated the conscientiousness values of the participants with the intention of reducing bias towards the findings which we discussed in Section 3.6.3.
After going through one iteration of the transcripts, we proceeded to code the recordings of the observations in a similar fashion. In order to make sure we could properly navigate the recordings we first had to iterate over the set of recordings once, annotating them according to their structure e.g *Challenge 2: Sum swap* and their timestamps. We then proceeded to annotate the recordings but realized a *sentence-by-sentence* or *word-by-word* coding would result in a huge pile of unnavigable data not only due to the sheer mass of annotations but also due to technical limitations.
After careful consideration, we changed our approach towards recording annotations in a more goal-oriented way. We defined a focus. With our research objective in mind, we started to code

anything which we deemed important towards the solving process of the participant, such as ideas, changes in approach, failed attempts, improvements, hints, when clarification was needed, the amount and length of thinking periods, when the participant started the implementation and periods of conceptualization in comparison to other participants.
This resulted in lengthy descriptive codes e.g "*The participant changed the data structure after being given a hint to think about them*"or "*The participant immediately started coding without any conceptualization period*".
In further iterations, we would refine these codes to properly reflect concepts rather than descriptions, as suggested by Strauss and Corbin [SC90]. This would frequently lead to taking apart previously generated codes and properly restructuring them e.g "*The participant immediately started coding without any conceptualization period*"we would refine to "*Immediately Coding*"and "*No Concept*".
We also applied this conceptualization and simplification to the codes within the interviews.
In parallel, we started the axial coding phase. We defined categories in which most of the codes fell. Those categories were then treated in the same fashion as we did the open codes and reduced them further in size of their descriptive content over the course of future iterations.
We ended up with two major coding groups for the interviews which were *Approach* and *Emotions and Character* and four major coding groups for the recordings, namely *Participant-Computer Interaction (PCI)*, *Participant Interviewer Interaction (PII)Thinking Periods (TP)* and lastly, not to become confused by the amount of unassigned structural codes *General Structure*.

Our Approach resulted in over 600 individual codes. Over 500 of those codes came from the interview transcriptions, while we used approximately 120 codes within the recordings. The vast differences in the number of codes between the two media can be explained by the reusability of the codes.
In the interviews, the majority of the generated codes occurred three times or less, e.g "*Kiss Principle*"or "*Sushi Chef Analogy*"which both occurred exactly once, while the codes for the recordings were more general and optimized towards reusability, e.g "*PCI: Debugging*"which had been used 24 times. Our top contender in the amount of times used was *TP* for Thinking Periods with 244 occurrences, Our top contender in the amount of times used was *TP* for Thinking Periods with 244 occurrences, followed by "*PII: Clarification*" 81 times, and "*PII: Hint (asked for)*"*Syntax* 56 times.
We used the clarification code for basically any additional information asked for or given, which lead to a conversation, a back and forth between the participant and interviewer, which wasn't resolved by a single statement. How this information was acquired or presented was derived from another code often used in parallel, such as the previously mentioned *Hint* code.
The hints were separated into two major categories, syntax and logic. A logical hint encompassed data structures, programming logic, and steering lost participants back on track while syntactical hints were used to help the participants struggling with the specific Java syntax.
The participants could ask for a hint at any time and any stage of the challenge solving process, or if they struggled for long enough or tried something that clearly does not work in Java, the interviewer interfered as practiced in technical interviews.

The categorization of every individual code into a code group resulted in 312 codes within the code group *Approach*, 209 within *Emotions and Character*, 18 in *General Structure*, 76 in *Participant-Computer Interaction*, 30 in *Participant-Interviewer Interaction* and 7 in *Thinking Periods*. Generally, we specified the codes to fall only into one code group and reworked the more ambiguous ones

Category	Score
High Conscientiousness	3.35
Intermediate Conscientiousness	2.75
Low Conscientiousness	2.42

Table 4.3: The average score per category

by splitting them into multiple codes with a clear description. However, in some cases, we decided against this procedure because of the loss of qualitative information. Further, throughout the process of annotating the codes, we used memos to describe and solidify our ideas and impressions.

4.0.3 Personality Test

Analyzing the personality test, we found conscientiousness values between 0.33 to 0.72, which are described in Table 4.2 and further in Figure 4.1. These values were comparable to the data provided by Wyrich et al. in "*A theory on individual characteristics of successful coding challenge solvers*" [WGW19], which were in the range of 0.38 to 0.86.

Part.	P01	P02	P03	P04	P05	P06	P07	P08	P09	P10	P11	P12
Cons.	0.61	0.61	0.5	0.5	0.72	0.36	0.64	0.33	0.47	0.58	0.61	0.58

Table 4.2: The participants and their respective conscientiousness values according to the Big 5 Inventory

Since our goal is to determine differences between lower and higher conscientious software developers, which impact the performance of coding challenges, we arranged the participants into different categories according to levels of conscientiousness measured with the Big 5 Inventory Test. To improve the significance of the findings the categories had to be rather distinguishable.
For this purpose, we chose the categorization according to the following criteria. Every participant with a conscientiousness value larger than or equal to 0.6 was put into the higher conscientiousness category. Every participant with a conscientiousness value smaller than 0.5 was put into the lower conscientiousness category, and participants between those values were put into the immediate conscientiousness category.

Upon calculation of the performance scores of the generated groups, which are described in Table 4.3 the data we found did not replicate the negative correlation between conscientiousness and coding challenge performance of the study conducted by Wyrich et al. [WGW19].
In fact, we found the exact opposite correlation.
In our sample, the developers with higher conscientiousness values significantly outperformed those of lower conscientiousness. Our sample consists of only twelve developers and can neither refute nor confirm the results of that study nor do we intend to. Since we applied Grounded Theory towards our interview transcriptions and session recordings before the calculation of the conscientiousness values, this did not affect our generated codes.

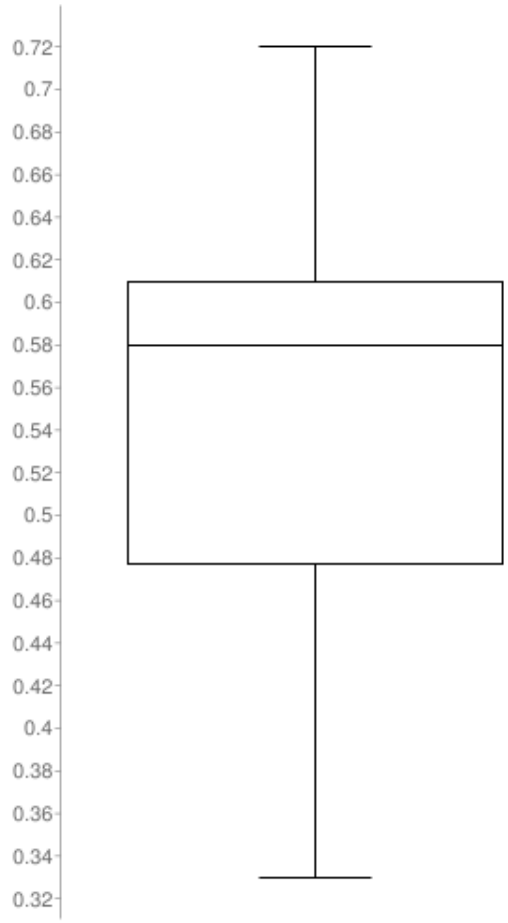


Figure 4.1: Conscientiousness Values of the Participants

4.0.4 Personal Questionnaire

The analysis of the personal questionnaire provided us with plentiful quantitative data to further specify our sample of software developers. As previously stated, The average age of our participants was 29.17 years, with a standard deviation of 2.41.
The average academic performance was 2.22 measured in the German grading scale, while students of a master's degree program had an average academic performance of 2.51 for their bachelor's degree.
The average programming experience was 10.46 years, with a standard deviation of 4.54. For Java, the average programming experience was 6.5 years, with a standard deviation of 3.47.
Out of the 12 participants, only P10 describes his programming experience compared to his student and professional colleagues below average. P3 assesses his programming capabilities compared to his student and professional colleagues as average. The remaining participants specify their programming experience above or exceptionally above those of their student and professional colleagues.
10 out of 12 participants have gathered professional experience working for companies within the realm of software development, only P1 and P2 have not. The average professional experience working for companies is 2.3 years, with a standard deviation of 2.22.

Nine participants do not come across coding challenges in their day to day life. The exceptions are P3, who stated he comes across coding challenges once or twice per semester, P4, who comes across them once or twice per month, and P11 comes in contact with coding challenges in a daily fashion.

From the personal questionnaires, we gathered data about 11 of our 12 participants and their current emotional data, according to the frequency of emotions used in the SPANE-Scale. 10 out of 11 participants provided their current emotional level in accord with the SPANE-Scale. Out of those, only P09 had a slightly negative SPANE-B value, while P02 had a neutral value. P08 and P10 had a slightly positive value, and the remaining participants had positive values at 7 or higher.

4.1 Findings

Applying Grounded Theory we found phenomena in four categories, approaches, thinking periods, conceptualization periods, and voluntary participant-interviewer interaction, which will be discussed in the following section. In Section 4.2, we use our previously generated categories of conscientiousness to attribute the phenomena to the different levels of conscientiousness, and for those phenomena related to conscientiousness, we describe the impact on the performance based on our observations. We define the performance in solving coding challenges as (a) the score, the individual participants achieved for his solutions, and (b) the impression the participants left on the interviewer. For phenomena, we observed for only a few participants we give specific examples. For the more general phenomena, we provide tendencies.

4.1.1 Approaches

The first distinctive approach we found was the use of a concept which we refer to as the **concept approach**. Out of our twelve participants, only two incorporated a true concept into their solving process.
Only P12 used a significant portion of his time, before his implementation, to take notes on a sheet of paper to solve and analyze the coding challenge at hand and only for the *SumSwap* and *First Nonrepeated Character* challenges. He did not complete the *SumSwap* challenge because he ran out of time but solved the *First Nonrepeated Character* successfully and in the BCR.
In the interview, he stated that this is his general approach towards more complex programming tasks and that there is a correlation between the complexity of the task and the scope of his concept. Further, he stated the time pressure forced him to conceptualize a minimalistic solution that barely meets the requirements and to start implementing a solution before he had completed the concept. Under different circumstances, without a strict time limit, he would have completed the concept prior to implementation.
P01 used notes and drawings which he created over the course of his solving process instead of before it for the *SumSwap* challenge, which he completed not optimally but successfully.

Further, we identified a **test driven development approach**. As *test driven development approach*, we defined the approach in which participants write various test cases before they start implementing the actual solution. Only one participant, P04, used this approach and only within the context of the *SumSwap* challenge. In the interview, he admitted he did this because had not had an idea how

to solve the challenge at the time and figured the testing would help him find one, so his primary incentive for the approach was to gain knowledge and guide his implementation. P10 also told us in the interview he thought about developing his solution using test-driven development but considered it to be out of scope.

Analyzing the session recordings, we identified four participants, P04, P08, and P11 used the Java documentation not only more extensively than the other participants but to conceptualize and generate ideas, we called this the **Java documentation research approach**. The other participants used the Java documentation primarily to fix their syntax and to ease the process of implementation by using what the Java libraries offer.

Another approach we detected in our session recordings, many participants were frequently deleting and commenting out parts of their solution. Some eventually decided to start over. We called this the **trial and error approach**. We observed this phenomenon in primarily three ways. (a) The participants who finished their solutions were unhappy about their algorithms, commented out their solution, and started over. (b) The participants became confused or stuck during the process of the implementation and deleted parts of their code. (c) They hastily implemented structural elements, which they reworked either directly after or when figuring out the next steps. Among those three types (c) was the most frequently displayed behavior. The analysis showed in many cases, participants used their previously gained knowledge to implement different, often more optimized, and fitting code structures.

During the analysis of the recordings for the *FizzBuzz* challenge, we discovered that most participants implemented their solutions a the **line-by-line approach**. Incorporating a *line-by-line approach* approach, the participant would iteratively implement their solution based on the description of the challenge.
In our description for the *FizzBuzz* challenge, we used, apart from the name and time limit, exactly four lines to describe the requirements of the challenge. The majority of participants were observed to incorporate this behavior, to implement exactly what was written in the line of the description, without future requirements in mind. Our observations show this worked well for the first three lines. However, trying to implement the fourth, many of our participants had to remodel their code structure to include the most specific case, the *FizzBuzz* statement. In a few cases, the participants appended the logic to the other statements rather than properly rearranging the *for-loop*, resulting in suffering code quality in exchange for the sake of simplicity of the implementation.

Related to the line-by-line approach, we discovered an iterative **step-by-step approach** for the two challenges of moderate difficulty. Using a *step-by-step approach*, the participants were observed to solve one problem after another, only dealing with subproblems at the time they encounter them. Elements of a step-by-step approach could be observed for all of our participants, including P12 and P01, who created concepts. None of our participants created concepts to envelop the entire logic of the program, as they admitted during the interviews, we hypothesize that the step-by-step approach might be the polar opposite of the *concept approach*.

Further, we found traces of what we call an **outside-in approach** and an **inside-out approach** among our participants. The *outside-in approach* we defined as constructing the larger framework before implementing the specifics, while the *inside-out approach*, as the polar opposite, we defined as solving and implementing the specifics before the larger framework.

We observed elements of both approaches being used so interchangeably, and none of our participants showed more than a tendency that we cannot determine if either of those approaches exists independently. Additionally, we observed elements of the *inside-out approach* have been used significantly more often than elements of the *outside-in approach*.

Another observation we made was the approaches taken by the participants to solve our challenges highly depended on the difficulty and complexity of the task. For our *FizzBuzz* challenge, the entirety of all twelve participants mostly used elements of a trial and error and sequential approaches while for the more difficult and complex challenges, *SumSwap* and *First Nonrepeated Character* the approaches varied to a much higher degree.
We also found that some of the approaches, especially the creation of a concept, test-driven development, and using the Java documentation for a research purpose, were deliberate and conscious choices while trial and error and step-by-step approaches were frequently used unconsciously. Additionally, the analysis of our interviews supports that some elements of the *test driven development*, trial and error and sequential approaches can be traced back to a learning-by-doing the behavior.

On a higher abstraction level, we found elements of all previously discussed approaches were used interchangeably.

4.1.2 Thinking Periods

Another phenomenon we found was differences between the thinking periods taken by the participants. During the coding phase, we found the amount of time, the frequency, and the timing of when thinking periods occur differ between the participants. While some participants actively avoided taking their hands off their keyboards, feeling uncomfortable in periods of silence, others would frequently stop coding and think.
As *Thinking Periods* (TPs), we defined situations in which the participant would stop implementing actively, and little to no conversation was hold between the participant and interviewer.
To not drown in detail, we set the minimum time period of a thinking period to three seconds. The amount of TPs varied between 8 and 34 for the entirety of all three coding challenges among our participants. The duration for individual TPs was between three seconds and three minutes. The session duration, calculated by the sum of all TPs of an individual participant for one recording, varied between 2 minutes and 6 seconds to 19 minutes and 32 seconds for entire sessions. We also found differences in the timings, when TPs occur, which have only been created for the conscientiousness groups rather than individuals and thus are discussed in Section 4.2.

4.1.3 Conceptualization Periods

Additionally, we observed the time the participants took to think about the current challenge, prior to their implementation, varied greatly. We called those periods *conceptualization periods* (CPs), which we defined by all thinking periods that happen prior to any form of implementation. A CP does not necessarily result in a concept, notes, diagrams, or any other tools in material form. As per our definition, CPs are purely based on the time taken to think prior to the implementation.
CPs were primarily used to form ideas on how to tackle the challenge.
The *conceptualization periods* varied significantly between participants as well as coding challenges. For *FizzBuzz*, we observed only two participants who had any form of CPs and those were 7 seconds

and 20 seconds long. The *SumSwap* challenge had the most evenly distributed form of CPs. Not a single participant started the implementation without at least a small period of conceptualization. The average time spend prior to the implementation was 92 seconds, the minimum was 16 seconds and the maximum was 8 minutes and 59 seconds. For the *First Nonrepeated Character* challenge, we observed the highest degree of variations between the participants. Three participants immediately started coding, while the others took 11 seconds to nearly 4 minutes to conceptualize their implementation.

4.1.4 Voluntary Participant-Interviewer Interaction

Analog to the thinking periods, we found differences in the amount and duration of the voluntary Participant-Interviewer Interaction (vPII). While some participants actively engaged in discussions and sought help from the interviewer, others refrained from talking, coded in silence, and even if they had gotten stuck, waiting for the interviewer to initiate the conversation.
We calculated the average frequency of the vPIIs per participant and we found significant differences between participants. The lowest amount of vPIIs we discovered was 0,67 for a participant per session while the largest amount we found was 11,67 vPIIs. The average of all participants was 5,42.
The duration of vPIIs varied considerably. We have observed vPIIs lasting from less than five seconds to four minutes. Most participants kept the vPIIs short and precise, for example asking how much time they had left, or how to write the specific syntax of a Hashmap in Java. However, for eight of our participants, we observed lengthy discussions on various topics and for different challenges, though the tendency for such an occurrence was more frequently observed in the more difficult tasks. Participant P6 was, in this regard, exceptionally striking, we discovered large vPIIs across all three challenges, which we did for no other participant.

4.2 Attribution to Conscientiousness and Performance

In the next step, we compare the previously created groups of conscientiousness with the phenomena we found using Grounded Theory. The goal is to find out whether or not these differences can be attributed to different levels of conscientiousness. Further, we analyze the performance impact of found phenomena that are related to conscientiousness.

Elements of the **concept approach** were only used by two participants, P1, and more pronounced by P12. P1 used a quickly created paper drawing for the *SumSwap* challenge, and P10 spent an extended period of time for a concept he had written using pen and paper for both the *SumSwap* challenge and the *First Nonrepeated Character* challenges. P1 is part of the highly conscientious group, while P10 is part of the group of intermediate conscientiousness. For our sample, only participants of lower conscientiousness had no representation for using a concept. This leads to the assumption that software developers of intermediate and higher conscientiousness are more likely to work using a concept.
Further, we observed that concept creation can lead to various outcomes. P01 used a concept and achieved a non-optimal solution for the *SumSwap* challenge. P12 failed to complete the *SumSwap* challenge due to running out of time but solved the *First Nonrepeated Character* challenge optimally. This has a few consequences on coding challenges in technical interviews as we conducted them.

The creation of a concept was, for the most part, done in silence, which can lead to a negative impression of the interviewee. However, the creation and use of a concept left a very positive impression of the interviewee. During periods of silent thinking, especially without any presented code, the interviewer can not provide any hints unless he interrupts the interviewee asking about the process. Additionally, the time for solving coding challenges is limited, and the creation of a concept takes time, ergo, creating concepts that increase the risk of running out of time and hence not completing the challenge.

Only one person incorporated a true **test driven development approach**. This participant was from the immediate conscientiousness group. This lead to no assumption towards conscientiousness. Additional data is needed to answer if there is a correlation between conscientiousness and a test-driven development approach. However, we hypothesize that experience factors into this phenomenon as P04 and P11 were the two most experienced participants in our study, and P04 incorporated the approach while P11 stated he thought about doing so but decided against it in the interviews. No other participants did or said anything related to test-driven development.
The creation of tests increases the risk of running out of time. However, since this a very visual process, we believe this can leave a very positive impression of the interviewee depending on the interviewers' values, the companies values, or both.

The **Java documentation research approach** was applied by three participants, P04, P08, and P11. Since each of those participants is in different conscientiousness categories, but all three of them gathered major experience working for companies, we do not see a relationship between conscientiousness and incorporating the Java documentation research approach. However, we hypothesize this is related to experience.

Elements of the **trial and error approach** have not been used during the *FizzBuzz* challenge by any participant. Over the course of the *SumSwap* challenge, four participants from the higher conscientious group, two of the lower conscientious group, and one participant from the intermediate conscientious group incorporated elements of the trial and error approach. During the final challenge, *First Nonrepeated Character*, four participants of the high conscientiousness group, two of the lower conscientiousness group, and two of the intermediate conscientiousness group used elements of the trial and error approach. Upon discovery of the trial and error approach, we believed participants with lower conscientiousness values would more frequently use elements of this approach. However, the analysis showed this is not the case for our sample. In fact, we found that the higher conscientiousness group has a higher tendency towards using elements of a trial and error approach.
Our observation showed the trial and error approach has two effects on performance in technical interviews. The participants using this approach receive more hints than the ones who do not, which increases the chance of completion but incorporating elements of trial and error also leads to a negative impression of the interviewee.

The **line-by-line approach** was used by participants of all three groups. However, the extent to which varied slightly. We found that two out of four participants of intermediate used elements of the line-by-line approach during the *FizzBuzz* challenge. For the developers high in conscientiousness, this was the case for three out of five participants, and only one of the three low conscientiousness participants applied this approach for the *FizzBuzz* challenge. This leads us to believe software developers of higher conscientiousness are more inclined to code solving their challenges in a line-by-line fashion. Using this approach resulted in proper solutions for most participants. However, we have seen this approach increases the risk of bad code style. We did not observe any positive or negative impressions towards the interviewee.

We observed participants using elements of the **step-by-step approach** for all three challenges to varying degrees. The only participant for whom we could clearly identify a distinct change in approach was P12. This change happened during the *SumSwap* challenge, and in the interviews, he admitted his concept was minimalistic and incomplete. This leads to the assumption that a step-by-step approach is the opposite of the concept approach. In our sample, we found no correlation between different levels of conscientiousness and using elements of a step-by-step approach. However, under the assumption that creating a concept is indeed the opposite of the step-by-step approach and the fact we found no concept creation among participants of lower conscientiousness, we believe that participants of lower conscientiousness have a higher tendency towards using a step-by-step approach. Additionally, since we observed every participant using elements of a step-by-step approach we think this is the most regularly used approach for participants in a technical interview.
Using a step-by-step approach left neither negative nor positive impressions of the interviewees. Solving problems as they occur was the common approach and resulted in different kinds of solutions for all challenges. We do not see any relation to the performance directly. However, we observed this approach to be very visible resulting in more hints given by the interviewer compared to the concept approach.

Since our tasks required no larger frameworks and the implementations were straight forward, we found no conclusive support in our data that connects the level of conscientiousness between the **inside-out** and the **outside-in approach**.

The amount of **thinking periods (TPs)** per participant averaged overall challenges for the three different groups were only marginally different. The group higher in conscientiousness showed the lowest amount with 20,6, the intermediate conscientiousness group 23, and the lower conscientiousness group 21,34. As the amount of TPs was evenly distributed among all our participants, we do not believe conscientiousness is a driving factor for the amount of TPs.

The time spent on **thinking periods** was the lowest among the lower conscientious participants, with an average of 517 seconds, followed by the higher conscientious participants with 611 seconds. The group of intermediate conscientiousness showed the largest amount, with 727,5 seconds on average. These numbers give us reason to believe there might be a correlation between the level of conscientiousness and the time taken to think in silence. To further support this, we found that experience gathered in working environments drastically reduces the amount and duration of TPs. Since P04, of the intermediate conscientiousness group and P11 of the higher conscientiousness group, have significantly more experience compared to the other participants, we argue that the differences between the groups should be even more pronounced.

To make observations about the placement of the **thinking periods**, we defined three categories, early TPs, intermediate TPs, and late TPs. The early TPs represented the conceptualization and early implementation stages. The intermediate TPs represented the primary implementation process, and the late questions represented the finalizing stages, which we frequently observed to be testing, debugging, and optimizing.

We found that more conscientious developers had more **early TPs** compared to the other two groups. On average, a more conscientious developer showed 7 early TPs, compared to 6 TPs for the two remaining groups. Although this effect is not very prevalent, we argue it indicates conscientious developers tend more TPs at the early stages since the difference in experience among our groups has decreased this effect.

For **intermediate TPs**, the highly conscientious developers had fewer (7,4) compared to the control group (10) and the lower conscientious developers (11). This leads us to the assumption, that developers higher in conscientiousness are less likely to run into structural or logical issues within the implementation since they, on average, spend more time on the conceptual work prior to the implementation.

The intermediate conscientiousness group displayed the highest number of **late TPs** with an average of 6, while the highly conscientious group asked 5,6 questions and members of the lower conscientious group 4,34. This result leads us to the assumption that less conscientious developers spend less time thinking about possible optimizations, testing, and debugging.

TPs are a crucial process in the solving of coding challenges. We found no indication in our data that thinking periods are directly related to the performance in coding challenges. However, we did observe participants with a lower duration within their TPs make more logical and syntactical mistakes and more typos. Those mistakes can have a negative impact on the perception of the participant, but we observed this effect to be faint. Additionally, long periods of silence are uncomfortable for both the interviewee and the interviewer.

For the **conceptualization periods (CPs)**, we found the highest time spent before the implementation among the intermediate conscientiousness group with 208 seconds on average. Participants of the more conscientious group spent on average 134,6 seconds before implementing and the lower conscientious group, on average 36,67. Since this is a striking effect, we conclude less conscientious developers take less time to conceptualize.

For our sample, we have observed no correlation between the conceptualization periods and performance within the coding challenges. P04 and P09 have both spend less than 20 seconds conceptualizing for all three of our coding challenges, and P09 achieved a score of 1.25 with his solutions while P04 achieved the second-best score of 4.75. On the other end, we have seen P10 spending four minutes of his time conceptualizing during the *First Nonrepeated Integer* challenge achieving an optimal solution and P12 spending nearly nine minutes on conceptualizing periods during the *SumSwap* challenge without achieving a working solution. However, we do have observed that lengthy conceptualization periods as well as no conceptualization periods have a negative impact on the impression the interviewer has about the interviewees.

The amount of **voluntary participant-interviewer interaction (vPIIs)** differed slightly between the groups. The lower conscientious developers showed, on average, with 4,78, the lowest amount followed by the highly conscientious developers with 5,13 and the intermediate conscientiousness group with 6,25.
We see no correlation between the vPIIs and conscientiousness and argue this is based on agreeableness since we found P06, the only participant displaying large vPIIs across all three coding challenges also has the highest agreeableness score in our study with 0.78.

Analyzing the **code quality**, we found participants of higher conscientiousness received an average of 0.15 bonus points, those of intermediate conscientiousness received an average of 0.2 and those of lower conscientiousness 0.08. Additionally, we observed participants of intermediate and higher conscientiousness talk about code quality more frequently and more in depth. This leads to the assumption that less conscientious developers provide solutions worse in code quality.

5 Discussion

5.1 Implications

Over the course of our data analysis, we found several phenomena that are related to the level of conscientiousness and to the performance in solving coding challenges. Based on our findings, we present the following hypotheses.

5.1.1 Hypothesis: Developers lower in conscientiousness are less likely to create a concept

We found no participants of lower conscientiousness created a concept or talked about their conceptualization in their interviews. Creating a concept takes time and is, as we observed, frequently done with minimal communication between the candidate and interviewer. This has some implications for the performance in coding challenges as well as other factors of technical interviews. Although we have not observed this behavior to have a direct impact on the scoring of their solutions, we have found a negative impact on the impression they leave on the interviewer. Additionally, we found no differences between developers of intermediate and higher conscientiousness.

5.1.2 Hypothesis: There is no correlation between conscientiousness and the use of a trial and error approach

Upon discovery of trial and error approaches, we assumed this might be an indicator for the participants being lower in conscientiousness. However, further analysis showed this is not the case. In fact, we found a slightly higher amount of more conscientious developers used elements of trial and error. We believe this effect is likely to be attributed to the small sample size and has no correlation with conscientiousness. Further, we have observed incorporating elements of trial and error approaches can increase the odds of successfully solving coding challenge in technical interviews. We reason this is because the interviewer is more inclined to provide hints to what he sees rather than having to ask what the problem is and elements of trial and error approaches are very visual. However, we also observed elements of trial and error approaches to have a negative effect on the impression the interviewer has of the participant.

5.1.3 Hypothesis: Developers lower in conscientiousness spend less time in pause for thought

Our data analysis provided us reasons to believe that developers of lower conscientiousness spend less time thinking in silence. Since thinking in silence is not giving the interviewer insight into the thought process of the candidate, the interviewer might opt to interrupt this process asking to be more involved or think the candidate might have gotten stuck and offer help. Developers spending less time thinking in silence and more in discussions reduce this risk of interruption according to our observation.

5.1.4 Hypothesis: Developers of lower conscientiousness take less time to conceptualize

We have observed notable differences between the time participants take to conceptualize. The conceptualization periods are, as defined in Section 4.1.3, periods time taken to think in silence prior to any form of implementation. We found that the group of less conscientious developers spend notably less time on conceptualization periods compared to the other two groups. We assume this might result in a more visible solving process which the interviewer is more involved in and thus provides more hints, which in turn increases the odds of successfully solving the coding challenges. Additionally, we did not observe major differences between developers of higher and intermediate conscientiousness.

5.1.5 Hypothesis: Developers of lower conscientiousness provide worse code quality

Developers of lower conscientiousness received fewer bonus points on elegance for their *FizzBuzz* solutions and talked about code quality less frequently during the interviews. We detected only minor differences between intermediate and higher conscientious participants. We have observed better code quality to lead to a better scoring and a better impression the participants leave on the interviewer.

5.2 Limitations

As the study is of exploratory and qualitative nature our results do not represent the general population. The goal of this study is to provide qualitative insights into differences less and more conscientious persons display while solving coding challenges that can impact the performance therein. As such, this study has some limitations in regards to internal and external validity to be discussed in the following sections.

5.2.1 Data collection: Semi-Structured Interviews

Although semi-structured interviews are valued within the qualitative research to provide rich data, they also pose a risk towards internal validity. This happens primarily in two ways. The researcher is responsible to create the atmosphere of trust required for the interviewee to open up and provide rich qualitative data. Without such an atmosphere, the interviewee may derive from the path of being truthful and tell the researcher what he wants to hear instead, which is commonly referred to as social desirability bias [Ned85]. To ensure the researcher was capable of providing the atmosphere required, we specified a code of conduct, the rules of behavior in advance, and forced the researcher to refresh his memory before every interview. Additionally, the code of conduct was placed in front of the interviewer during the sessions to provide reassurance in cases of doubt or insecurity. Further, the researcher is a primary driving force within the interview. He can control the course of the semi-structured interview by steering the participants into desired directions. This is usually perceived as an advantage semi-structured interviews have over structured interviews since additional qualitative data can be gathered upon discovery. However, it is also susceptible to bias. The decisions made by the researcher in this regard are defined by his area of interest and have a direct influence on the quality of data gathered. In our specific case, this could have lead to the interviewer to take different paths with interviewees he perceived as conscientious, and those whom he thought were not. We used two steps to overcome this threat. We designed the predefined questions in such a way that all major categories were covered, and we used a reflexivity journal, where the researcher had to describe his perception of the interviewee.

5.2.2 Data analysis: Transcriptions and Coding

While transcribing the interviews, the researcher has to define the granularity of the contents to separate the data put into the transcription. This can lead to a loss of potentially important information due to subjective decisions made by the researcher. To circumvent this issue, we used an accurate word-by-word approach, only transcribing what is explicitly said, including only the thinking pauses and references to code, with no interpretation of the meaning and intention. To compensate for this loss of information, we used a two-step approach of not only analyzing the transcripts of the interviews but also annotating the session recordings themselves, specifically looking for anything we did not capture within the interviews.

The coding procedure applying Grounded Theory is, to a degree, subjective in nature. Purely descriptive codes cannot lead to the discovery of new hypotheses and theories. However, too much interpretation can be a threat to reliability and credibility. To find a decent balance between the two we used a concoction between *objective-descriptive* and *subjective-evaluative* codes as suggested by Salinger et al. [SPP08].

5.2.3 Relationship towards Technical Interviews

Technical Interviews conducted within the industry can differ from the design of this study in various ways. Within the industry, two different types of technical interviews are commonly used, remote interviews and on-site interviews. For on-site interviews most companies still use either a whiteboard or a pen and paper approach. Code written in either of those media does not benefit from any advanced features an IDE might offer and the sole source of additional information is the

interviewer him- or herself. Remotely conducted Interviews also vary to a degree. Some companies let candidates code in plain text editors to mirror on-site interviews, others use properly set up IDEs to represent professional programming instead. The difficulties of coding challenges in technical interviews also varies depending on the circumstances. Some companies use coding challenges of higher difficulties for a better assessment of their cognitive abilities and filtering out candidates they deem inappropriate. Since we intended to find differences in the approaches of our participants and use multiple coding challenges, we refrained from taking such actions.

Additionally, the interviewer is usually a trained and experienced professional, we are neither in this regard, however, to minimize this effect we extensively researched this topic according to current literature and used structured guides to support the interviewer to create an environment more closely related to professionally conducted interviews.

5.2.4 Guidance for the Participants

To simulate real technical interviews, we decided on providing extended guidance for our participants. As previously admitted, we are neither trained nor experienced in this regard, so we can not guarantee that the extent to which we, in the role of the interviewer, provided help was equal towards all participants. In particular, we could have provided too much additional information to struggling participants, which might have resulted in a closer gap between better and worse participants in regards to performance. Further providing hints on certain subjects such as data structures might have influenced the participants towards attempting a solution they would not have discovered on their own

5.2.5 Lack of Preparation

The lack of preparation was a major concern for us. Unlike Candidates of professionally conducted technical interviews who are hoping to receive a job offer, our participants had nothing to gain from the interview except for the experience itself. There was no incentive to do well, which had a clear impact on their preparation. At best, as we thought, the participants would do some minor research towards the syntax of Java, which they might not have programmed actively in for years. If a participant struggled with the java syntax and refrained from asking, we would provide the hints anyway, to even the playing field. Our intention behind this was to receive information on their thought- and solving processes rather than to test their knowledge. To capture the effect of preparation by our participants, we asked them if and how often they come in contact with coding challenges.

5.2.6 The inclusion of Code Quality

In an attempt to stay true to the nature of the study, which has been designed as a simulation of a technical interview with a few minor tweaks, the quality criteria have been included for one of the challenges. This meant we had to determine possible improvements towards simplicity, readability, maintainability, compactness, and other quality criteria. Since there was no certainty we could capture all possible improvements, we left our scoring scheme for the first challenge subject to change. Besides, the participants were not informed about the exclusion of the quality criteria for

the two more challenging tasks. We did not want to risk the participants taking an easier approach and spend less time on improvements since they would have known it had no impact on their scoring. Further, if participants finished their tasks early, we would prompt them if they wanted to continue improving their solutions towards elegance.

5.2.7 Conscientiousness

Although we found comparable values for conscientiousness in the data provided by Wyrich et al. [WGW19], we are aware our values are from a small sample and do not account for the general software development population. In fact, we have seen different ranges for conscientiousness in larger samples such as recent research on social media-predicted personality traits and values by Kern et al. [KMCR19] who found much lower values for conscientiousness in the range of 0.05 to 0.61 for three different categories of software developers which amount to over 600.

6 Conclusion

A recent study found a negative correlation between conscientiousness and the ability to solve coding challenges well and within the given time frame [WGW19]. Conflictingly conscientiousness is widely acknowledged to be in a positive correlation with job performance, which in the industry, should be highly sought after. Since coding challenges have become a popular tool to assess the problem-solving ability of applicants, this has serious implications.

To investigate this correlation, we conducted an exploratory study to find differences between more and less conscientious developers and what impact those differences have on solving coding challenges, especially in the context of technical interviews. We found differences in two categories, those who affect the achieved score of the candidates directly and those which affect the interviewers' impression. Our findings indicate that software developers of intermediate and high conscientiousness are more likely to create concepts, think in silence for longer periods of time, start implementing later than less conscientious software developers but provide better code quality. In a regular work environment, these differences have a positive influence on development. In some scenarios of technical interviews they pose a threat. We observed the creation of a concept can take up so much time the candidate is not able to finish his solution. Longer periods of silence can not only be uncomfortable for both, the interviewer and the candidate but give the interviewer no insight into the thought processes of the candidate, which can lead to a worse evaluation.
For any future candidates we suggest, take your time to think but involve the interviewer in every decision you make and every step you take. Consciously choose your approach, the more visual, the better, and discuss your thoughts with the interviewer so he or she can assist in finding a solution. Finally, be talkative and share specific knowledge, it will lead to a better impression.

6.0.1 Future work

We identified several components in our study, which benefit from future research. In this study, we simulated an off-site technical interview, incorporating a properly set up IDE with access to the Java documentation, using screen-sharing software. We did not use a web camera to record potentially important facial expressions, and the participants were not allowed to do any internet research.
Altering any of those components can lead to new insights, more differences between more and less conscientious software developers can potentially be found, and our findings can be evaluated for other forms of technical interviews.
Since our participants were unprepared, similar studies can repeat this process with prepared participants to gain additional insights and compare the differences. Another topic of interest we can think of is studying different levels of guidance, and the degree of impact guidance has on the solutions and the behavior candidates display. Further future studies can statistically confirm or refute our hypothesis.

Bibliography

[AGJ09] S. T. Acuña, M. Gómez, N. Juristo. "How do personality, team processes and task characteristics relate to job satisfaction and software quality?" In: *Information and Software Technology* 51.3 (2009), pp. 627–639 (cit. on p. 23).

[AI10] B. Alexander, C. Izu. "Engaging weak programmers in problem solving". In: *IEEE EDUCON 2010 Conference*. IEEE. 2010, pp. 997–1005 (cit. on p. 25).

[ALP12] A. Aziz, T.-H. Lee, A. Prakash. *Elements of Programming Interviews: The Insiders' Guide*. EPI, 2012 (cit. on p. 24).

[Are06] A. S. Arefin. *Art of programming contest: C programming tutorials, data structures, algorithms*. 2006 (cit. on p. 24).

[Ast04] O. L. Astrachan. "Non-competitive programming contest problems as the basis for just-in-time teaching". In: *34th Annual Frontiers in Education, 2004. FIE 2004.* IEEE. 2004, T3H–20 (cit. on pp. 25, 26).

[BH08] B. A. Burton, M. Hiron. "Creating informatics olympiad tasks: exploring the black art". In: *Olympiads in Informatics* 2 (2008), pp. 16–36 (cit. on p. 24).

[Bla13] A. Blandford. "Semi-structured qualitative studies". In: Interaction Design Foundation, 2013 (cit. on pp. 48, 50).

[BLM+18] M. Behroozi, A. Lui, I. Moore, D. Ford, C. Parnin. "Dazed: measuring the cognitive load of solving technical interview problems at the whiteboard". In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. 2018, pp. 93–96 (cit. on p. 19).

[BM91] M. R. Barrick, M. K. Mount. "The big five personality dimensions and job performance: a meta-analysis". In: *Personnel psychology* 44.1 (1991), pp. 1–26 (cit. on pp. 21, 22, 48).

[BMS93] M. R. Barrick, M. K. Mount, J. P. Strauss. "Conscientiousness and performance of sales representatives: Test of the mediating effects of goal setting." In: *Journal of applied psychology* 78.5 (1993), p. 715 (cit. on p. 23).

[BPB19] M. Behroozi, C. Parnin, T. Barik. "Hiring is broken: What do developers say about technical interviews?" In: *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2019, pp. 1–9 (cit. on pp. 13, 18).

[BSBP20] M. Behroozi, S. Shirolkar, T. Barik, C. Parnin. "Debugging Hiring: What Went Right and What Went Wrong in the Technical Interview Process". In: *ACM/IEEE International Conference on Software Engineering (ICSE), SEIS Track*. 2020 (cit. on pp. 13, 18).

[BWPO91] W. C. Borman, L. A. White, E. D. Pulakos, S. H. Oppler. "Models of supervisory job performance ratings." In: *Journal of Applied Psychology* 76.6 (1991), p. 863 (cit. on p. 23).

[CA10a] L. F. Capretz, F. Ahmed. “Making sense of software development and personality types”. In: *IT professional* 12.1 (2010), pp. 6–13 (cit. on p. 21).

[CA10b] L. F. Capretz, F. Ahmed. “Why do we need personality diversity in software engineering?” In: *ACM SIGSOFT Software Engineering Notes* 35.2 (2010), pp. 1–11 (cit. on p. 21).

[CC08] D. Cohen, B. Crabtree. *Semi-structured interviews. Robert Wood Johnson Foundation Qualitative Research Guidelines Project. URL*. 2008 (cit. on p. 48).

[CH17] N. Cheng, B. Harrington. “The Code Mangler: Evaluating Coding Ability Without Writing any Code”. In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 2017, pp. 123–128 (cit. on p. 26).

[Cha06] K. Charmaz. *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006 (cit. on p. 53).

[Cha14] K. Charmaz. *Constructing grounded theory*. sage, 2014 (cit. on p. 53).

[CHMG00] A. Carbone, J. Hurst, I. Mitchell, D. Gunstone. “Principles for designing programming exercises to minimise poor learning behaviours in students”. In: *Proceedings of the Australasian conference on Computing education*. 2000, pp. 26–33 (cit. on p. 25).

[Chr] J. J. J. Christopher J. Soto. *Five-Factor Model of Personality*. URL: https://www.oxfordbibliographies.com/view/document/obo-9780199828340/obo-9780199828340-0120.xml?print (cit. on pp. 21, 48).

[CJW11] D. Coles, C. Jones, E. Wynters. “Programming contests for assessing problem-solving ability”. In: *Journal of Computing Sciences in Colleges* 26.3 (2011), pp. 28–35 (cit. on p. 25).

[CKLO03] B. Cheang, A. Kurnia, A. Lim, W.-C. Oon. “On automated grading of programming assignments in an academic institution”. In: *Computers Education* 41.2 (2003), pp. 121–131. ISSN: 0360-1315. DOI: https://doi.org/10.1016/S0360-1315(03)00030-7. URL: http://www.sciencedirect.com/science/article/pii/S0360131503000307 (cit. on p. 26).

[Cla07] A. E. Clarke. “Situational analysis”. In: *The Blackwell Encyclopedia of Sociology* (2007), pp. 1–2 (cit. on p. 53).

[CM08] P. T. Costa Jr, R. R. McCrae. *The Revised NEO Personality Inventory (NEO-PI-R)*. Sage Publications, Inc, 2008 (cit. on pp. 21, 22).

[CM92] P. T. Costa, R. R. McCrae. *Neo personality inventory-revised (NEO PI-R)*. Psychological Assessment Resources Odessa, FL, 1992 (cit. on p. 22).

[CM98] P. T. Costa Jr, R. R. McCrae. “Six approaches to the explication of facet-level traits: examples from conscientiousness”. In: *European Journal of Personality* 12.2 (1998), pp. 117–134 (cit. on pp. 22, 23).

[CS12] D. A. Cobb-Clark, S. Schurer. “The stability of big-five personality traits”. In: *Economics Letters* 115.1 (2012), pp. 11–15 (cit. on p. 21).

[CSC15] S. Cruz, F. Q. [Silva], L. F. Capretz. “Forty years of research on personality in software engineering: A mapping study”. In: *Computers in Human Behavior* 46 (2015), pp. 94–113. ISSN: 0747-5632. DOI: https://doi.org/10.1016/j.chb.2014.12.008. URL: http://www.sciencedirect.com/science/article/pii/S0747563214007237 (cit. on p. 20).

[Dag10] V. Dagienė. “Sustaining informatics education by contests”. In: *International Conference on Informatics in Secondary Schools-Evolution and Perspectives*. Springer. 2010, pp. 1–12 (cit. on p. 25).

[Diea] E. Diener. *Scale of Positive and Negative Experience (SPANE)*. URL: http://labs.psychology.illinois.edu/~ediener/SPANE.html# (cit. on pp. 22, 47).

[Dieb] E. Diener. *Scale of Positive and Negative Experience (SPANE)*. URL: https://eddiener.com/scales/8 (cit. on pp. 22, 47).

[Dig90] J. M. Digman. “Personality structure: Emergence of the five-factor model”. In: *Annual review of psychology* 41.1 (1990), pp. 417–440 (cit. on p. 22).

[DS04] V. Dagiene, J. Skupiene. “Learning by competitions: olympiads in informatics as a tool for training high-grade skills in programming”. In: *ITRE 2004. 2nd International Conference Information Technology: Research and Education*. IEEE. 2004, pp. 79–83 (cit. on p. 25).

[DWB+09] E. Diener, D. Wirtz, R. Biswas-Diener, W. Tov, C. Kim-Prieto, D.-w. Choi, S. Oishi. “New measures of well-being”. In: *Assessing well-being*. Springer, 2009, pp. 247–266 (cit. on p. 22).

[FBRP17] D. Ford, T. Barik, L. Rand-Pickett, C. Parnin. “The tech-talk balance: what technical interviewers expect from technical candidates”. In: *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE. 2017, pp. 43–48 (cit. on p. 19).

[For10] M. Forišek. “The difficulty of programming contests increases”. In: *International Conference on Informatics in Secondary Schools-Evolution and Perspectives*. Springer. 2010, pp. 72–85 (cit. on p. 24).

[Fur96] A. Furnham. “The big five versus the big four: the relationship between the Myers-Briggs Type Indicator (MBTI) and NEO-PI five factor model of personality”. In: *Personality and Individual Differences* 21.2 (1996), pp. 303–307. ISSN: 0191-8869. DOI: https://doi.org/10.1016/0191-8869(96)00033-5. URL: http://www.sciencedirect.com/science/article/pii/0191886996000335 (cit. on pp. 20–22).

[GF09] G. Gárcia-Mateos, J. L. Fernández-Alemán. “A course on algorithms and data structures using on-line judging”. In: *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*. 2009, pp. 45–49 (cit. on p. 25).

[GFWA18] D. Graziotin, F. Fagerholm, X. Wang, P. Abrahamsson. “What happens when software developers are (un) happy”. In: *Journal of Systems and Software* 140 (2018), pp. 32–47 (cit. on p. 47).

[Gla92] B. G. Glaser. *Basics of grounded theory analysis: Emergence vs forcing*. Sociology press, 1992 (cit. on p. 53).

[Gol81] L. R. Goldberg. “Language and individual differences: The search for universals in personality lexicons”. In: *Review of personality and social psychology* 2.1 (1981), pp. 141–165 (cit. on p. 21).

[GS67] G. Glaser Barney, L. Strauss Anselm. “The discovery of grounded theory: strategies for qualitative research”. In: *New York, Adline de Gruyter* (1967) (cit. on p. 53).

[GWA13] D. Graziotin, X. Wang, P. Abrahamsson. “Are happy developers more productive?” In: *International Conference on Product Focused Software Process Improvement*. Springer. 2013, pp. 50–64 (cit. on p. 47).

[GWA14] D. Graziotin, X. Wang, P. Abrahamsson. “Happy software developers solve problems better: psychological measurements in empirical software engineering”. In: *PeerJ* 2 (2014), e289 (cit. on pp. 22, 47).

[HA05] S. E. Hove, B. Anda. “Experiences from conducting semi-structured interviews in empirical software engineering research”. In: *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE. 2005, 10–pp (cit. on pp. 49–51).

[HGS12] E. Hahn, J. Gottschling, F. M. Spinath. “Short measurements of personality–Validity and reliability of the GSOEP Big Five Inventory (BFI-S)”. In: *Journal of Research in Personality* 46.3 (2012), pp. 355–359 (cit. on p. 21).

[HIKM12] J. Helminen, P. Ihantola, V. Karavirta, L. Malmi. “How do students solve parsons programming problems? an analysis of interaction traces”. In: *Proceedings of the ninth annual international conference on International computing education research*. 2012, pp. 119–126 (cit. on p. 27).

[Hun80] J. E. Hunter. *Validity Generalization for 12.000 Jobs: An Application of Synthetic Validity and Validity Generalizatoin to the General Aptitude Test Battery (GATB)*. US Department of Labor, Employment Service, 1980 (cit. on p. 22).

[Hun86] J. E. Hunter. “Cognitive ability, cognitive aptitudes, job knowledge, and job performance”. In: *Journal of vocational behavior* 29.3 (1986), pp. 340–362 (cit. on p. 22).

[JDK91] O. P. John, E. M. Donahue, R. L. Kentle. “Big five inventory”. In: *Journal of Personality and Social Psychology* (1991) (cit. on pp. 21, 22, 55).

[Joh] S. S. John O P. *BIG FIVE INVENTORY (BFI)*. URL: https://fetzer.org/sites/default/files/images/stories/pdf/selfmeasures/Personality-BigFiveInventory.pdf (visited on 09/28/2020) (cit. on p. 55).

[JWB+10] J. J. Jackson, D. Wood, T. Bogg, K. E. Walton, P. D. Harms, B. W. Roberts. “What do conscientious people do? Development and validation of the Behavioral Indicators of Conscientiousness (BIC)”. In: *Journal of Research in Personality* 44.4 (2010), pp. 501–511 (cit. on p. 23).

[KMCR19] M. L. Kern, P. X. McCarthy, D. Chakrabarty, M.-A. Rizoiu. “Social media-predicted personality traits and values can help match people to their ideal jobs”. In: *Proceedings of the National Academy of Sciences* 116.52 (2019), pp. 26459–26464 (cit. on p. 73).

[KPJK16] H. Kallio, A.-M. Pietilä, M. Johnson, M. Kangasniemi. “Systematic methodological review: developing a framework for a qualitative semi-structured interview guide”. In: *Journal of advanced nursing* 72.12 (2016), pp. 2954–2965 (cit. on p. 48).

[KVC06] G. Kemkes, T. Vasiga, G. Cormack. “Objective scoring for computing competition tasks”. In: *International Conference on Informatics in Secondary Schools-Evolution and Perspectives*. Springer. 2006, pp. 230–241 (cit. on pp. 26, 27).

[Lee02] B. L. Leech. “Asking questions: Techniques for semistructured interviews”. In: *PS: Political science and politics* 35.4 (2002), pp. 665–668 (cit. on pp. 49–51).

[Leh] M. Lehr. *Working better with conscientious people as problem solvers*. URL: https://omegazadvisors.com/2018/07/02/conscientious-people-as-problem-solvers/ (visited on 09/28/2020) (cit. on p. 23).

[Lev05] A. Levitin. "Analyze That: Puzzles and Analysis of Algorithms". In: *SIGCSE Bull.* 37.1 (Feb. 2005), pp. 171–175. ISSN: 0097-8418. DOI: 10.1145/1047124.1047409. URL: https://doi.org/10.1145/1047124.1047409 (cit. on p. 26).

[LFT+17] P. Lenberg, R. Feldt, L. G. W. Tengberg, I. Tidefors, D. Graziotin. "Behavioral software engineering-guidelines for qualitative studies". In: *arXiv preprint arXiv:1712.08341* (2017) (cit. on pp. 20, 51).

[LFW15] P. Lenberg, R. Feldt, L. G. Wallgren. "Behavioral software engineering: A definition and systematic literature review". In: *Journal of Systems and software* 107 (2015), pp. 15–37 (cit. on p. 20).

[LS03] J. P. Leal, F. Silva. "Mooshak: a Web-based multi-site programming contest system". In: *Software: Practice and Experience* 33.6 (2003), pp. 567–581. DOI: 10.1002/spe.522. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.522. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.522 (cit. on p. 26).

[Max08] J. A. Maxwell. "Designing a qualitative study". In: *The SAGE handbook of applied social research methods* 2 (2008), pp. 214–253 (cit. on p. 29).

[MBS99] M. K. Mount, M. R. Barrick, J. P. Strauss. "The joint relationship of conscientiousness and ability with performance: Test of the interaction hypothesis". In: *Journal of Management* 25.5 (1999), pp. 707–721 (cit. on pp. 22, 23).

[McD19] G. L. McDowell. *Cracking the coding interview: 189 Programming Questions and Solutions*. CareerCup, 2019 (cit. on pp. 24, 37, 42, 52).

[MCM05] R. R. McCrae, P. T. Costa Jr, T. A. Martin. "The NEO–PI–3: A more readable revised NEO personality inventory". In: *Journal of personality assessment* 84.3 (2005), pp. 261–270 (cit. on p. 22).

[Mea] E. D. Measurement Instrument Database for Social Sciences. *Scale of Positive and Negative Experience (SPANE)*. URL: https://www.midss.org/content/scale-positive-and-negative-experience-spane-0 (cit. on p. 55).

[MFP04] J. Moutafi, A. Furnham, L. Paltiel. "Why is conscientiousness negatively correlated with intelligence?" In: *Personality and Individual Differences* 37.5 (2004), pp. 1013–1022 (cit. on p. 22).

[MJ92] R. R. McCrae, O. P. John. "An Introduction to the Five-Factor Model and Its Applications". In: *Journal of Personality* 60.2 (1992), pp. 175–215. DOI: 10.1111/j.1467-6494.1992.tb00970.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-6494.1992.tb00970.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-6494.1992.tb00970.x (cit. on pp. 21, 48).

[MKG12] J. Mongan, N. S. Kindler, E. Giguère. *Programming interviews exposed: secrets to landing your next job*. John Wiley & Sons, 2012 (cit. on pp. 24, 42, 52).

[Mye] I. B. Myers. *The Myers Briggs Foundation*. URL: https://www.myersbriggs.org/my-mbti-personality-type/ (cit. on p. 21).

[Mye62] I. B. Myers. "The Myers-Briggs Type Indicator: Manual (1962)." In: (1962) (cit. on p. 21).

[Ned85] A. J. Nederhof. “Methods of coping with social desirability bias: A review”. In: *European journal of social psychology* 15.3 (1985), pp. 263–280 (cit. on p. 71).

[Pat90] M. Q. Patton. *Qualitative evaluation and research methods*. SAGE Publications, inc, 1990 (cit. on p. 49).

[PY18] R. Poonam, C. M. Yasser. “An experimental study to investigate personality traits on pair programming efficiency in extreme programming”. In: *2018 5th International Conference on Industrial Engineering and Applications (ICIEA)*. IEEE. 2018, pp. 95–99 (cit. on p. 21).

[Que09] N. L. Quenk. *Essentials of Myers-Briggs type indicator assessment*. Vol. 66. John Wiley & Sons, 2009 (cit. on p. 21).

[RHS17a] T. Rahm, E. Heise, M. Schuldt. “Measuring the frequency of emotions—validation of the Scale of Positive and Negative Experience (SPANE) in Germany”. In: *PloS one* 12.2 (2017) (cit. on pp. 22, 47).

[RHS17b] T. Rahm, E. Heise, M. Schuldt. “Measuring the frequency of emotions—validation of the Scale of Positive and Negative Experience (SPANE) in Germany”. In: *PLoS ONE* 12.2 (Feb. 2017), e0171288. DOI: 10.1371/journal.pone.0171288 (cit. on pp. 22, 47).

[RMSA12] M. Rehman, A. K. Mahmood, R. Salleh, A. Amin. “Mapping job requirements of software engineers to Big Five Personality Traits”. In: *2012 International Conference on Computer & Information Science (ICCIS)*. Vol. 2. IEEE. 2012, pp. 1115–1122 (cit. on p. 21).

[RP10] L. Rupsiene, R. Pranskuniene. “The variety of grounded theory: Different versions of the same method or different methods”. In: *Socialiniai mokslai* 4.70 (2010), pp. 7–20 (cit. on p. 53).

[SC90] A. Strauss, J. Corbin. “Basics of Qualitative Research: Grounded Theory Procedures and Techniques, Sage Publications, Inc”. In: (1990) (cit. on pp. 53, 54, 59).

[SC97] A. Strauss, J. M. Corbin. *Grounded theory in practice*. Sage, 1997 (cit. on p. 54).

[SCEB11] A. Sbaraini, S. M. Carter, R. W. Evans, A. Blinkhorn. “How to do a grounded theory study: a worked example of a study of dental practices”. In: *BMC medical research methodology* 11.1 (2011), p. 128 (cit. on p. 53).

[SJ17] C. J. Soto, O. P. John. “The next Big Five Inventory (BFI-2): Developing and assessing a hierarchical model with 15 facets to enhance bandwidth, fidelity, and predictive power.” In: *Journal of personality and social psychology* 113.1 (2017), p. 117 (cit. on p. 21).

[SKL+14] J. Siegmund, C. Kästner, J. Liebig, S. Apel, S. Hanenberg. “Measuring and modeling programming experience”. In: *Empirical Software Engineering* 19.5 (2014), pp. 1299–1334 (cit. on p. 47).

[SPP08] S. Salinger, L. Plonka, L. Prechelt. “A coding scheme development methodology using grounded theory for qualitative analysis of pair programming”. In: *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments* (2008) (cit. on pp. 53, 54, 71).

[WGW19] M. Wyrich, D. Graziotin, S. Wagner. “A theory on individual characteristics of successful coding challenge solvers”. In: *PeerJ Computer Science* 5 (2019), e173 (cit. on pp. 3, 13, 17, 22, 60, 73, 75).

[YO12] M. Yilmaz, R. V. OConnor. “Towards the understanding and classification of the personality traits of software development practitioners: Situational context cards approach”. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE. 2012, pp. 400–405 (cit. on p. 21).

All links were last followed on September 28, 2020.

A Appendix A

Persönlicher Fragebogen

Zur Studie über Coding challenges.

* Erforderlich

1. E-Mail-Adresse *

Abschnitt persönliche Angaben

Tragen sie im folgenden Abschnitte bitte ihre persönlichen Daten ein, die wie schon zuvor beschrieben nicht an Dritte weitergereicht werden.

2. Vor- und Nachname. *

3. Alter *

4. Geschlecht *

Markieren Sie nur ein Oval.

- ◯ Männlich
- ◯ Weiblich
- ◯ Divers

5. Ich..

Markieren Sie nur ein Oval.

- ◯ studiere momentan
- ◯ habe mein Studium bereits beendet

6. Ich studiere bzw. Ich habe studiert (nur der höchste Grad notwendig): *

Markieren Sie nur ein Oval.

- ◯ M.Sc Softwaretechnik
- ◯ B.Sc Softwaretechnik
- ◯ M.Sc Informatik
- ◯ B.Sc Informatik
- ◯ Sonstiges: _____

7. Im wievielten Fachsemester befinden sie sich? Bzw. Wieviele Semester haben sie studiert? *

_____

8. Aktueller Notenschnitt *

_____

9. Aktueller Notenschnitt des Bachelors, falls sie diesen bereits beendet haben.

_____

Abschnitt Programmiererfahrung

10. Wie häufig haben sie im letzten Jahr Erfahrungen mit Coding Challenges gemacht? *

Markieren Sie nur ein Oval.

- ◯ Nie
- ◯ 1-2 mal pro Semester
- ◯ 1-2 mal pro Monat
- ◯ Einmal pro Woche
- ◯ 2-3 mal pro Woche
- ◯ Täglich

11. Programmiererfahrung in Jahren *

12. Programmiererfahrung mit Java in Jahren *

13. Wie hoch würden Sie ihre Programmiererfahrung einschätzen verglichen mit ihren (Ex-) Studienkollegen.

Markieren Sie nur ein Oval.

	1	2	3	4	5	6	7	8	9	10	
Sehr unerfahren											Sehr erfahren

14. (Vorrausgesetzt sie haben ihr Studium bereits abgeschlossen) Wie hoch würden Sie ihre Programmiererfahrung gegenüber ihren Arbeitskollegen in vergleichbaren Positionen einschätzen?

Markieren Sie nur ein Oval.

	1	2	3	4	5	6	7	8	9	10	
Sehr unerfahren											Sehr erfahren

15. Arbeitserfahrung mit Tätigkeitsschwerpunkt auf Softwareentwicklung in Unternehmen in Jahren?

16. Falls Sie an Open-Source Projekten mitarbeiten, geben sie bitte einen Link zu Ihrem Profil an.

17. Falls sie einen "Stack Overflow"-Account besitzen, geben Sie bitte einen Link zu Ihrem Profil an

_____

Abschnitt aktuelle Gefühlslage

Bitte denken Sie an das, was Sie in den letzten 4 Wochen getan und erlebt haben.
Anschließend kreuzen Sie bitte in der folgenden Liste an, wie häufig Sie sich so gefühlt haben.

18. positiv *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

19. negativ *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

20. gut *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

21. schlecht *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

22. angenehm *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

23. unangenehm *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

24. glücklich *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

25. traurig *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

26. von Freude erfüllt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

27. ängstlich *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

28. zufrieden *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

29. wütend *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
sehr selten oder Nie	◯	◯	◯	◯	◯	sehr oft oder immer

B Appendix B

Persönlichkeitstest

Im Folgenden finden Sie eine Reihe von Beschreibungen, die auf Sie zutreffen können oder nicht. Bitte kreuzen Sie eine Zahl unter jede der aufgeführten Beschreibungen an, um anzuzeigen, wie sehr diese Aussage auf Sie zutrifft oder nicht zutrifft.

* Erforderlich

1. Vor- und Nachname *

_____

Skala

0. Trifft überhaupt nicht zu.
1. Trifft wenig zu.
2. Trifft teils/teils zu
3. Trifft gut zu.
4. Trifft sehr gut zu.

Kontext

Ich sehe mich als jemanden, der ...

2. 1. gesprächig ist, sich gerne unterhält *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

3. 2. dazu neigt, andere zu kritisieren *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

4. 3. Aufgaben gründlich erledigt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

5. 4. deprimiert, niedergeschlagen ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

6. 5. orginell ist, neue Ideen entwickelt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

7. 6. eher zurückhaltend und reserviert ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

8. 7. hilfsbereit und selbstlos gegenüber anderen ist. *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

9. 8. etwas achtlos sein kann *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

10. 9. entspannt ist, sich durch Stress nicht aus der Ruhe bringen lässt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

11. 10. vielseitig interessiert ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

12. 11. voller Energie und Tatendrang ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

13. 12. häufig in Streitereien verwickelt ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

14. 13. zuverlässig und gewissenhaft arbeitet *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

15. 14. leicht angespannt reagiert *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

16. 15. tiefsinnig ist, gerne über Sachen nachdenkt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

17. 16. begeisterungsfähig ist und andere mitreißen kann *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

18. 17. nicht nachtragend ist, anderen leicht vergibt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

19. 18. dazu neigt, unordentlich zu sein *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

20. 19. sich viele Sorgen macht *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

21. 20. eine lebhafte Vorstellungskraft hat, phantasievoll ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

22. 21. eher still und wortkarg ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

23. 22. anderen Vertrauen schenkt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

24. 23. bequem ist und zur Faulheit neigt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

25. 24. ausgeglichen ist, nicht leicht aus der Fassung zu bringen *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

26. 25. erfinderisch und einfallsreich ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

27. 26. durchsetzungsfähig und energisch ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

28. 27. sich kalt und distanziert verhalten kann *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

29. 28. nicht aufgibt ehe die Aufgabe erledigt ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

30. 29. launisch sein kann, schwankende Stimmungen hat *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

31. 30. künstlerische und ästhetische Eindrücke schätzt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

32. 31. manchmal schüchtern und gehemmt ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

33. 32. rücksichtsvoll und einfühlsam zu anderen ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

34. 33. tüchtig ist, flott arbeitet *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

35. 34. ruhig bleibt, selbst in angespannten Situationen *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

36. 35. routinemäßige und einfache Aufgaben bevorzugt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

37. 36. aus sich herausgeht, gesellig ist *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

38. 37. schroff und abweisend zu anderen sein kann *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

39. 38. Pläne macht und diese auch ausführt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

40. 39. leicht nervös und unsicher wird *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

41. 40. gerne Überlegungen anstellt, mit Ideen spielt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

42. 41. nur wenig künstlerische Interessen hat *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

43. 42. sich kooperativ verhält, Zusammenarbeit dem Wettbewerb vorzieht *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.						Trifft sehr gut zu

44. 43. leicht ablenkbar ist, nicht bei der Sache bleibt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

45. 44. sich gut in Musik, Kunst und Literatur auskennt *

Markieren Sie nur ein Oval.

	0	1	2	3	4	
Trifft überhaupt nicht zu.	◯	◯	◯	◯	◯	Trifft sehr gut zu

C Appendix C

Einleitung zur Studie

Patrick Lux

Juli 2020

1 Allgemeines

- Beschreibung der Studie
  - Bei der Studie geht es um die Beobachtung und Evaluierung einer Person während der Programmierung von Coding Challenges ähnlich zu derzeitig praktizierten technischen Interviews innerhalb von Vorstellungsgesprächen der Industrie.
- Studienaufbau
  - Der geplante Zeitraum der Studie liegt zwischen eineinhalb und zwei Stunden.
  - Die Studie besteht aus drei Coding Challenges, einem anschließendem Interview (je Challenge), einem persönlichen Fragebogen und einem Persönlichkeitstest.
  - Während der Studie wird der Bildschirminhalt des Teilnehmers sowie die Kommunikation zwischen Teilnehmer und Interviewer aufgezeichnet.
  - Die Coding Challenges werden durch ihren Kopfkommentar innerhalb des Codes erklärt, sollten Sie Fragen haben, bitte stellen Sie diese.
  - Die Lösungen der Coding Challenges sollen eingereicht werden. Diese werden zu einem späteren Zeitpunkt bewertet.
- Regeln
  - Während der Coding Challenges können Tips beim Interviewer eingeholt werden.
  - Die Nutzung von externen Quellen, speziell die Internetrecherche ist während der Coding Challenges untersagt.
  - Bitte schalten Sie ihr Smartphone auf stumm, lautlos oder aus und benutzen dieses während der Coding Challenges nicht. Auch andere elektronische Geräte mit Internetanschluss, beispielsweise Tablets, sind hier eingeschlossen.

  - Ich bitte Sie, sich während der Coding Challenges nicht vom PC bzw. Laptop zu entfernen, zwischen den Coding Challenges und/oder während des Interviews können gerne Pausen eingelegt werden.
  - Bitte verändern sie die vorgegebenen Methoden Signaturen nicht.
  - Es gibt ein Zeitlimit für jede der Challenges, dieses ist aus dem Kopfkommentar jeder Challenge zu entnehmen.

- Sie dürfen..
  - .. bei Bedarf zusätzliche Methoden erstellen.
  - .. die Main-Methode ändern, nicht aber die gegebenen Signaturen.
  - .. Zusätzliche Methoden erstellen (Hilfsmethoden)
  - .. den Code ausführen.
  - .. sich Notizen machen. Holen Sie sich gerne Stifte und Papier (Es wäre sehr nett, wenn Sie diese später mit einreichen)
  - .. jederzeit nachfragen, wenn etwas unklar ist.

- Bewertung
  - Korrektheit
  - Effizienz (O-Notation, Zeitkomplexität (time complexity))
  - Stil (Eleganz, Kommentare, Länge des Codes)
  - Auf die Platzkomplexität (space complexity) wird nicht eingegangen.
  - Eine vorzeitige Abgabe bietet keinen Vorteil bei der Bewertung, nutzen Sie ihre Zeit für mögliche Verbesserungen. Sollten sie dennoch vorzeitig abgeben wollen, sei Ihnen das gestattet.

2 Einverständniserklärung

- Ich wurde für mich ausreichend mündlich/schriftlich über die wissenschaftliche Studie informiert.
- Ich erkläre mich bereit, dass im Rahmen der Studie Daten über mich gesammelt werden und anonymisiert aufgezeichnet werden. Es wird gewährleistet, dass meine personenbezogenen Daten nicht an Dritte weiter gegeben werden. Bei der veröffentlichung in einer wissenschaftlichen Zeitung wird aus den Daten nicht hervorgehen, wer an dieser Studie teilgenommen hat. Ihre persönlichen Daten unterliegen dem Datenschutzgesetz.
- Mir ist bewusst, dass es sich bei den erfassten Daten um personenbezogene Daten handelt, speziell um mein Verhalten während des Lösens der Coding Challenges.
- Ich gestatte die veröffentlichung der aufgenommenen, personenbezogenen Daten in anonymisierter Form.

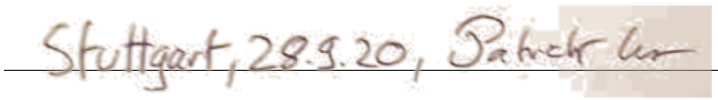
- Ich weiß, dass ich jederzeit meine Einverständniserklärung, ohne Angaben von Gründen, widerrufen kann, ohne dass dies für mich nachteilige Folgen hat.
- Ich weiß, dass die Teilnahme an dieser Studie freiwillig ist. Ich kann die Studie jederzeit, ohne Angabe von Gründen, abbrechen.
- Mir ist bekannt, dass alle Daten nach der Auswertung permanent gelöscht und damit unzugänglich gemacht werden.
- Mit der Teilnahme entsteht für mich kein Risiko, außer dass Sie mentale und physische Müdigkeit durch die Arbeitsbelastung erfahren. Die Vorteile liegen im Wissen, dass die Forscher aus den gesammelten Daten erhalten sowie der gewonnenen Erfahrung, die sich positiv auf potentielle Bewerbungsprozesse auswirken kann.
- Mit der vorstehend geschilderten Vorgehensweise bin ich einverstanden und bestätige dies mit meiner mündlichen Zusage zu Beginn der Aufnahme.

3 Fragen zu Beginn der Aufzeichnung

- Haben Sie bezüglich der Einverständniserklärung Fragen?
- Sind Sie mit dieser einverstanden?
- Sie studieren Informatik oder Softwaretechnik bzw. haben ein solches Studium abgeschlossen?
- Sind Sie mit Java vertraut?
- Sind Sie mit Eclipse und/oder IntelliJ IDEA vertraut?
- Haben Sie Fragen bevor wir anfangen?

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.



place, date, signature