Chemnitz University of Technology
Faculty of Computer Science
Professorship Software Engineering

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Master's Thesis

For obtaining the academic degree

Master of Science

Submitted by

Luisa Bartl

Matriculation number: -
Course of study: Course of study: Computer Science for Graduates in the Humanities or Social Science

Designing and Validating Code Comprehension Tests: An Empirical Studyon Task Design

Supervisors: Prof. Dr. -Ing. Janet Siegmund
Dr. Marvin Wyrich

Chemnitz, September 22,2025

Abstract

Code comprehension is a central activity in software development and research that has been insufficiently defined to date. Its measurement is characterized by heterogeneous approaches and a lack of theoretical foundation. This work examines how key cognitive facets of code comprehension can be precisely defined and translated into valid, reliable test tasks in order to enable more systematic research and practical application. By systematically incorporating test design aspects — such as feedback, task variability, motivation, and objectivity — the work contributes to providing practical and scientifically sound methods for measuring code comprehension. The empirical study was conducted with master's students in computer science, who completed various tasks related to code comprehension in Python in a 30-minute test. The test design combined different task and answer formats, supplementary personality and debriefing questions to maximize validity, reliability, and objectivity despite limited conditions. The results show that although students have strong analytical skills and critical thinking, they exhibit significant weaknesses in abstraction. In addition, typical error and strategy patterns were identified, and the diagnostic significance of reasoning for capturing mental models was highlighted. The study thus contributes to the theoretical clarification of the concept of code comprehension, provides empirical evidence on its cognitive facets, and opens up perspectives for standardized, fair, and practical testing procedures.

Table of Contents

Chapter 1

Introduction

Understanding source code — often summarized under the term code comprehension — is one of the central activities in software development and maintenance. Studies suggest that developers spend around 30% to 70% of their work time on reading and understanding already existing code, instead of writing new code (Xia et al., 2018, Minelli et al., 2015). Therefore, code comprehension is not only part of everyday programming practice but also a significant cost factor for developing software (Haiduc et al., 2010). Despite its importance, it is still unclear what exactly is meant by code comprehension (Wyrich, 2023) and how it can be measured validly (Wyrich et al., 2023).

A central problem is the lack of a generally accepted definition of code comprehension. Similar to research regarding the term of intelligence, there is no overarching theoretical framework that describes the cognitive processes involved in understanding source code (Calvin, 1994). As a result, studies differ vastly regarding their theoretical assumptions and measurement methods. Oftentimes, code comprehension is implicitly defined by the type of measurement – for example, by correctly answered comprehension questions, the time taken to locate errors, or by participants' self-assessments (Wyrich et al., 2023).

With the advancement of technology and the emergence of other measurement methods, such as eye-tracking, the resulting diversity not only makes it difficult to compare individual studies, but also complicates meta analyses and systematic theory formation.

In addition to research, this uncertainty seems to be relevant while learning and teaching programming. Since neither a uniform model nor a precise definition of code comprehension has been established to date, teaching and learning processes cannot be systematically tailored to it. A lack of basic understanding of code can have negative long-term effects on students and their progress in learning and delay their development into competent programmers (Goodfellow et al., 2025).

In light of this, an important goal of research is to develop a well-founded and as universal as possible definition of code comprehension and identify the underlying mental models. This is the only way to develop reliable and comparable measurement methods that can be used in both research and practice.

Therefore, this thesis aims at identifying central facets of code comprehension and collecting criteria important for designing tasks and tests to reliably measure code comprehension. In a broader sense, this contributes to a more sound definition of the concept.

We pose the following research questions:

> **R1: What is code comprehension, and what cognitive facets does it include?**
>
> **R2: How must tasks and tests be designed in order to obtain a differentiated picture of code comprehension abilities and measure them in a valid, reliable way?**
>
> **R3: To what extent do the tasks presented in this thesis both capture the intended facets of code comprehension and prove to be comprehensible as well as appropriately challenging for assessment, and what insights can be derived from them?**

In a broader sense, factors that may influence the measurement could also be found during the study.

Question **R1** is relevant for this thesis as it provides the theoretical foundation - the definition and facets of code comprehension - and at the same time, examines the practical usefulness of the definition.

R2, by contrast, builds on this by examining how tasks need to be designed in order to measure these facets reliably and in a differentiated manner. Therefore, it addresses the methodological core of the test. It is not sufficient to solely define the theoretical framework (**R1**). Instead, we must show how this definition can be translated into valid and reliable measurement instruments. Without well-designed testing and task development, code comprehension cannot be assessed in a differentiated manner, and measurements could be distorted by various factors.

Question **R3** ultimately, combines the theoretical foundation (**R1**) and the methodological design (**R2**) with a practical execution. This question provides empirical evidence on whether the designed tasks can measure code comprehension in a differentiated manner and are practical to use at the same time.

This thesis aims at taking a step closer towards establishing a theoretically sound and, at the same time, practically relevant basis for researching and promoting code comprehension.

The work begins with a brief overview of the development of the research field of "code comprehension," its resulting definition of the term used here, and its central cognitive facets (Chapter 2). Building on this, criteria for the design of tasks and tests as well as their potential applications are presented (Chapter 3). Chapter 4 then describes the methodological implementation and Chapter 5 describes the test tasks, questionnaires, and debriefing elements. The empirical part (Chapter 6) presents the test implementation and results, before Chapter 7 interprets them, analyzes the tasks, and derives implications for research. Finally, Chapter 8 discusses limitations while Chapter 9 draws conclusions and provides an outlook.

Chapter 2

Background and Related Work

Research on code comprehension underwent a continuous advancement in the past decades. Yet, it is still characterized by a central issue: to date there is no generally accepted definition of code comprehension, leading to the use of many different terms such as readability, understandability, program comprehension, or task difficulty within studies that are either used in parts as synonyms or overlap in meaning. Additionally, studies regarding code comprehension differ vastly in their used methods and measuring approaches, which further limits the comparability of the results (Wyrich et al., 2023). The following section outlines key contributions in the historical development of code comprehension.

2.1 Development of Research on Code Comprehension

An early work of Baecker, 1988 initially focused on the readability of source code. The authors combined the dimensions of *readability* and *comprehensibility* and showed that not only logic and programming language but also visual factors such as typography, layout, colors, and symbols have a significant impact on code comprehension. In their study, they tested students with proficient skills with the programming language C and therefore chose it as the language of their test. 18 comprehension questions had to be answered and the number of correct answers and the time needed was used to measure the object.

For the first time, this demonstrated that code comprehension can be reduced not only to cognitive processes, but also to the form of representation. Benander et al., 1996 later shifted the focus onto structural constructs and therefore compared the comprehensibility of recursive and iterative algorithms. Even though the term "code comprehension" appears explicitly in those studies, it remains undefined. Instead, code comprehension was operationalized as the ability to correctly understand the purpose of a code, which was measured in terms of processing time and accuracy. These early works illustrate that comprehending smaller code segments is a key prerequisite for software activities, such as debugging, maintenance, or reviews.

With the 2000s the research focus expanded further to didactic and cognitive perspectives.

Once again, code comprehension was closely linked to code reading. Lister et al., 2006 used the SOLO taxonomy model to compare novices and experts and examined their handling of short code examples in this context. In doing so, he coined the metaphor “not seeing the forest for the trees,” meaning that students may see and understand all parts of a construct or code, but not being able to grasp the relationships between them. This metaphor would later be used frequently in different studies. A concise, albeit not formally theoretical, definition of code comprehension has been proposed: "Students who were not able to read and describe code on a relational level are intellectually unprepared to write functional code themselves." In doing so, Lister highlighted the close connection between reading and writing code and formulated a perspective that was taken up repeatedly in subsequent works.

However, some research, such as Haiduc et al., 2010, used a different term, "program comprehension," which also suggests code comprehension looking at the used within the study. Although they also never defined the term further, the authors again showed the importance of reading code for comprehension by illustrating two different approaches used by developers: skimming (e.g., reading only the function header) and detailed reading (complete comprehension of the function body). This tension between efficiency and accuracy was repeated in later research, particularly in the context of task and test designs.

Since the 2010s, there has been a clear expansion in the methodologies used. Turner et al., 2014 examined with the use of eye-tracking whether the programming language impacts the process of comprehending. The study with students showed that Python has advantages over C++ when it comes to overview and bug-finding tasks, since Python was seen as easier to understand. Even though code comprehension was again not directly defined, the study made it clear that language choice can be a significant factor in comprehensibility. Already at this stage, technologies such as eye-tracking were established to further analyze visual attention processes while reading code. With that, code comprehension could be measured not only by performance indicators (accuracy, time), but also by psycho-physiological measures. By contrast, Börstler and Paech, 2016 deliberately refrained from measuring code comprehension solely on the basis of subjective readability scores. Instead, they combined open questions, free question formats, cloze tests, and time recordings to achieve a more multifaceted picture of code comprehension. Participants were not allowed to go back to earlier tasks or to take any notes and almost half of them had a high level of general programming experience. The evaluation of the answers was described as "Not strict, answers that are close in meaning are also accepted." This approach emphasizes the fact that code comprehension goes beyond purely recognizing syntax and semantics. Instead, it is about the ability to reconstruct contexts of meaning.

Asenov et al., 2016 added to this strand by analyzing the impact of visualizations on code comprehension. The use of visual legends in methods with multiple parameters has shown that visualization elements can increase productivity and comprehensibility. The authors emphasized that code reading is inherent in almost all software activities (including writing, debugging, and testing), which highlights its importance within and as a basic skill of code comprehension. They conducted their study with the use of Java.

In 2018, Bednarik et al., 2018 reinforced the integration of eye-tracking and video examples as methods to record differences between novices and experts while dealing with realistic programs. This study revealed significant differences in competence, although the low variability of the code examples (only two programs) limited the generalizability. They also did not offer a definition of code comprehension but mentioned reading as being part of the construct.

Stapleton et al., 2020 used Java programs to measure code comprehension by answering open questions. Therefore, eight different questions were used to cover aspects such as data access, arguments, function call conditions, and structural relationships. This explicitly operationalized code comprehension in various dimensions, even though no theoretical definition was formulated.

More current studies, such as Lewis, 2023, emphasized strategic dimensions of code comprehension. They described the term as the comprehension of already existing code and explicitly distinguished it from code writing and debugging. With think-aloud studies, typical strategies of novices and experts were identified, e.g., recognizing familiar elements, creating contextual references, and mentally going through different inputs. This brings the research closer to literature and reading research, which emphasizes similarities between reading strategies and code reading.

At the same time, Costa and Gheyi, 2023 and Wyrich et al., 2023 criticized missing foundations and standards. The former showed with the use of Python and eye-tracking experiments with beginners how *atoms of confusion* (small spellings or patterns in code written by others) have a partly contradictory influence on comprehension. Even though they did not explain what their definition of code comprehension is and what it includes. Wyrich et al., 2023 conducted a comprehensive systematic mapping study resulting in the analysis of 95 studies. Among other things, they showed that "comprehension" is used as a term in 76% of cases, while other terms such as "readability" or "task difficulty" occur much less frequently. It is also striking that activity terms are mostly used instead of properties and that "program comprehension" dominates, even though "code comprehension" would be more precise.

Methodologically, they found a high degree of heterogeneity, a lack of standards, and an unclear distinction between code comprehension and related constructs. They particularly criticized the use of inappropriate task types (e.g., pure recall), the neglect of industry code, and missing a clear cognitive model of comprehension.

AI-supported methods also became more popular at the same period. Denny et al., 2024, Smith IV et al., 2025b and Smith IV et al., 2025a used Large Language Models (LLMs) to automate the evaluation of "Explain in plain English" tasks. These enable more efficient analyses but raise questions about reliability and fairness. Initial studies see didactic potential in LLMs, especially for supporting novices. Still, within the work of Denny et al., 2024 a precise definition of code comprehension was not given.

Goodfellow et al., 2025 proposed an explicit definition of code comprehension: "Code comprehension refers to the ability to read, understand, and explain code effectively." Generally, code comprehension is now mostly seen as a multidimensional term (Denny

et al., 2024), where dimensions such as data flow recognition, structural understanding, purpose description, and context integration are distinguished. At the same time, the field remains characterized by terminological ambiguities and methodological heterogeneity.

The historic development shows code comprehension as an established but diffusely defined field of research. While early work emphasized primarily visual and algorithmic aspects (Baecker, 1988,Benander et al., 1996), the close connection between reading and writing code became more prominent (Lister et al., 2006) from the 2000s onwards. In the 2010s, new methodical innovations such as eye-tracking were added to used approaches (Turner et al., 2014, Bednarik et al., 2018), supplemented by differentiated task formats and visualizations (Börstler and Paech, 2016, Asenov et al., 2016).

A recurring pattern within code comprehension research is the strong focus on specific methods and task types. Researchers often used “Explain in plain English” tasks (Denny et al., 2024), supplemented by think-aloud studies or eye-tracking analysis (Bednarik et al., 2018). A central theme is the differences between novices and experts with a stronger focus on novices, which have been addressed in both, individual studies (Vielsack, 2024) and in systematic reviews (Wyrich et al., 2023). The latter study also takes a critical look at the composition of the test groups: more than half of the studies examine only students, while pure professionals are rarely represented (9.5%).

Furthermore, the terminology remains inconsistent throughout the history. While some studies refer to code comprehension as "program comprehension" (Lewis, 2023, Schulte et al., 2010, Haiduc et al., 2010, Stapleton et al., 2020) others use similar terms such as readability (Baecker, 1988).

Despite those differences, similar patterns can be found in the referenced studies: Java seems to be the dominant choice of programming language (Wyrich et al., 2023), while Python and C/C++, for example, are not seen as often; code snippets are predominantly generated artificially and presented on screens, and performance is usually assessed based on correctness and processing time. At the same time, there is a high degree of variability in design, as tasks and measurements are highly heterogeneous and often based on the intuition of the researchers (Wyrich et al., 2023).

Recent studies show that code comprehension is multidimensional and influenced by factors, such as programming language, visual representation, experience, strategies, and cognitive processes (Lewis, 2023, Costa and Gheyi, 2023). LLMs open up a new field that could change both, the operationalization and evaluation of code comprehension (Denny et al., 2024Smith IV et al., 2025b). Nevertheless, the field remains characterized by a lack of theoretical foundation, inconsistent terminology, and methodological inconsistencies (Wyrich et al., 2023). This highlights the need to develop a clear definition and a validated, reliable testing instrument for code comprehension — a goal of this work.

2.2 Definition and Facets of Code Comprehension used in this Study

The definition of code comprehension we used in this work is based on the systematic mapping study of Wyrich et al., 2023, which looks back on four decades of research in this field. This work takes studies into account that either directly examine code comprehension (each with its own definition) or indirectly test aspects that suggest code comprehension. Based on an overview of these results, an integrative definition was developed that combines the different methodological approaches and understandings.

Wyrich et al., 2023 identified a central problem within the research field: despite decades of research, we still lacked a consistent and comprehensive overview. Their mapping study closes this gap by describing the research field and used design practices of bottom-up-code-comprehension studies. In doing so, they differentiate central terms: Code comprehension is a subset of program comprehension and focuses exclusively on source code, while higher-level aspects, such as architecture or documentation, are excluded. Furthermore, it is important to avoid terminological ambiguities: "Understandability" and "comprehensibility" describe the same construct and solely differ on a linguistic level, while comprehensibility (property of the code) is not he same as comprehension (activity of readers). The same applies for: "readability" or "legibility", which merely describe how easy something is to read, but does not guarantee actual understanding.

The analysis of Wyrich et al., 2023 shows the high level of fragmentation within the terminology. In total, ten different terms were identified, with "comprehension" being used the most (76%). Other terms included "readability" (in seven studies), „task difficulty“ (in four studies) as well as others with a lower frequency. Most studies used terms of activity, while only a few referenced terms of property. Regarding the lexical modifiers attached to "comprehension", a heterogeneous picture emerged: "program" (in 47 studies), "code" (in 24 studies), or no addition (also in 24 studies). Especially problematic is the frequent use of program comprehension, even though the content of the study suggests code comprehension.

Based on the results of this study, we use the following definition of code comprehension for the scope of this thesis:

> *Code comprehension describes the process by which individuals analyze, interpret, and evaluate source code in order to (1) understand what the code does, (2) assess whether the code fulfills its intended functionality, and (3) recognize its structure and intention.*

This definition combines two levels: first, the dimension of process (analyzing, interpreting, evaluating), and second, the goal of this process (understanding, assessing, recognizing structure and intention). Additionally, three cognitive facets that are central for code comprehension can be differentiated:

> *Analytical skills: dissecting code into smaller parts to comprehend their functionality.*
>
> *Ability to abstract: integration of individual code elements into an overall understanding/-construct.*

Critical thinking: Evaluating the correctness, efficiency, and comprehensibility of code.

The strength of this definition lies in the fact that it emphasizes the active, procedural nature of code comprehension while also integrating the different objectives and cognitive dimensions. This creates a framework that can both systematically classify existing studies and structure future empirical work.

Methodologically, this approach can be seen as exploratory: it is not based upon a hypothesis, but rather on an observational and descriptive evaluation of the existing literature. This resembles an "exploratory validation study" or "pilot analysis", from which hypotheses for future research can be derived.

For this scientific work, the definition serves as a foundation for the development and evaluation of a competence test for measuring code comprehension. Within the scope of this study we will take an exploratory approach and use the test as part of a course to assess competence. However, there are many other possible application scenarios—for example, in empirical research, in the evaluation of teaching methods, or in industrial practice to support training and assessments.

Chapter 3

Conceptual Foundations

This chapter first explores and presents the potential applications of a test designed to measure code comprehension. The use cases presented in the first section show where and for what purposes the code comprehension test can be used. They illustrate that different scenarios place different demands on the test tasks, e.g., in terms of complexity, level of abstraction, or time frame.

Before such a test can be developed, key criteria influencing the design of the tasks must be defined. In order to develop a valid, reliable, and practical code comprehension test, it is first necessary to systematically clarify which criteria influence the design of tasks. Section 3.2 summarizes the criteria derived from the literature. These criteria form the methodological basis on which the specific test tasks are later designed, evaluated, and interpreted. The section thus establishes the connection between the theoretical concepts of code competence (Chapter 2) and the practical implementation of the test (Chapters 4–6), ensuring that the test design is systematic, comprehensible, and application-oriented. Therefore, this chapter provides an overview of the conceptual foundations of test development and explains which criteria play a role.

3.1 Potential Applications of a Code Comprehension Tests

For this master's thesis, we used the presented test as a competency review for code comprehension within the framework of a lecture. However, there are other possible examples of application, where the use of this test might be beneficial. Therefore, we will briefly describe different scenarios in the following sections. As a result, we will examine the objectives, target audiences, settings, and focal points, which may vary depending on the scenarios.

3.1.1 Use in Scientific Studies

Similar to the use within this master's thesis, the test could also be used in other scientific research and empirical studies to collect and analyze findings on code comprehension.

Researchers in the field of computer science didactics or program learning might benefit from a tool to systematically examine and measure code comprehension with various participant groups to gain insights into learning strategies, educational standards, and learning difficulties.

The tasks cover a wide range of difficulty levels and cognitive demands in order to evaluate various aspects of program comprehension. The validity and reliability of the test are crucial in order to achieve scientifically sound and comparable results.

3.1.2 Use within Examinations

Another possible use case, which is similar to this here-executed study, may present itself within examinations. In this case, teachers of IT, computer science degrees or related courses represent the target group. The aim is to objectively determine the progress of code comprehension and to check the performances based on standardized requirements. Therefore, the test may be used as (part of) an examination to evaluate the earlier discussed competencies of the students on various levels. As a result, the tasks include questions on code comprehension on a basic level, as well as more complex exercises. Especially important for the use in examinations is a coherent and differentiated assessment of competencies on different levels of difficulty.

3.1.3 Exam Preparation and Self-Assessment

In addition to its use as an exam itself, the task could also be used to prepare for one or as a form of self-study without regard to a course or test. Therefore, it could be used by learners or students who want to independently exercise or assess their skills to determine possible knowledge gaps and strengthen their abilities. The test will be used as an optional self-learning tool and should be (openly) available, for example via an online platform or as a PDF file. To increase the motivation, the tasks should be designed to be varied and realistic in the sense of being practical and closely resemble real-world scenarios. Different task types make it easier to determine and train possible weaknesses and to develop an understanding for the different facets of code comprehension. Feedback mechanisms, such as solutions and explanatory notes could be integrated into the test or made available elsewhere.

3.1.4 As part of Learners' Career Information

Students who would like to gain some experience with coding and get to know possible career fields in the IT sector could use this test to obtain a basic level of knowledge and concepts, as well as a realistic picture of possible requirements. The test could be offered as part of career information events in schools, universities, or fairs as a playful introduction to programming. To attract a younger target audience and to heighten their motivation and interest, the tasks should be designed in an interactive and creative way. In contrast to other presented use cases, the focal point should shift to more basic structures of code and simple intuitive tasks that playfully promote an understanding of program logic and problem solving in the IT sector.

3.1.5 Suitability Tests for the Job Market

As (part of) a suitability test for career guidance or within the framework of support programs, the test could be used on the one hand by employment agencies. By contrast, people who seek a career change or retraining, as well as job seekers with an interest in IT professions, could benefit from the use as well. Testing the suitability and basic knowledge of programming concepts to give out and receive recommendations for training programs within the IT sector represents the main objective of this use case. Therefore, the tasks should be on an elementary level and contain fundamental concepts of code comprehension to determine suitability. By using different levels of difficulty, specific strengths and potentials of the participants could be identified and used for even more individual consulting.

3.1.6 Recruitment and Hiring Processes

The test may be used as part of recruitment processes in the IT sector and is aimed at companies and organizations that want to hire developers and IT specialists. The main goal for using this test consists of evaluating the competencies of applicants regarding code comprehension and the validation of the corresponding requirements for the respective position. It may be used within the process of application, either before, after, or during a job interview. The main focus of this scenario are practical tasks mimicking everyday challenges within the job, that assess candidates' knowledge of coding and problem-solving abilities. This may include the interpretation and optimization of code, as well as the identification and correction of errors, among other things. Using a clear structure and different levels of difficulty, the test could provide a more precise assessment of whether applicants have the necessary knowledge to successfully master the challenges that they may face in everyday working life.

Table 3.1: Overview of Use Cases for Code Comprehension Tests

Use Case	**Target Audience**	**Task Types / Focus**
Scientific Studies	Researchers in CS education and program learning	Wide range of difficulty levels, cognitive demands; focus on valid and reliable measurement of code comprehension
Examinations	Students in IT/computer science courses; teachers	Basic to advanced code comprehension tasks; standardized assessment; evaluation of competencies at multiple difficulty levels
Exam Preparation / Self-Assessment	Learners/students for independent practice	Varied and realistic tasks; self-assessment of knowledge gaps; feedback via solutions/explanations; multiple task types to cover different facets of code comprehension

Use Case	Target Audience	Task Types / Focus
Career Information	Students exploring IT careers	Simple, playful, interactive tasks; focus on basic structures and intuitive understanding of programming logic
Suitability Tests for Job Market	Job seekers, career changers, participants in IT training programs	Elementary-level tasks; assessment of fundamental programming concepts; identification of strengths and potentials
Recruitment and Hiring Processes	Companies hiring developers/IT specialists	Practical tasks simulating real job challenges; code comprehension, problem-solving, debugging, optimization; tasks at varying difficulty levels

3.2 Literature-Based Criteria for Tasks/Tests

Determining which criteria of design need to be taken into account was one of the primary steps in creating a code comprehension test. Because of this, an iterative method was selected, in which independent reflection and subsequent discussions provided initial results.

The first round of identifying criteria involved brainstorming, discussions, and reviewing relevant sources such as studies from the field. It served to gather an initial set of potential criteria, which were then refined and expanded in the subsequent evaluation based on Moosbrugger and Kelava, 2020. In this process, new criteria were added and already defined criteria were revised. Below you can find a comprehensive list of the necessary components. At this point we should note that interactions between different criteria may exist.

For the test design in the scope of this work, it is important to choose all criteria that are suitable and important for the given use case. It should be noted that, depending on the use case, a different set of criteria or focal points of criteria might be necessary. Differences between the target groups should also be taken into account - for example, with regard to the choice and length of additional material or the complexity of the tasks - in order to adapt the tests optimally to the respective needs. One aim of this thesis is to combine the below-mentioned criteria and examine their reciprocal effects within the test design. In the following, the identified criteria will be presented and briefly described.

3.2.1 Complexity

The degree of complexity that influences how challenging a task is or is being perceived and what level of understanding is required to solve the task adequately.

Different gradations may enable testing of different groups of people, for example novices

or experts. According to Sweller, 2010 and his Cognitive Load Theory (CLT), cognitive load can be categorized into three different types: intrinsic load, germane load, and extraneous load. Accordingly, the definition of complexity used here can be equated with intrinsic load (due to the difficulty of the task) and extrinsic load (unnecessary cognitive load caused by inefficient design of learning materials). Used within the scope of a recruitment or hiring process, recruitment tests must be targeted to the position applied for and designed with appropriate complexity, for example, for beginners (e.g., for junior positions) or experts (e.g., for senior developers).

Various measures can be taken to ensure the most benefit and the least unnecessary cognitive load for the participants. Firstly, additional information can be presented via different senses (e.g., visual diagrams or auditory explanations) in order to reduce the burden of one modality alone. Unnecessary, duplicate, and distracting information should be avoided as a matter of principle. Further measures can be found in Beddow, 2018.

3.2.2 Duration

The influence of the time that is required on the concentration, motivation, etc., of learners. It can be interpreted as the time span members of the target group approximately need to solve the test or task or as the maximum time span given them to solve it. Depending on the target group, the time limit should be chosen carefully. On the one hand, a brief period of time could result in a feeling of stress for participants. However, an excessively extended time span may interfere with motivation and reasonableness and lead to subjective cognitive fatigue (Ackerman and Kanfer, 2009). The duration of the test depends not only on the participants and the external circumstances (e.g. maximum time available due to the duration of a teaching unit) but also on the subject matter to be measured. Depending on the scope of the concept (narrow or broad), the number of items per test is reduced or increased in order to test as many aspects of the concept as possible (Moosbrugger and Kelava, 2020).

3.2.3 Validity

The following information can be found in greater detail in Moosbrugger and Kelava, 2020. A key aspect of validity is the question of whether the test actually measures what it claims to measure. Depending on the interpretation of the test results, different types of validity exist, which will be described and subsequently discussed in the context of a (code comprehension) study.
Construct validity describes whether a test actually measures the theoretical concept it is intended to measure. Construct validity plays a central role here, as the test design must ensure that "code comprehension" is really being tested - and not just reading skills or general problem-solving ability.
Criterion validity is tested by examining whether the test results and an external criterion match. Scientific studies often compare the test results with other established measurement instruments or real programming performance. If there is strong agreement, this indicates

high criterion validity. However, if there is no clear link between the test scores and the results of another validated programming skills test, this could serve as a negative example of a lack of validity.
Face validity refers to whether the validity of a test appears meaningful and plausible at first glance - even for non-experts. If the test does not seem to make much sense to computer science students at first glance, or if the questions do not feel like real code comprehension, this could be a problem for face validity.
Content validity describes how well the test content matches the definition of the construct to be measured and all of its relevant aspects. If certain key capabilities are missing (e.g. analytical thinking, code optimization), this would be a negative example of low content validity.

Other types of validity are **validity of agreement, predictive validity, incremental validity, convergent validity, discriminant validity**, and **criteriaial validity**. These are not necessarily of relevance for this thesis but may be taken into consideration for other test designs or use cases and therefore should not be disregarded. Further information on this topic can be found in Moosbrugger and Kelava, 2020.

3.2.4 Variability and Reusability

Tasks should have a certain degree of variability to not to become too monotonous and possibly negatively affect the motivation of the participants or the didactic benefit. In addition, the tasks can be used several times by making minor adjustments, thus reducing the planning and design effort.

3.2.5 Feedback

Proper feedback ("information provided by an agent [...] regarding aspects of one's performance or understanding" (Hattie and Timperley, 2007)) can be a useful tool to increase the motivation and effort for participants and can lead to a better understanding of the tested constructs.

To be effective, feedback must be phrased clearly and be focused on and congruent with the participant's knowledge. Therefore, it should not propose a threat to the participant's self-esteem or be used solely as an assessment tool (Hattie and Timperley, 2007).

The choice of response medium offers different possibilities regarding the design of feedback. While digital tests can provide (digital) feedback directly following an answer and thereby may boost motivation, it can also lead to a learning effect that distorts the results for a scientific study. Whereas immediate feedback can cause this effect, later given feedback (e.g., for tests solved on paper) or feedback that has to be requested avoids it. Participants may find that less motivating.

3.2.6 Task Type

Different types of tasks test different cognitive abilities and levels of understanding. A variation of different task types can lead to a more varied test experience and therefore

possibly an increase in motivation. Moosbrugger and Kelava, 2020 describe the following task types:

Tasks with free response formats include short essay tasks and fill-in-the-blanks tasks. Short essay tasks allow testing reading comprehension and the application of knowledge. However, they are time-consuming and difficult to evaluate objectively, as a complex response category system is required. Fill-in-the-blanks tasks test only the reproduction of knowledge and can be designed to be assessed more objectively. Nevertheless, they may allow suggestive effects and ambiguities, which can make evaluation more difficult.

Tasks with a bound task format in contrast, take less time and enable a more objective evaluation. The tasks include matching and reordering tasks. Matching tasks are particularly suitable for testing knowledge, as they only require recognition and can be evaluated simply and objectively. A higher number of possible answers than necessary increases the ability to differentiate. Reordering tasks, by contrast, test more complex cognitive abilities such as reasoning or understanding cause-and-effect relationships. Reordering tasks can be used to test, for example, logical reasoning, cause-and-effect connections, or the ability to abstract.

Selection tasks, such as dichotomous questions or multiple-choice tests, are widely used. Dichotomous tasks (e.g., yes/no or true/false) are easy to evaluate, but have a high guess probability and are therefore rarely used in performance tests. Additionally, creating them is considered more difficult, as one must ensure that guessing the answers is not too easy. Multiple-choice questions require the selection of several possible answers, whereby the choice of suitable distractors and the random positioning of the correct answer are decisive for the quality. Careful design can minimize random ticking by including incorrect answers in the scoring. Assessment tasks, such as analog scale or rating scale tasks, require a subjective assessment by the participants and may especially be useful for questions of personality.

Finally, there are also **atypical answer formats** that are particularly suitable for creative questions. Short essays or supplementary questions are also often used here. Combinations of different types of questions are possible as well, such as multiple-choice questions where a justification for the chosen answer is required.

3.2.7 Level of Abstraction

Distinct levels of abstraction can test different aspects of program comprehension and offer the possibility to focus specifically on certain facets of the tested criterion. Therefore, it can lead to an improved recognition of individual competencies. Within exams, different levels of abstraction can be used to measure different competencies. In the area of programming, for example, the following levels of abstraction could more accurately identify students' individual strengths and weaknesses: Syntax level (e.g. recognizing syntax errors), Semantic level (e.g. understanding what a section of code does), Algorithmic level (e.g. analysis and optimization of algorithms). A well-designed test can differentiate between beginners and advanced learners by asking questions at different levels of abstraction.

3.2.8 Contextualization

Adjusting the context or setting of a task to either real or fictitious use cases may have different impacts on the motivation for participants. Settings that are based on possible real-life work experience could possibly be more attractive for students or young professionals.

Fictitious scenarios, by contrast, could appeal to younger participants (Cordova and Lepper, 1996). Used within the scope of exam preparation, the tasks should be as near to the actual exam tasks as possible. The test could also be used to assess applicants to check their suitability for IT professions. For the assessment to be meaningful, the tasks must include realistic, practical scenarios, e.g., Optimizing an existing algorithm or implementing a small function with real-world requirements. This type of contextualization also increases the motivation of applicants, as the tasks are directly related to their future work. If the test in recruiting only asked abstract theoretical questions (e.g. definitions of programming language syntax), this could reduce the validity of the assessment, as the test does not directly measure whether a person will be successful in the job. In this case, the contextualization would be insufficient, as it does not connect to the actual world of work.

Apart from motivation, a Contextualized Teaching and Learning approach (CTL) could positively affect the learning of the students by connecting knowledge with personal live experiences (Nawas, 2018).

3.2.9 Response Format

The type of medium that is used to take the test (e.g. paper and pen, verbal, digital answer options with the use of IDEs or similar tools).

Different media enable different options for implementation and evaluation. Among other things, digital tests allow the results and current status of the test to be saved and retrieved directly, times of individual tasks to be recorded, etc. In addition, there are more options regarding the design of the test and the use of additional materials such as images or videos and interactive design options. This is particularly useful in the context of working with code, as working with code on paper is not very realistic and may cause more mistakes and errors through spelling mistakes.

However, answering on paper with a pen has the advantage that the test can be carried out independently of the device and platform, which opens up the tests to a wider range of people. However, it requires more material and you are more limited in the design and use of additional material.

3.2.10 Motivation and Reasonability

Motivation as a psychological concept describes the drive that makes people pursue certain goals or actions and stick to them in order to achieve them (Urhahne and Wijnia, 2023).

Some use cases may have a higher degree of participant motivation due to the nature and structure of the scenario. This could include exams and exam preparation where passing an exam is seen as a direct or indirect goal. Other use cases, for example, scientific studies,

may require an external resource, such as compensation to increase participation, since their participants usually do not gain any reward.

The criterion of reasonableness is met when the tests' negative aspects (e.g., excessive time, mental, or physical hardship) do not exceed its benefits.

However, this is highly dependent on criteria, such as use case, individual opinion of participants, and social norms (Moosbrugger and Kelava, 2020).

If adequate reasonableness or motivation is not sufficiently given or not roughly balanced, there is a risk that participants will not complete the test at all or not truthfully and to the best of their ability, thus falsifying the results.

3.2.11 Additional Materials

Additional materials may need to be provided to ensure, that the participants have all the requirements and knowledge they need to solve a task. This may include specific explanations, diagrams and other graphics or code. It is important, that those materials fit the task and its complexity and are helpful without generating too much overhead/information that overwhelms the participant. Additionally, the materials should be visually appealing and clearly designed.

3.2.12 Error Tolerance and Assessment

Clearly defining what constitutes a mistake and why it is classified as one is necessary to ensure objective evaluation and should be done before executing the test. Certain types of tasks are shown to be more beneficial than others in that context. For example, free text tasks have a higher assessment effort than multiple choice questions, due to the various possibilities to answer them in different scopes.

There are different ways to assess a task, and the system behind it should be designed before the actual assessment. Different task types may be more beneficial with some assessment systems than others.

While fixed answer formats, such as multiple choice, leave little room for interpretation within the answers, free answer formats usually open up the problem of being ambiguous. Accordingly, a clear and structured definition of what is considered 'correct' or 'incorrect' is essential. Sample answers can support the evaluation process.

It is also important to define the evaluation scheme in advance. Among other things, an evaluation using points can be useful here.

3.2.13 Objectivity

A test is referred to as being objective, which means it is a test procedure so well-defined that it may be executed regardless of time, place, test administrator, or evaluating person, and still leads to the same result and interpretation for a particular test participant regarding the trait being assessed. This is where the test manual's implementation, assessment, and result interpretation guidelines are useful (Moosbrugger and Kelava, 2020). In scientific

studies, it is essential that the test results are reproducible regardless of time, location, test administrator, or the person evaluating the test. A well-defined test manual with clear implementation, scoring, and interpretation guidelines ensures that the results are comparable and reproducible. Standardized test conditions ensure that external criterias (e.g., different test administrators or different environments) do not influence the results.

3.2.14 Scaling

A test is perceived as scalable when the obtained test values (numerical relative) accurately represent the real characteristic relations (empirical relative) (Moosbrugger and Kelava, 2020). In examinations, it must be ensured that the test values (scores, grades) reflect the actual abilities of the students. A good scaling system means that a student with better code comprehension skills also receives a higher score than someone with weaker skills. This requires a clear hierarchy of tasks so that the point scale correctly reflects the actual differences in performance. If the scale is not linear or differentiated enough, students with different levels of ability could receive similar scores. Scoring scales that are unsuitable for the application (e.g. only "correct/incorrect" without partial points) can also lead to the test not correctly reflecting the true differences in performance. In recruitment and hiring, scaling is more of a secondary issue, as qualitative criterias (soft skills, experience) are often included, meaning that test scores alone are not always sufficient. Scaling is less important for self-tests, as there is no formal assessment.

3.2.15 Standardization

A test is considered standardized if a reference system has been created for it, with the help of which the results of a test person can be clearly classified and interpreted in comparison to the characteristic values of other persons in the target group (Moosbrugger and Kelava, 2020). In scientific studies, it is crucial that the test results can be systematically recorded, compared, and interpreted. This requires a reference system (e.g. norm values, comparison groups) in order to classify the test results of individual people uniformly and objectively in relation to a larger population. A well-standardized test makes it possible to analyze differences between different groups, e.g., between beginners and advanced programmers. Standardization can ensure that the test provides comparable results across different studies. If the test is not standardized, there is no reliable basis for comparison, so results cannot be interpreted unambiguously. Examinations also require standardization, but this is often only valid within a university or a course, not across larger target groups. Standardization could help with recruiting, but companies often assess candidates individually according to internal criteria, so there is no general basis for comparison Standardization is less important for self-tests, as they are often used for personal orientation and do not require a formal assessment.

3.2.16 Economy

Economy concerns the cost-effectiveness of a test. The central variable is the costs incurred, which should be kept as low as possible without negatively affecting the quality or other quality criteria of the test. Costs can be of a financial or time-related nature and represent

an expense (Moosbrugger and Kelava, 2020). One measure to keep costs and effort low would be to carry out group tests instead of individual tests. The "Exam Preparation and Self-Assessment" use case shows particularly well how economy can work: high cost efficiency through digital provision and flexible use, but potential quality problems if cost reduction comes at the expense of validity or user-friendliness. Scientific studies often require high investments in data analysis, standardization and test design, making economy a problem here. Companies often invest in complex test procedures to assess applicants validly, which can result in high costs. These tests often have to be organized through institutions, which incurs costs for personnel, consulting, and infrastructure.

3.2.17 Usability

A test is considered as usable if the measured feature represents a practical relevance and if the decisions, made within that process generate more benefit than harm. Additionally the criteria is met, when the test has useful options for application (Moosbrugger and Kelava, 2020).

3.2.18 Fairness

If the execution and the results of a test do not discriminate against people solely based on their gender, race, sociocultural background or any other individual characteristic that is irrelevant to the characteristic tested, the criterion of fairness is met (Moosbrugger and Kelava, 2020). Additionally, a test should be designed such that as many people as possible have access to it (Beddow, 2018).

3.2.19 Falsification

The requirements for this criterion are fulfilled when the procedure is designed in a way that prevents the test taker from purposefully manipulating an inappropriate test behavior in an effort to falsify the specific features of their test results (Moosbrugger and Kelava, 2020). Exams often have strict proctoring mechanisms, but students could cheat by using unauthorized aids if the test is not well designed. As self-tests are voluntary tests, there is no need for manipulation, so this criterion is less relevant.

3.2.20 Reliability

Reliability is ensured by measuring the chosen characteristic accurately, meaning without any measurement errors (Moosbrugger and Kelava, 2020). High reliability is essential in scientific studies, as the results must be reliable and reproducible. The test must measure in a standardized and error-free manner so that researchers can gain systematic insights into code comprehension. If the test is poorly designed or there is no clear differentiation between difficulty levels, reliability may suffer. Inconsistent scoring methods or ambiguous items could lead to different results in repeated measurements. If the test is not validated or external criterias (e.g. motivation, daily form of the participants) strongly influence the results, reliability would be jeopardized.

In summary, all identified criteria for code comprehension tests are presented below. We follow the classification according to Moosbrugger and Kelava, 2020 into general quality criteria and secondary quality criteria.

The first group, general criteria, include quality requirements that are of a general planning nature and do not require any special test-theoretical substantiation. They need to be taken into consideration for all aspects of construction of any test and are especially important during early stages of planning and the execution (Moosbrugger and Kelava, 2020).

The second group, secondary quality criteria, includes criteria whose significance relates more to the optimization and supplementation of the test and which require special theoretical justification (Moosbrugger and Kelava, 2020).

3.2.21 Summary of Criteria

In total, we identified 20 central criteria that are relevant for constructing and executing code comprehension tests. These range from fundamental aspects such as complexity, duration, validity, objectivity, and reliability to supplementary considerations such as variability, feedback, motivation, fairness, and falsification. While the general criteria are primarily essential for planning and implementation, the secondary criteria serve to optimize and contextualize the test design. This makes it clear that the selection and weighting of individual criteria always depends on the respective context of use and the target group. Table 3.2 provides an overview of all criteria and their classification into general and secondary quality criteria.

Table 3.2: Overview of Criteria for Code Comprehension Tests

Criteria	**Description / Comments**	**Type**
Complexity	Determines how challenging a task is, based on intrinsic and extraneous cognitive load. Critical for adapting difficulty to target groups (novices vs. experts).	General
Duration	Time allocated affects concentration, motivation, and fatigue. Must match scope and target group.	General
Validity	Measures whether the test actually assesses code comprehension. Includes construct, criterion, content, and face validity.	General
Variability / Reusability	Ensures tasks are not monotonous and can be reused with minor adjustments.	Secondary
Feedback	Provides performance information to aid learning and motivation. Timing and medium impact effectiveness.	Secondary
Task Type	Different formats test different cognitive skills (free response, bound response, selection, atypical).	General
Level of Abstraction	Tasks can operate on syntax, semantic, or algorithmic level to measure distinct competencies.	General

Criteria	Description / Comments	Type
Contextualization	Realistic or fictitious settings impact motivation and relevance. Must connect to practical coding experience.	Secondary
Response Format	Medium of answering (paper, digital, IDE) affects usability and error likelihood.	Secondary
Motivation / Reasonability	Tasks should be motivating and reasonable; excessive hardship can reduce validity.	General
Additional Materials	Support comprehension without overwhelming participants; diagrams, explanations, code snippets.	Secondary
Error Tolerance / Assessment	Clear rules for mistakes and evaluation; affects objectivity and reliability.	General
Objectivity	Test results should be reproducible regardless of administrator or location.	General
Scaling	Scores should accurately reflect relative ability levels.	Secondary
Standardization	Provides reference system for comparison across participants or groups.	Secondary
Economy	Cost-effectiveness, balancing resource use with test quality.	General
Usability	Practical relevance and usefulness for application.	General
Fairness	Ensures no discrimination based on irrelevant characteristics.	General
Falsification	Prevents manipulation by participants.	General
Reliability	Measures characteristic accurately without error. Essential for reproducibility.	General

Chapter 4

Methodology

Building on the theoretical and conceptual foundations presented in Chapter 3, Chapter 4 describes the practical methodological approach used to construct the test. The factors and quality criteria identified earlier – such as complexity, validity, motivation, and feedback – guided the design of tasks and test structure. The goal of Chapter 4 is to outline the methodological foundations and considerations that led to the construction of the test. For that we followed an iterative approach: at first we designed tasks (partly based on Bartl, 2024) that were then revised and refined in several discussions. At the same time we collected possible questions for debriefing to identify any disruptive factors early on.

Additionally, we drafted possible justifications for correct answers, which should support both the design and subsequent evaluation. Due to the uncertain state of research and a missing generally accepted definition of code comprehension and its relevant characteristics, we chose an intuitive and heuristic strategy. This approach made it possible to develop the first valid test elements while leaving room for adjustments and further development.

4.1 Selection and Design of Test Items

Below, the general and external conditions for carrying out test are presented. These ultimately form the external framework of the test. In addition, the criteria that are generally relevant for the tasks of this specific use case are discussed once again. Except for **Measured Variable Participants, Programming Language, and Test Duration**, the following sub-chapters have been taken from the previous overview and adapted to this test. The rest resulted from the nature of the test, which requires clarification of these questions for further design. These could therefore also be regarded as additional secondary criteria that are explicitly required for this test. Task-specific features and criteria are then described in Chapter 5.

4.1.1 Measured Variable

This tests intends to examine students code comprehension abilities and their facets. Therefore, those are the measured variable in this case. In the scope of this test, the previously presented definition of code comprehension is multidimensional. That means that the definition is based on multiple aspects that need to be taken into consideration. As a result, the items can also be multidimensional and examine the various characteristics alone or in combination. The test itself represents mainly as a performance test with elements of a personality test. This is because the characteristic to be tested is based on cognitive abilities and the answers to the relevant items can be rated as either correct or incorrect.

4.1.2 Participants

The group of participants for this study consisted of students participating in the course "Empirical Software Engineering" which is targeted at masters students in the field of computer science. It can therefore be assumed that the participants have a background in any computer science degree program. In total, 34 students participated in the execution of the test.

4.1.3 Choice of Programming Language

The programming language should be chosen according to the involved test group and their acquired skills and knowledge. Since the background of the participants and their programming knowledge was relatively heterogeneous and unknown concerning their learned programming languages, we had to find a language that suited the majority of the skills of the group. In this case, the choice of Python as the only programming language is therefore based on the assumption that, with its relatively simple set of syntax rules, it may appear easier to understand for people without any further experience in Python. Unfortunately, the scope of this thesis did not allow us to offer additional programming languages to choose from. If that can not be ensured, no programming language-specific questions should be asked.

4.1.4 Test Duration

Since the test was taking place within a lecture of the course "Empirical Software Engineering" and after another experiment, the students had a maximum time limit of 30 minutes for answering the question. After that time span, they were given another three minutes to finish the test and give comments about it, if needed.

Five task complexes were constructed, with each containing 1-3 sub-tasks. There are therefore a total of twelve tasks. An overview of the tasks can be found both in Chapter 5 and in the appendix. The basic idea behind the number of items was to have as many items as possible (preferably at different levels of difficulty) in order to achieve a more accurate measurement value and thus higher reliability. However, due to the time constraints, it was highly unrealistic to answer all items adequately. Nevertheless, to ensure that each task received at least a few answers, ten different test versions were offered, which differed

only in terms of their order. However, sub-tasks that belonged to a task complex were always presented one after the other. Another common practice would be to sort the items according to difficulty. By putting the easiest tasks first, the aim is to avoid losing a lot of time at the beginning. However, this was not an option in this case due to the aforementioned topic.

4.1.5 Validity

The criteria of validity proved to be problematic in the context of this thesis, since there is no universally valid definition of code comprehension or underlying theoretical concept in science. Therefore, part of this thesis is to contribute to finding that. Consequently, this can only be compared with the here given definition. However, this work serves as an expandable proposal.

Construct Validity The test was explicitly designed to measure code comprehension rather than general cognitive skills. However, potential confounding factors such as general reading ability or prior programming experience may still influence performance, which should be addressed in future refinements.

Content Validity A wide range of task types (multiple choice, short essays, flowchart selection, etc.) was included to reflect the multidimensional nature of code comprehension (e.g., abstraction, analytical thinking, error detection).

Criterion Validity Since no external standardized programming test or real-life performance measure was used as a comparison, criterion validity could not yet be established. Future studies could strengthen this aspect by correlating test scores with performance in programming assignments or other validated instruments.

Face Validity To guarantee the criteria of face validity, students were not told what the study was about before its execution. To obtain the perspective of the target group, the students were asked what they think the study was about after participating in it.

4.1.6 Complexity

The complexity of the items was not discussed or determined in advance. Preliminary considerations regarding the complexity of the items were taken into account during the initial design phase, ensuring that the tasks were neither too easy nor excessively difficult. A precise operational definition of item complexity, however, has to be established post-hoc, based on iterative testing and refinement of the tasks.

4.1.7 Feedback

Since this use case is neither an exam itself nor a kind of exam preparation, the result of the test seems rather irrelevant to the students. Nevertheless, the students were given the possibility to find out their results and the solutions to the tasks using a unique and individual code, also ensuring their anonymity. However, this offer was not taken up.

4.1.8 Task Types

The specific task types are discussed in more detail in Chapter 5.2. However, we avoided free response formats, as they are more complex and less clear in their evaluation. Exceptions to this are also discussed and justified in the next chapter.

4.1.9 Contextualization

For the use case, the context was less important, as the participants did not gain any advantages from their answers and the use case did not represent an exam or job interview. We ensured that the tasks fit into the context of the course and did not deviate too much from other tasks set during the course.

4.1.10 Response Format

Two kinds of response formats where offered: a digital version, meant to be answered with a device, such as a notebook or phone (although using a phone is not recommended due to the screen size and inconvenience of answering), and on paper. Only two paper copies were provided, as this option was intended solely for students without notebooks. Both copies were used. That being the case, more copies could be offered than actually expected. The reasoning for the digital version as the main format was the comparatively low evaluation workload, due to not having to digitalize and manually save the answers. It was also initially planned to measure the times students take to answer the tasks. However, this decision proved to make little sense due to the limited time available and was discarded. Additionally, there was the assumption that code on a display is closer to the application and more pleasant to read. For tasks in which code has to be written, this would also be recommended, as writing on paper is not close to the application and could result in extra errors (e.g., syntax errors due to spelling mistakes that are not easily noticeable on paper). However, code did not necessarily have to be written in this test. A digital version be ensured that students were not able to switch between the task. Therefore, possible learning drawn from answering the following questions could not be used to correct former answers (Moosbrugger and Kelava, 2020). The students were informed of this restriction in advance.

4.1.11 Motivation

Basically, it can be assumed that there was little intrinsic motivation on the part of the students. This is due to the fact that the students had no particular incentive or advantage from participating. The test was neither an examination nor an examination preparation. Instead, it was obligatory for students to attend the experiments to finish the course. In addition, there was no financial compensation or anything similar. Participation in the test was based solely on attending the lecture. Accordingly, the aim was to at least create the tasks in such a way that a general reasonableness is given. However, it should be noted that it cannot be guaranteed that all participants have answered to the best of their knowledge and belief.

4.1.12 Additional Materials

The materials offered in the test were limited to code snippets, explanations and images. These are discussed in more detail in Chapter 5. With the exception of assessment tasks (regarding the readability of code), attention was given to make sure the code was straightforward to read and followed basic programming rules. However, the variable names should not reflect the purpose of the functions, as determining this was often part of the task. Also, in cases where the code was used in more than one sub-tasks, students were informed that the code was the same as the one previously presented.

4.1.13 Error Tolerance and Assessment

For most of the tasks (multiple choice with justification), the initial evaluation system only allowed an answer to be rated as "correct" if it was adequately and sufficiently justified. This justification should show why the answer choice is correct or why the options not chosen are incorrect. In order to facilitate and standardize the evaluation of the justifications, possible correct justifications were formulated in advance, which can be used as a guide. However, since the final results with this method were not optimal, the answers were also considered independently of their reasons, which will be discussed in more detail at a later point of this thesis.

For multiple choice questions, only one point was awarded for correct answers, even if several possible options were correct. The students were not told how many options per task were correct in order to avoid random guessing. Accordingly, the evaluation of these tasks was binary (either correct or incorrect). Which makes sense if the evaluation workload is supposed to be kept as low as possible, to guarantee reliability. This may not be the case with other evaluation systems, where the level of comprehension may be more significant. For instance, when utilizing the SOLO-Taxonomy (Bartl, 2024) for evaluation.

For tasks with a free response format, it was specified exactly which components must be included in a correct answer. The results were divided into 0 points (none of the components named), 1 point (some, but not all components named) or 2 points (all specified components named). The subdivision into 0, 1 or 2 points is intended to prevent the answers from allowing too much room for interpretation, which could be interpreted as multiple points.

4.1.14 Objectivity

This thesis functions as a sort of guide. However, not much can be written about the test at this time because it was only conducted once. However, after reviewing the results, the necessary modifications must be made before it can be repeated and remarks given.

4.1.15 Fairness

The language chosen for the test and its execution was English, as the language of the course is also English and the heterogeneity of the group made this the most sensible choice. Additionally, the participants were asked, whether they had any issues with the

test's language, and they responded negatively. In addition, as already mentioned, two different versions were offered - digital and on paper. Initially, however, in the scope of this thesis this test was limited to participants of the "Software Engineering" course. Additional adaptations to the test to make it accessible to the general public could be offered in the future (e.g. audiovisual explanations).

4.1.16 Falsification

Students were instructed not to use any other aids. They were also roughly monitored while answering. However, this did not include monitoring the devices, as this was not possible with the software used. As a result, it was not possible to ensure that aids such as AIs were not used. Although conducting individual tests would be safer at this point, it would not be realistically feasible due to the high cost and the risk of few participants. However, it can possibly be assumed that students cheated less, as answering the test incorrectly did not have any negative effects on them, such as a poor grade.

4.1.17 Reliability

Through aspects such as the clearly described scoring system and the relatively high number of items, we tried to be design the test as reliable as possible. Additionally, the standardized execution further contributes to this: all participants were given the same instructions, tasks were set in a controlled test situation within the lecture, and identical materials were used.

Another factor is the parallelization of the test versions. Although the order of the tasks varied, the structure and content remained the same, which reduced random order effects and ensured comparability between participants.

4.2 Development of Supporting Questionnaire Elements

The construction of the supporting questionnaire elements will be shown in the following. These components complement the performance items in order to provide additional context and ensure a more comprehensive understanding of the test results.

The elements were created at the same time as the performance items. Following a discussion they were specified, simplified and improved in wording.

Within the test you can find two distinct types of questionnaire elements. Placed following a sub-task of the performance item pool type one or "debriefing questions" are intended to identify or uncover the degree to which the response provided is impacted by external or individual factors. Conversely, type two, or "personality questions," are meant to gather some personal information about the test respondents and must be answered prior to any performance items. Their positioning is invariable.

However, the different types are equally subject to certain standards, which are presented below and can also be found in greater detail in Moosbrugger and Kelava, 2020.

Clarity within the expression is a highly relevant aspect. As well phrased without double negatives or nested sentences, since this could lead to confusion. With the renunciation of negative and universal expressions such as “always”, “never” or “all” , since those may lead to a cognitive overload or confusion.

In the case that technical terms are being used, it should be either guaranteed, that all of the students know and understand those. If that is not possible, a synonym or a definition should be provided.

To conclude the used standards, it should be as easy as possible for the students to understand the elements after a single reading. Otherwise it could lead to misinterpretations or the loss of motivation, which could further lead to distorted results.

The response solely relied on self-disclosure about the individual tasks. Subsequently, those subjective details could be distorted in order to cover up possible lack of knowledge. The measured variable should not be mentioned in advance and should be difficult for students to identify in order to reduce that likelihood. The components should also be appropriate for an objective personality assessment. But it is difficult to put the used elements in a generic approach because they are directly tied to the performance items.

Completing the questionnaire elements also restricts the amount of time that can be spent, since they are mandatory and can not be skipped. Students should therefore be informed and encourage to respond as quickly, but also as truthful as possible. To be as specific as feasible, the test’s design should have a small number of significant questionnaire components.

The instructions for the personality questions are included in the overall instructions for the performance test and students are therefore given a short training period.

4.2.1 Personality-Related Questions

At the beginning of the test, after the introductory information but before answering the first performance questions, four personality questions were asked. The questions deal with the participants’ programming experience compared to their fellow students, general experience with logical programming, experience with the Python programming language and, if applicable, other programming languages. The question about experience with other programming languages is an exception in its task type, as this had to be answered with a separate entry in a text field. All other questions were answered by rating them on a scale from “very inexperienced” to “very experienced” with a total of five possible levels. These findings could possibly be used to identify correlations between the participants’ level of experience and their results. In addition, findings could be gathered that with experience in other programming languages, it may still be possible to be successful in Python.

4.2.2 Debriefing Questions

The debriefing questions were placed immediately following each sub-task with the intention of identifying external factors that might have a detrimental impact on the

response. Therefore, the answers of those questions were supposed to filter out potential disruptive factors, such as language, time or medium related problems and separate them from the possible lack of knowledge. The participants are asked whether the task itself is clear and understandable, whether the level of difficulty is appropriate to their experience and abilities and whether lack of time was a problem. External factors, such as the just mentioned harbor the risk of affecting the response of the actual performance item. Moreover, by analyzing the responses of debriefing questions, one could draw conclusions about the quality of the assignment itself. Therefore, they were positioned after every performance item, in order to evaluate its quality and to make sure, the students can still remember the actual task.

Additionally, students were not able to skip those questions or leave statements of it unanswered, to ensure that this information is provided.

At this point, however, it should be noted that preventing skipping can lead to satisficing. An effect which refers to the response behavior of participants who, due to a lack of motivation or fatigue, only complete tasks superficially and without real effort. (Moosbrugger and Kelava, 2020). The anonymity in answering the test and the lack of consequences are intended to counteract the selection of socially undesirable answers. In addition, no neutral answer option was given in order to avoid the tendency towards the middle Moosbrugger and Kelava, 2020. Since the debriefing refer to the personal experience with the task, there is no correct or incorrect answer. However, students were given the option to choose the option "task not started", in the event that they have not started the task and therefore cannot provide any information about the quality of the task or the external factors. This is to prevent the information from being falsified.

Answers to the statements were given in the form of approving or disapproving attitudes in the sense of a high or low expression of the characteristic of interest. Very easy or very difficult statements that are answered in the affirmative or negative by almost everyone should not be selected, although this presents little difficulty when answering ability questions. In addition to “task not started”, the answer selection consisted of a range of “Strongly disagree”, “Disagree”, “Agree” and “Strongly agree”. In order to keep the questions as simple and short as possible, only one statement was given per question.

4.3 Test Assembly in LimeSurvey

The actual construction of the online questionnaire was ultimately carried out using the LimeSurvey platform. This decision was based on several reasons that are particularly relevant for conducting empirical surveys in a university context.

A key advantage of LimeSurvey is the possibility of conducting surveys anonymously - a crucial aspect for sensitive data collection in the context of scientific work. LimeSurvey also provides export functions such as exporting as an Excel sheet which helps preparing the data for further analysis.

Another advantage is the ability to integrate complex question types such as matrix questions, scaling or rankings. Logical jumps and conditions can also be incorporated into

the questionnaire, enabling an individual and user-dependent survey structure.

Many universities offer LimeSurvey on their own servers free of charge and without advertising. This is particularly advantageous for students, as no additional costs or external user accounts are required.

Another feature of the platform is the integrated evaluation functions: Basic statistics and graphical representations can already be viewed during the survey.

In terms of content, the questionnaire was structured in such a way that uniformly structured questions were asked for each task. For better analysis, the tasks were given specific codes consisting of the number of the main task, an "x" as a separator (compatible with LimeSurvey) and the number of the respective sub-task (e.g. 3x2 for sub-task 2 of main task 3). Although LimeSurvey offers the option to randomize the sequence of questions of a survey, this was, unfortunately, not possible for the desired structure of this test, where only certain types of items should be randomized. Since randomization was originally planned to make sure of an approximately equal distribution of answers for each task, different versions of the test were, as already mentioned, offered.

It should be noted, that the visual representation of the tasks in Chapter B Item Pool differs slightly from the visual representation in LimeSurvey. This is due to a better readability within this thesis. However, the content is the same.

Chapter 5

Overview of the Test Items and the Supporting Questionnaire Elements

Chapter 5 provides a detailed overview of the performance items, debriefing questions and personalty-related questions used in the scope of this study. The full list of tasks can be found in the Appendix. Here, it should be noted that due to the length of the appendix, it has been removed from this paper and can be found under Bartl, 2025.

5.1 Personality-Related Questions

Before starting the actual performance part of the test, students had to answer four personality-related questions to support the evaluation and the discovery of possible findings. The specific questions can be found in Chapter A of the appendix.

By gathering personal information, we could possibly make connections between the overall results of the test and the results of those personality-related questions. E.g., a student that shows a better general understanding of programming may also perform better on the test. On the contrary, poor results could be attributed to missing experiences.

5.1.1 Question P1.1

Question P1.1 focuses on the experience and knowledge of the students with programming on a basic, fundamental level. That excludes specific languages or paradigms. To simplify the answering of that question, students are asked to compare their own knowledge with that of their fellow students. That way we could make sure that everyone uses the same standard to rate their experiences.

Answers resulting from this question could additionally be used to make claims regarding the ability to better respond to specific programming languages due to good general programming skills.

5.1.2 Question P1.2

Experiences with the general logical paradigm and, consequently, the ability to identify patterns and principles rather than strictly following what code does are the main topics of **Question P1.2**. Additionally, it integrates ideas, such as logic, data structures, etc.

5.1.3 Question P1.3

Question P1.3 shifts the emphasis to skills in Python, the specific programming language that is used in the test. We can either verify or disprove the assumption that a lot of individuals have already worked with it by requesting this information. Additionally, we could link and compare successful and unsuccessful outcomes with the given skill level.

5.1.4 Question P2

In order to get a more comprehensive picture of their knowledge and experiences, **Question P2** asked them to list any other language they have previously worked with. To reduce the time needed for answering, students were encouraged to only name the three most relevant languages and the number of the remaining. That information might make it easier to connect programming language capabilities. E.g., someone who is proficient in C has less difficulty adapting to Python than someone who is not. To answer that, students were provided with a text field.

5.2 Test Items

After presenting the broad criteria in Chapter 4, the particular performance tasks are provided in greater detail below. A brief overview of all the items, their task type and the skills we intended to test using the task can be found in Table 5.1. Chapter B of the appendix contains the list of items together with their solutions and potential justifications in greater detail. Items are identified by their code. The coding consists of "Fr" for 'Frage' (German for "question"), a number to number the complex task, an x to distinguish the complex task from the sub-task, and another number to assign a number to the sub-task.

Table 5.1: Overview of performance test items

Code	Task Title/Short Description	Task Type	Skills Tested
Fr1x1	Loop & Index Understanding	Multiple Choice	Analytical, Abstraction
Fr1x2	Edge Case Handling	Multiple Choice	Abstraction
Fr1x3	Flowchart Comparison	Dichotomous Task	Analytical, Abstraction
Fr2x1	Detect Output	Multiple Choice	Analytical, Abstraction

continued on next page

Table 5.1 – *continued from previous page*

Code	Task Title/Short Description	Task Type	Skills Tested
Fr2x2	Assigning Inputs to Outputs	Matching Task	Analytical, Abstraction
Fr2x3	Detecting Correct Statements on Behavior	Multiple Choice	Analytical, Abstraction
Fr3x1	Method Naming	Short Essay	Analytical, Abstraction
Fr3x2	Error Detection	Multiple Choice	Analytical, Abstraction, Critical Thinking
Fr4x1	Code Snippet Comparison	Multiple Choice	Analytical, Abstraction, Critical Thinking
Fr5x1	Functionality Check	Multiple Choice	Analytical, Abstraction, Critical Thinking
Fr5x2	Code Evaluation	Short Essay	Critical Thinking
Fr5x3	Code Improvement	Short Essay	Critical Thinking

5.2.1 Fr1x1 - Loop & Index Understanding

Fr1x1 was created to test the participants' capability to abstract and analytical abilities. Students have to follow the code step by step and understand how loops and index checks work. By having students identify patterns, they can determine the result of the program. The task moves from concrete figures to a general abstraction. To not give away any indications about the purpose of the code, we chose neutral variable and function names. As this is part of the answer and could have a positive influence on the result.

The answer options are similar in order to prevent easy elimination. Answer options a) and b), for example, appear similar but differ in the type of return. Anyone who overlooks this detail may not have thoroughly understood the code. Only one answer is correct.

5.2.2 Fr1x2 - Edge Case Handling

By not giving a specific input list, students do not have to calculate the complete sequence but rather simulate a general program flow in thought. It is important to understand that result = [0] * len(data) produces an empty list, both for loops do not execute any iterations, and final_value therefore remains unchanged at 0 and is returned. This approach requires the ability to switch from specific code to a conceptual description of behavior, rather than calculating individual steps with numerical examples. So the item tests if students are capable of understanding the program flow even without visible data processing, which can be an indicator for the ability to abstract.

It should be noted that the behavior of the program with this particular input could be a

language-specific result. Other languages may handle empty inputs differently. Therefore, this task may not be the best example. Students may draw incorrect conclusions based on their experience with other languages in order to cover up gaps in their knowledge.

5.2.3 Fr1x3 - Flowchart Comparison

Along with the code, other additional materials were used for this sub-task. This contains two flowcharts, one representing the correct program and another one with a small error within the code. To make sure that every student knows the purpose of a flowchart, a general explanation was given beforehand.

It is important that the presentation is clear and understandable. Color coding could contribute to better understanding, but it could also be distracting or confusing. Accordingly, it was decided not to use color coding in this case and to opt for a neutral, unobtrusive presentation instead.

This sub-task tests analytical abilities and the ability to abstract at the same time, because it requires additional cognitive transfer to the flowcharts.

Analytical: The student must fully understand the given code — i.e., loops, conditions, and data flow.

Ability to abstract: They have to transfer the textual, line-by-line logic to a visual, symbolized representation. It is not just required to identify the result of code but also whether someone recognizes the internal structure (control flow, decision points, and iteration structure) and correctly recognizes it in another form of representation. Since only one flowchart can represent the correct code, only one answer is correct.

5.2.4 Fr2x1 - Detect Output

Analytical skills are tested because participants have to understand the code step by step (start value of i, loop condition, how and when i is incremented, which list elements are processed). The ability to think abstractly, by contrast, is being tested by having to translate the specific code into generalized logic ("even number $\rightarrow$ B, odd number $\rightarrow$ A, then always C") and then apply this to the given data without getting lost in the details. Only one answer is correct for this task.

5.2.5 Fr2x2 - Assigning Inputs to Outputs

The tested abilities in this task are also analytical skills and the ability to abstract. Analytical skills: the code must be precisely analyzed step-by-step. E.g., recognize that i is incremented at the beginning of the loop (i += 1), which means that processing does not start with the second element but with the third (numbers[2]). You also have to pay attention to the loop condition (i < len(numbers) - 1) to know which indices are actually being checked.

Ability to abstract: students have to translate the code into a general rule and identify that the code starts from the element at index 2, checks the number, whether it is even or odd,

and then makes an output accordingly (B for even, A for odd, then always append C). Then apply this rule to different input lists.

The matching format (assigning inputs to outputs) requires students to simulate this abstract rule several times in mind and apply it to all inputs before comparing the possible answers — without this, error-free assignment is not possible.

It should be noted that a clear presentation of the relevant inputs and outputs can be crucial for understanding, as a confusing presentation could negatively influence the answers without saying anything about the participants' knowledge. Unfortunately, this was not possible with LimeSurvey, as the presentation formats for questions are limited. We recommend a static presentation of the inputs and a dynamic presentation of the outputs, where the participants can add the individual outputs to the inputs using drag and drop. Only one correct order of assignment is correct.

5.2.6 Fr2x3 - Detecting Correct Statements on Behavior

Participants are being tested on analytical skills by having to precisely comprehend how the start of the loop and the termination condition affect the processed elements. It can be seen that not all elements are processed (option a is incorrect) and that the number of "C" characters does not directly correspond to the length of the list (option b is incorrect). Additionally, participants are required to generalize the specific behavior of the loop when evaluating the answer options. Two of the four statements are correct.

5.2.7 Fr3x1 - Method Naming

Task Fr3x1 is a free format task, where students can answer within a short text field. The tasks require students to identify the purpose of a code and give it a suitable name accordingly. Since only a method name is needed, a short text field with a limited number of characters is sufficient. To further mask the purpose, we chose neutral and meaningless variable names. The code used in this task was taken from Izu and Mirolo, 2020. To identify a suitable name, students have to understand what the code does line by line (analytical skills): Two pointers (i at the beginning, j at the end) move towards each other, and values are subtracted at the ends of the list until the values are equal, then the left pointer moves on. After recognizing the behavior, the overall pattern must be recognized from this sequence: The method reduces or equalizes values at both ends of the array by iteratively calculating the difference. From this abstract function description, you can then derive a meaningful method name.

It is essential to examine the evaluation more closely here, as linguistic diversity allows multiple names that convey the same or similar meanings. Accordingly, various suggestions must be offered in advance. However, it is important to be prepared for the possibility that more names may exist, making future discussion necessary. It is important to find various suitable terms, as long as they accurately describe the algorithm. Therefore, instead of one correct answer, a variety can be correct, but only one suffices as an answer.

5.2.8 Fr3x2 - Error Detection

By analyzing the code line by line with all of the given inputs, students have to comprehend how the specific values in v negatively affect the code and show possible issues in the implementation. It can be seen that when input b) ([1, 0, 18]) is entered, the mean value 0 is never changed or checked because the loop only increases i if v[i] == v[j], and otherwise works by subtracting at the ends.

From that precise behavior, participants can derive an abstract rule: The procedure only processes elements at the current end positions, and values in the middle may remain unaffected under certain circumstances. This abstraction then helps to quickly check new inputs to see if they trigger this problem.

For this task, students do not only have to understand the code's behavior but also whether it does what it is intended to. For a) and c), it must be recognized that the processing in terms of the algorithm is complete and error-free, while b) reveals a logical error (the middle element remains 0 and is not "reduced"). Therefore, only one answer is correct.

5.2.9 Fr4x1 - Code Snippet Comparison

Task Fr4x1 uses three different code snippets that are similar in what they are doing but differ in small, subtle details. Variables as well as functions have the same names in all three snippets. For analytical skills, students must comprehend those snippets line by line and identify which snippets fit the given specification.

In snippet 1, you can see that negative values in the original array are replaced directly by 0. In snippet 2, you can see that a new list is created in which negative values are replaced by 0 and all others remain unchanged. In snippet 3, it is noticeable that only positive numbers are changed (they are increased by 1), which contradicts the specification.

Students can also show their ability to abstract by deriving the underlying functionality from the specific lines: Snippet 1 and Snippet 2 follow the general rule "negative $\rightarrow$ 0, everything else unchanged." Snippet 3 follows a completely different rule. Therefore, also finding snippet 3 to be not suitable.

Along with analytical skills and the ability to abstract, this task also tests critical thinking by evaluating the implemented logic and comparing it to a given specification. It is important to check not only whether the code "works," but also whether it solves the required task exactly. Since two of the three given snippets fit the specification, the answer is only correct by choosing both.

5.2.10 Fr5x1 - Functionality Check

Analytical skills are being tested since the participant must analyze the Python code line by line, understand what each statement does, and check whether this matches the task description. Ability to abstract is tested by having to abstract the specific code details (loops, variables, conditions) into a general functional principle ("calculates the largest difference between consecutive numbers?"). Also, critical thinking is tested, because students must not be misled by the form of the code but must check whether the logic

actually achieves the desired goal or contains errors (e.g., incorrect handling of negative differences). Students can therefore choose between two options, either confirming or denying that the code does what is said in the task. Only one answer is correct, as one excludes the other.

5.2.11 Fr5x2 - Code Evaluation

Unlike the other tasks, **task Fr5x2** and the following **Fr5x3** did not require the students to additionally justify their answer in a few words. Instead those tasks require students to answer with a short essay in a text field. It is only testing critical thinking by concentrating on the quality of the code. Students need to evaluate its efficiency and readability; that requires critical evaluation, because they do not just execute, they have to assess and identify issues:

Inefficient sections (e.g., unnecessary loop in line 7, unnecessary multiplication in line 8), poor readability (e.g., inappropriate variable names such as m), and logical errors (incorrect handling of negative differences). Optionally, the lack of descriptive comments can also be listed as a problem.

The assessment is gradual (partial credit scoring) and not binary as in the before-discussed task because the answer is assessed on the basis of several predefined criteria. A partial credit scoring refers to a system that classifies answers not only as correct or incorrect (zero or one point) but rather depending on the quality and completeness of the answer. This allows for a more nuanced representation of differences in performance (Masters, 1982). A maximum of two points can be achieved to show the differences between incorrect answers, correct but incomplete answers, and complete answers. Accordingly, the assessment criteria must be clearly described and recorded in advance.

5.2.12 Fr5x3 - Code Improvement

This task follows on from **Fr5x2** and requires the problems already identified in the code to be corrected. **Task Fr5x3** also requires students to provide a short answer in the form of a text field. Critical thinking is tested by focusing on the quality of the code. The assessment scheme is the same as in **task Fr5x2**. Critical thinking is required because the test participant must formulate suggestions for improvement themselves – not just find errors but propose concrete optimizations. Meaningful changes must be derived from the previously identified problems: finding meaningful variable and function names, optimizing its structure by eliminating unnecessary steps, correcting the logical error, and possibly adding useful comments. This requires analyzing the existing code, abstracting it, and proposing targeted changes based on good programming practice. Even if analytical skills and abstract thinking are not the main focus here.

5.3 Debriefing Questions

In the pages that follow, the consecutive debriefing questions will be described in more detail. You can find an overview of those items in Chapter C of the appendix. It is

important to mention, that the coding of the questions utilized in this chapter as well as in the overview, differ from the coding used in Chapter D of the appendix. The questions are presented in general terms here. Therefore, it is sufficient to use a code that consists of a number to number the questions and a "D" for debriefing. Chapter D, by contrast, presents the debriefing questions corresponding to their performance items. Consequently, a distinction between the individual items with their debriefing results is needed.

5.3.1 Question D1

The purpose of this question was to rule out the possibility of having an overly complicated task and to identify tasks that are confusing and therefore more difficult to answer. A task that cannot do this distorts the results, as it does not reveal a lack of knowledge of the question but rather the incomprehensibility of the task.

5.3.2 Question D2

This question is intended to reveal whether the skills being asked about are outside the scope of those already learned, in which case the answers will not reveal a lack of understanding but rather a lack of resources.

5.3.3 Question D3

Since the time for this test was generally limited, evaluating this debriefing question is somewhat more difficult. However, its main purpose is to ensure, especially when answering the last questions, that poorer results in the corresponding tasks are due to a lack of time and not simply or exclusively to a lack of knowledge.

5.3.4 Question D4

Question 4 refers only to students who did not answer the test on paper but in digital form using a smartphone, tablet, or laptop, etc., since technical problems can constantly arise with these devices and therefore, ultimately distort the answers. Technical problems here refer to problems unrelated to the test or the participants, such as insufficient battery power, poor connections, long loading times, display errors, etc., which also distort the results. Students could first choose either yes or no for a single-choice question. If they chose yes, a free text field opened in which they could specify the type of problem.

Chapter 6

Empirical Study and Evaluation

In Chapter 6 we first describe the test execution to clarify its methodological framework. Subsequently, any feedback received from the students after taking part in the test will be summarized and presented. In the last section, we present the results both in a general form and broken down by individual tasks. A tabular overview of the results can be found at the end of this chapter in table 6.1. The presentation is purely descriptive, without any further interpretation, in order to create a neutral base for the subsequent analysis in Chapter 7.

6.1 Test Execution and Data Collection

The test was conducted during a lecture, following an initial independent study. Beforehand, students were informed that another study would take place; more details were not given. It has to be noted that the first study could have had a negative impact on the concentration of the students.

During the test, the measured characteristic was not explained at first, as students were supposed to guess and discuss what it could be during a follow-up discussion. This could provide insights into the design and the validity of the tasks and will be discussed in Chapter 6.2.

At the beginning, students received relevant instructions — verbally, in writing within the test, and additionally on the board in front of them. The full written test instructions can be found in Bartl, 2025.

- Information about the type of answers required, preferably with an example, permitted (only pen and paper) and non-permitted aids such as AI. Even though these were not permitted we could not properly ensure that students did not use them. Conversations and working in groups were also not allowed.
- The length and time of the test were also explained. We additionally noted that they should not stress, as the aim was not to answer all the questions in full.
- Reference to anonymity when answering questions.

- The most important information was that answers only count as correct when also correctly justified. The importance of the justification was made clear.

After ensuring that every student understood the given instructions and the paper forms were distributed, the 30-minute working time began. During this time, students were able to ask questions. At the end, they were given an additional 2-3 minutes to finish the started task and leave any general comments.

By entering some personal information at the end, students were able to generate a unique code that they could use to find out their results.

6.2 Analysis of Participant Responses

Initial feedback was provided immediately after the test was completed, guided by questions from the test administrators. We asked the participants the following questions:

1. Did you have any struggle with the programming language Python?
2. Did the language of the test (English) pose any problems for you?
3. What do you think the study was about?

The majority of participants gave affirmative answers to the **first question**, which contradicts the assumption that Python is a suitable language, even if not all participants have already worked with it. This issue will be addressed again later in the feedback. In contrast, **question 2** was answered in the negative by all participants, thus ruling out the possibility that a lack of English language skills could have led to self-censorship in the tasks and thus to poorer results. **Question 3** was asked to make statements about the extent to which the test meets the criterion of face validity. The following answers were given by students: "Code understanding." "How we evaluate ourselves?" and "Code snippet performance in exams." While the second answer does not really meet the measured variable, the first and also the last answer tie into the measured construct. "Code understanding." shows that non-experts (represented by the student) are able to immediately recognize the underlying construct of the test, which indicates that the task appears meaningful and relevant at first glance and thus supports the face validity of the instrument. While "Code snippet performance in exams." does not directly support face validity, it is still an important aspect of this study and therefore not completely wrong or irrelevant and to some extent supportive.

A second round of feedback was conducted one week after the test took place. Again, the students were asked specific questions to further evaluate the test:

4. Did you have any problems understanding the question "Did you experience any technical difficulties while completing the task?"?
5. Why were no or only justifications given? Was that an active decision?
6. What would have helped answering the test (e.g., longer explanations in the beginning)? Was it too much information overall?

For **question 4**, students mentioned that the debriefing questions after every task disrupted the workflow and were seen as "annoying". While the constant questioning after every task does disrupt the working and takes a lot of time, it was necessary within this test to evaluate the quality of the individual tasks. Also, one of the problems regarding Question D4 was the fact that not everyone understood the term "technical problems" the same way and as it was intended. While the intention was to rule out any hardware, software, or network issues, a large number of students interpreted the term differently (e.g., "technical" in the sense of the syntax of the code). So the phrasing of question D4 was overall too ambiguous and should have been specified further.

Question 5: Students explained that their primary focus was on completing the tasks and actually understanding the code, which is why they often did not provide explanations. Why some students provided explanations but no answers was not addressed. Therefore, only the assumption could be made that they solely forgot to answer.

Question 6: Along with the visual example that was given in the beginning, a warm-up task would have been helpful to fully comprehend the requirements of the tasks and debriefing questions and thus, rule out any confusions.

After discussing the guided questions, students were able to provide further comments and remarks: Even though every task that reuses already shown code had a note stating that, students pointed out that a verbal note in the beginning would have helped further. They justified this with the high amount of information given with every task, which resulted in overlooking some notes and therefore spending time at identifying the code again.

Students also criticized being unable to return to former questions. Even though this is understandable from the participants' point of view, the intention was to avoid any transfer effects where participants could learn from the tasks before and possibly distort the results (Mahdavi and Abedjan, 2020).

The overall structure made answering the test "unpleasant", since every sub-task was presented on its own and not as one big complex task. While this could be done in the future (after evaluating the tasks), it was intended for this test to ensure the assessment of every single sub-task.

Again, Python as the utilized language proposed the biggest problem within this feedback session. Students claimed to have used ChatGPT. Not to answer the tasks, but to explain the code itself, since the syntax of Python was problematic. Instead, they suggested the use of pseudo-code as an addition or other programming languages to choose from. Students claimed that switching between programming languages is challenging for novices and may be simpler for professionals, defying the presumption that Python was simpler for novices or programmers working with other languages. Despite the note that there is no reason to stress or hurry, as the tests' aim was not to be completed, students felt rushed and in a situation similar to an exam. Also, answering the test using a smartphone was not recommended by students who did so, as the limited screen size did not show the whole task or code at once.

In addition, students had the opportunity to provide feedback and evaluations on the

test by means of a general comment at the end. A general overview of these comments can be found in the appendix in Chapter D in Diagram D.3 Overall Comments. Eight people provided comments on one or more topics. The comments could be divided into three different groups: problems with the limited time, problems with Python being the programming language in the test, and a note about an incorrect task. Although students were informed that the aim was not to complete the entire test within the time limit, five students mentioned the limited time. Four people mentioned problems with the programming language, especially with the syntax of the language. Accordingly, it was recommended that participants either be informed about this in advance or that participants should have at least a basic level of knowledge of Python. However, one student responded during a feedback session that he would not have attended if the programming language for the test had been known in advance. Another comment referred to the option of offering several programming languages to choose from, which unfortunately was not possible within the scope of this thesis. One participant pointed out a possible error in job **Fr2x1**, but the error could not be confirmed.

6.3 Results

6.3.1 Method for Assessment under Optimal Circumstances

Definition of Optimal Circumstances

The ideal case in the scope of this thesis is defined as a case in which data collection and analysis take place under ideal conditions. This means that all attending students clearly and completely understand the given tasks and the subsequent debriefing questions. They also interpret the technical terms used in their intended sense and give their answers spontaneously and truthfully. Furthermore, an optimal case requires students to answer honestly and without any unapproved aids, such as Large Language Models. Also, all students should have a similar and comparable level of knowledge and preferably have some kind of experience in the programming language used in the test.

In the best-case scenario, the obligation to provide reasons for the tasks is also fully met. Each answer should be accompanied by a reason that clearly explains why a particular decision is made. It is not essential for every reason to be technically correct. What is more important is that the student's thought process is clear. Statements such as "I don't know" for example, do not meet this requirement. Ideally, all answers should contain such a justification; it would also be acceptable if the absence of a justification is only a rare exception. Therefore, in an ideal case, points are only given for a correct answer and an adequate justification for it.

Finally, the quality of the answers to debriefing questions is guaranteed: They reflect the students' actual experiences with the tasks and do not lead to contradictory information. A negative example of this would be if a person states that they have not started a task but at the same time gives a corresponding rating in another category.

Ideal Procedure for Evaluation

The evaluation is carried out in several steps. First, all answers are transferred and classified as correct or incorrect according to the respective evaluation scheme. On this basis, points can be awarded and an initial overview of the results can be created. In addition, diagrams are created for the debriefing questions.

When calculating average values for debriefing questions and their diagrams, an adjusted average should be taken into account: This includes ignoring tasks that have been proven not to have been started (e.g., through corresponding information in the debriefings) and those for which the processing time and other information clearly indicate that they have not been processed (e.g., comments such as "no time" in the Technical Problems section, or a very short time used to answer).

The actual task solutions are then analyzed together with their justifications. Both response rates and the distribution of results are recorded. Points are awarded exclusively for correct answers (and their justification). Originally, it was planned that points would only be awarded with adequate justifications. Due to the insufficient number of justifications, this evaluation was expanded to include only correct answers. Multiple answers were possible within the answer distributions, as the number of expected correct solutions was not clearly defined in all cases. After that, the justifications are analyzed further, and any visible approaches, thinking processes and strategies are derived and interpreted.

Personality-Related Questions: Two sub-areas are evaluated as part of the personality part:

P1 (P1.1, P1.2, P1.3): Here, the numbers indicate the level of experience. The scale ranges from 1 (very inexperienced) to 5 (very experienced), with 3 representing the average and 2 and 4 representing intermediate levels.

P2: Here, the additional programming languages mentioned were simply listed.

The general overview of those questions can be found in appendix Chapter A Personality-Related Questions and their evaluation in the appendix in Chapter D Figure D.1.

Justification Analysis: A central component of the evaluation is to record the rate of justifications and correct justifications. First, a definition was established to distinguish between good and poor reasoning (see appendix B Item Pool). A reason could be correct in terms of content but still not sufficiently support the answer choice. In this case, it is still considered insufficient for awarding points. Justifications consisting only of a line or a part of code are also not sufficient, since they do not show any thinking processes of the participants.

According to the original assessment scheme, in the ideal case, a point is only awarded if both the answer was correct and a comprehensible explanation was provided. In order to record this in a differentiated manner, the explanations are categorized, for example, as correct, incorrect, or incomplete but generally correct. Justification can therefore be directly compared to each other (e.g., comparing a correct and sufficient justification to an incomplete justification). A separate summary is then created for each task, on the basis of the total number of points and justifications.

Time measurement: Processing times are also recorded and adjusted. Times for tasks that are proven not to have been started are excluded. In principle, processing times could provide valuable insights into solution strategies and difficulties. In the present case, however, the significance is limited because the overall processing time was limited and the tasks were presented in different orders. As a result, average values may be distorted by tasks at the end of the processing time. The processing time for the paper formats could not be recorded. Overall, time measurement makes sense above all when it is not limited and participants can work on the tasks freely according to their individual needs. In this case, however, it can be used to check the plausibility of the information provided in the debriefing questions.

Reflection on Abilities to be tested: Another important step is examining, whether the students have actually been properly tested on the intended abilities through the tasks. Therefore, identified thought patterns and procedures are being compared to the task and its requirements.

The analysis focuses on what insights can be derived from the given answers and the corresponding justifications, which mental models were reflected in them and which difficulties are evident in the individual tasks.

Assessment of the Debriefing Questions: Finally, the answers of the debriefing questions are being analyzed. Again, data cleansing beforehand was needed and therefore, tasks that were not started were not included in the calculation of average scores. The evaluation itself was based on appendix C using the following scale:

0 – task not started

1 – strongly disagree

2 – disagree

3 – agree

4 – strongly agree

The unanalyzed list of the results of the tasks and the debriefing questions can be found in Chapter 6.3. However, the final analysis and evaluation of the findings can be found in Chapter 7, divided into the sub-tasks.

6.3.2 Personality-Related Questions

Students were asked to rate their programming experiences in three different dimensions: general programming experience, experience with the logical programming paradigm as well as the experience with the programming language Python. The results can be found in Chapter D1 of appendix D.

General Programming Experience (P1.1):

The majority of participants ranked in the middle range (21 people). Eight students rated their experiences as below that level, while five of them stated their own level to be "quite

experienced". One person each stated to be either very inexperienced or very experienced. The mean self-assessment score was 2.91.

Experience with the Logical Programming Paradigm (P1.2):

Also, the main focus lies in the center with 20 students. Twelve students rated themselves below that value and three above. One person stated they were very inexperienced, while none of them rated themselves as very experienced. The average value was 2.69.

Experience with Python (P1.3):

Regarding the experiences with Python, 14 participants stated they have an average skill level. Eleven rated their experiences just below that, while six students rated theirs above as "quite experienced". Five students stated they have had no experiences with Python. Again, no one claimed to be very experienced with that programming language. The average value was 2.58.

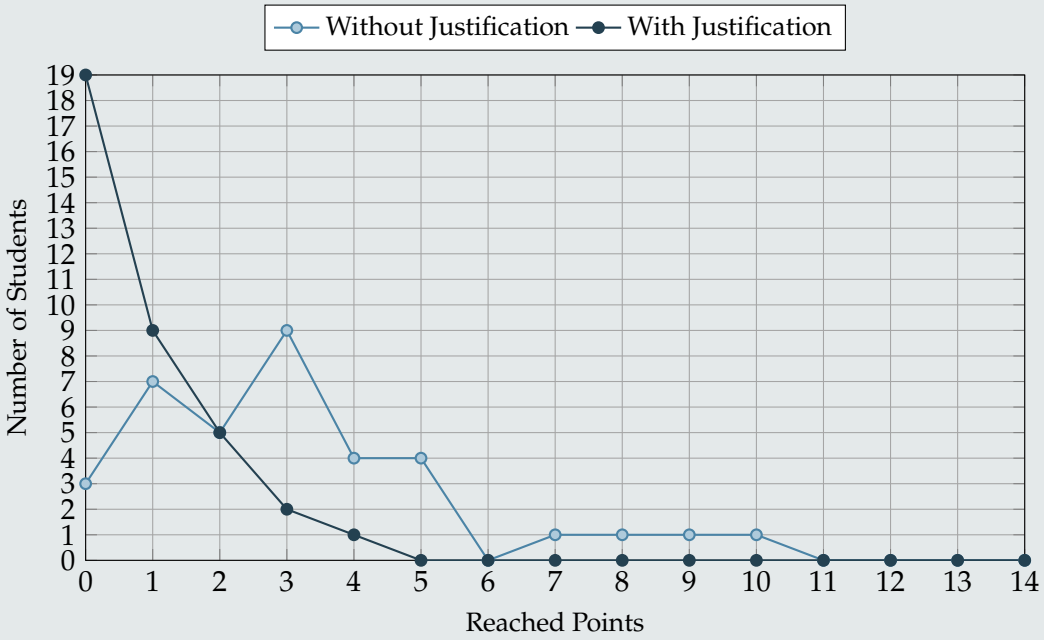
To conclude those insights, it can be said that the vast majority of students rated themselves as mediocre in all three dimensions. Differences can be seen in the fact that the general programming experiences were, on average, higher rated than experiences in specific areas, such as Python or the logical paradigm.

Question P2 shows all of the answers regarding the known programming languages besides Python. Even though the question specifically asked for languages besides Python, students still named it regardless. Other than that, the most used languages were C, Java, C++, JavaScript and more, as the figure shows. Languages, such as Angular, SQL, embedded C, R or Apex proved to be an exception. Only two students specified a number, standing for the number of other languages they are experienced in.

6.3.3 General Results

A maximum of 14 points could be achieved. Originally, it was planned that points would only be awarded if a correct justification was provided. However, as too few justifications were given and even fewer correct ones, a distinction was made in the results: points achieved with correct justifications and points achieved without correct justifications. The results of this can be found in D.4 General Results. No one achieved the full number of points, either with or without justification, which was to be expected given the limited time available. For both options, zero was the lowest number of points achieved, with the number of students achieving this score being 19 without a justification and only three with a justification. The highest score achieved with correct explanations is four, by one student. If the explanations are not taken into account, the highest score achieved is ten, also by one person. However, it should be noted that this is not the same person. Instead, the person with four correct justifications also achieved a total score of four without taking the justification into account. This means that this person was able to correctly justify all of their correct answers. This only happened twice more, once with a total of one point and another time with a total of two points.

D.4: General Results



There were ultimately 36 test submissions, with the difference between this number and the number of students (35) being due to one test being abandoned and restarted due to technical problems. However, this was not mentioned in the question about technical problems, only in the feedback afterwards.

In total 420 correct answers could have been given by all of the attending students. Of these, 111 were completely correct.

Fr1x3 with 20 correct answers (regardless of reasons) has the highest number of correct answers. The lowest values, each with one correct answer, are **Fr2x3** and Fr3x1. A more detailed analysis can be found in Chapter 6.3.3 to 6.3.15.

Justifications were requested for ten of the twelve sub-tasks, which means a maximum of 350 justifications. However, only 139 justifications were given in total. Of these, 120 were meaningful in the sense that conclusions about the way of thinking could be drawn from an incorrect or correct justification. Incorrect explanations also included incomplete or overly vague explanations.

Justifications that were not meaningful were characterized by statements such as "no idea," explanations that were too short, and explanations that were difficult to understand or incomprehensible. Correct (but sometimes incomplete) explanations were limited to 39.

Fr2x1 was the task with the highest number of justifications given. Even though only three of them were correct. **Fr2x3** by contrast, had the least amount of justifications (eight), with also only three of them being correct. Most tasks had a percentage of correct answers ranging from 10 to 30%. Only two sub-tasks had percentages above this range. These were **Fr1x2** (43.75%) and **Fr1x3** (61.54%). One task was below this range, with 0% for task **Fr3x1**.

Table 6.1 shows an overview of all of the tasks (correct) answers and number of (correct) justifications. A detailed analysis of correct/incorrect justifications for each task can be found in Chapter 7.1. In addition, the response time for each task was recorded. However, the needed times of students using the paper version of this test could not be recorded.

6.3.4 Fr1x1 - Loop & Index Understanding

The processing times for the task varied greatly: the shortest time needed (by students that probably did not start the task or solely read it) was 9.97 s, while the longest time used was 541.83 s. The average time for all participants was 149.93 s. (5,997.48 s in total, spread across 33 people). After adjusting the data — excluding individuals who had not started the task—the average was 229.06 s (5726.41 s over 25 individuals). A total of three individuals reported not having started the task.

18 students agreed that the phrasing of the task was clear and unambiguous, while ten people strongly agreed to that. The average score for the corresponding debriefing question was 3.16. Only four students (strongly) disagreed with that statement.

Regarding the difficulty of the task, 21 students agreed that their skill level matched the required skills and knowledge; three students strongly agreed. The average value is 2.78.

Also, 18 students stated that the given time was sufficient to complete the task, while eight students strongly agreed to that. Only six people either disagreed or strongly disagreed. The average for that statement is 2.97.

In terms of technical problems, the picture was very positive: 27 participants stated they did not experience any technical difficulties (according to their own interpretation of the term). One of the students reported a technical problem while answering the test but did not specify or mention what kind of problem occurred. Seven other people also stated they had experienced a problem. However, this did not correspond to the intended definition of “technical problems.” Instead, they named "trouble with forming a mental simulation of the code" (one person), "understanding the code" (two people), "problems with technical information" (one person; "technical information" was not further specified), "Python as a programming language" (one person), and two of them had trouble with the limited time.

In total 30 answers were given, with eleven of them being correct. The distribution of given answers was as follows: Answer a) was chosen 18 times, answer b) eight times, and answer c) twelve times.

Of 18 justifications, six explanations were deemed correct and sufficient, which adequately justified the choice of answer. Four of these explanations belonged to correct answers, while two belonged to an incorrect answer. Eleven justifications were either incorrect or incomplete.

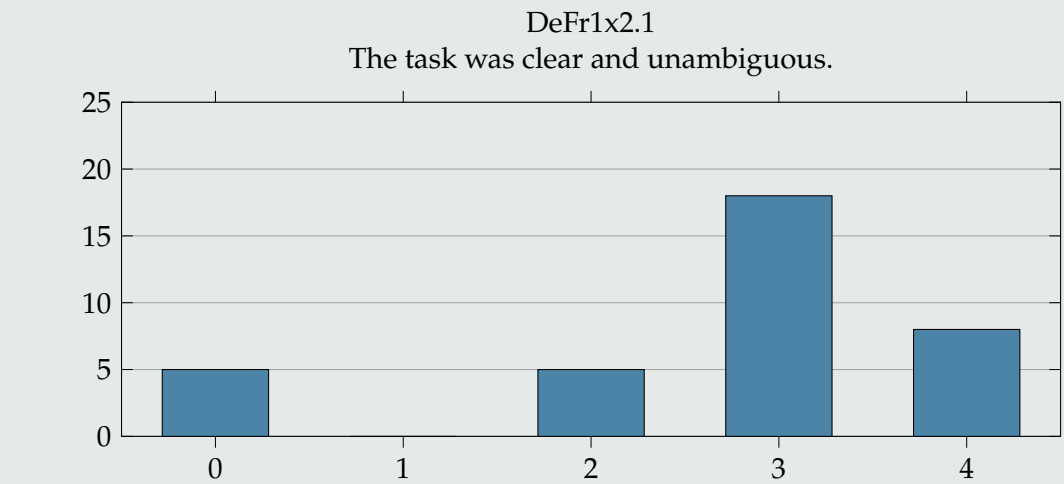
6.3.5 Fr1x2 - Edge Case Handling

Task Fr1x2 also showed a wide range of completion times: the shortest recorded time, for students who likely did not start answering the task, was 2.97 s. The longest, on the contrary, was 531.78 s. On average, students needed 127.81 s (4345.41 s in total) and 159.64 s

(4310.29 s and 27 people) after data cleansing. Four to five students stated to not have started the task. The difference is justified in one student stating to not have started the task in one debriefing question, but not the others.

18 students agreed that the phrasing of the task was clear and unambiguous, while eight people strongly agreed to that. Only five students disagreed with that statement. Therefore, reaching an average score for the corresponding debriefing question of 3.1.

Fr1x2



17 students agreed that their skill level matched the required skills; additionally, six people strongly agreed, seven disagreed, and only two strongly disagreed. The average here was 3.03.

With regard to the time available, 15 people agreed that it was sufficient, while seven people strongly agreed. However, eight disagreed and two strongly disagreed. The mean value was 2.84.

Regarding any technical problems 28 participants did not state to have experienced any problems (according to their own interpretation of the term). Two people reported having had a problem without specifying what it was. Five others mentioned difficulties that did not correspond to the intended definition. Problems mentioned included understanding the code (one person), understanding the task (one person), understanding the selection options (one person), and the time available (two people).

In total 27 students answered the task, with 14 of them being correct. The distribution of the selected answer options showed the following picture: Answer a) was selected eleven times, answer b) 15 times, and answer c) three times. Of 27 answers, only 18 of them were justified. Twelve justifications were considered correct and therefore sufficiently and adequately justified the chosen answer.

6.3.6 Fr1x3 - Flowchart Comparison

The time used to answer task **Fr1x3** ranged between 2.92 s (for participants that did not start the task or may only have read it) and 372.98 s. Therefore, the average for all participants was 73.74 s (2507.27 s in total). After excluding students that did not start the task, the average was 93.30 s (2519.03 s for 27 students). In total, three students specifically stated to not have started the task.

The task was considered clearly and unambiguously phrased by 26 students (18 agreed and eight strongly agreed). With five people disagreeing and one strongly disagreeing, the average reached 3.03.

14 students agreed and eight strongly agreed that the required skill level matched their own. Meanwhile, eight people disagreed, and two even strongly disagreed with that statement. The average value for that is 2.88.

With regard to the time available, 14 people agreed that it was sufficient, while eight strongly agreed. Three disagreed completely, and seven disagreed. The mean value was 2.84.

According to their own interpretation of the term, 32 did not confirm to have any technical problems. Two people reported having had a problem without specifying what it was. Two others mentioned difficulties that did not correspond to the intended definition. Both cited time limits.

26 students answered **task Fr1x3**. 20 of them chose the correct flowchart, and even ten justified their answer correctly. Three justifications were either incorrect or insufficient and could therefore not justify their decision adequately.

6.3.7 Fr2x1 - Detect Output

The processing times for **task Fr2x1** varied greatly. The shortest recorded time (for people who had not started the task or had only read it) was 2.83 s, while the longest processing time was 624.53 s. The average for all 34 participants was 189.50 s (6443.1 s in total). After adjusting the data —excluding people who had not started the task —the average was 200.76 s (6424.42 s across 32 people). A total of two people stated that they had not started the task.

Twenty-four people agreed that the task was clear and unambiguous, while seven strongly agreed with this statement. Two disagreed and one strongly disagreed. The mean value of the corresponding debriefing question was 3.09.

With regard to the level of difficulty, 16 people stated that it matched their abilities and level of knowledge, and eight strongly agreed. No one disagreed strongly with the statement, and ten disagreed. The average here was 2.94.

Regarding the time available, 16 people also agreed that it was sufficient, while eleven strongly agreed, four disagreed, and three strongly disagreed. The average was 3.03.

With regard to technical problems, 32 students reported that they had not encountered

any difficulties (according to their own interpretation of the term). One person stated that they had encountered a problem without specifying what it was. Three others mentioned difficulties that did not correspond to the intended definition. These were time problems (one person), difficulties with the programming language (one person), and problems related to their own level of knowledge (one person).

A total of 31 responses were submitted, seven of them being correct. Seven corresponded to answer a), 14 to answer b), and ten to answer c). 18 of the responses included a justification. Seven of these justifications were assessed as correct and sufficient; however, four of them led to an incorrect answer.

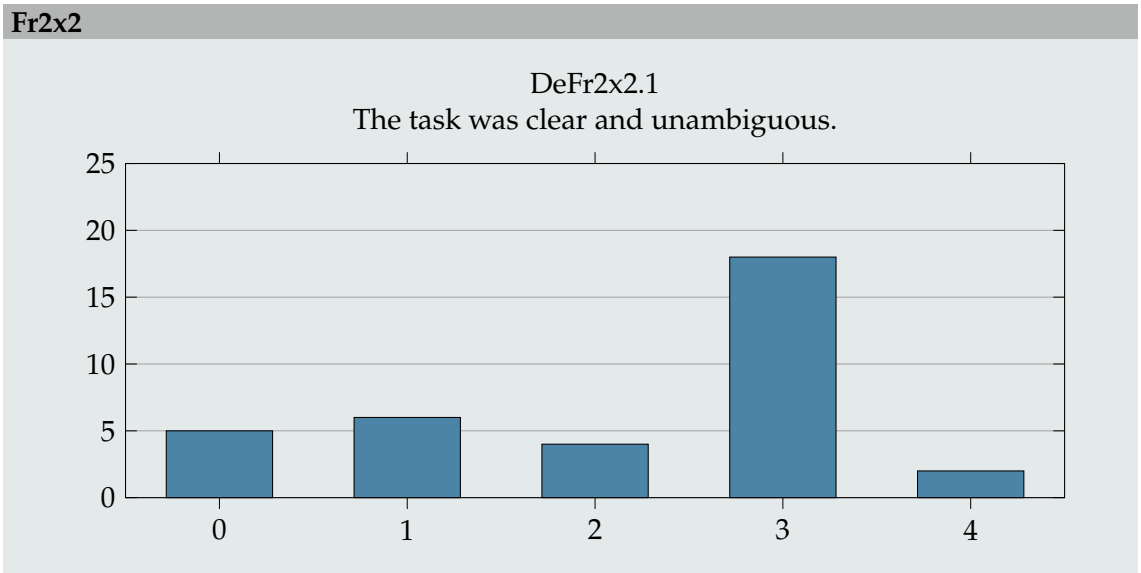
6.3.8 Fr2x2 - Assigning Inputs to Outputs

The processing times for **task Fr2x2** ranged from 3.08 s (for people who did not start the task or only read it) to 611.75 s. The average for all 34 participants was 138.49 s (4708.63 s in total). After adjusting the data — excluding responses that were highly unlikely to have been started — the average was 167.12 s (4678.98 s across 28 people). A total of five people stated that they had not started the task.

Of the remaining participants, 18 agreed that the task was clear and unambiguous, while two strongly agreed, four disagreed, and six strongly disagreed. The mean value of the corresponding debriefing question was 2.53.

Regarding the level of difficulty, 14 people stated that it corresponded to their knowledge and abilities; two people strongly agreed. However, twelve disagreed and two strongly disagreed. The average here was 2.45.

Fr2x2

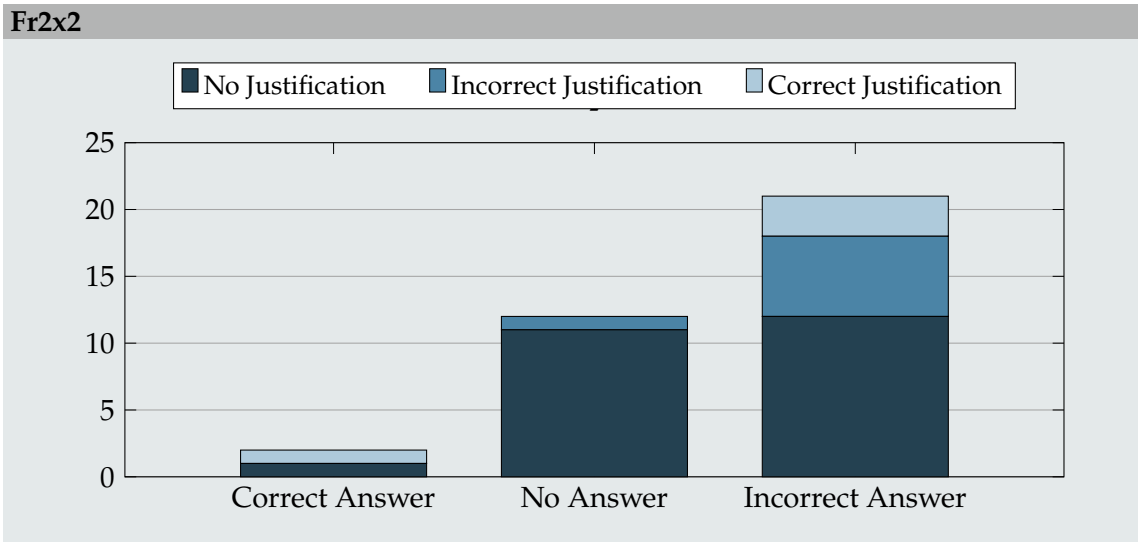


With regard to the time available, 14 people also agreed that it was sufficient, while two strongly agreed, eight disagreed, and six strongly disagreed. The mean value was 2.32.

With regard to technical problems, 31 students reported that they had not encountered any difficulties (according to their own interpretation of the term). One person stated that

they had encountered a problem without specifying what it was. Three others mentioned difficulties that did not correspond to the intended definition. These were problems understanding the task (two people) and difficulties with the programming language (one person).

A total of 23 answers were submitted. Only one of these was in the correct order. Eleven of the answers included a justification. Four of these justifications were considered correct and sufficient; however, three of them still led to an incorrect answer.



6.3.9 Fr2x3 - Detecting Correct Statements on Behavior

The processing times for **task Fr2x3** ranged from 2.55 s (for people who did not start the task or only read it) to 198.02 s. The average for all participants was 50.53 s (1718.18 s in total). After adjusting the data—excluding people who had not started the task — the average was 62.40 s (1684.86 s for 27 people). A total of five people stated that they had not started the task.

Of the remaining participants, 20 agreed that the task was clear and unambiguous, while four strongly agreed with this statement. Four disagreed and two strongly disagreed. The mean value of the corresponding debriefing question was 2.87.

Regarding the level of difficulty, 21 people stated that it matched their skills and knowledge level; two people strongly agreed. Five people disagreed and two strongly disagreed. The average here was 2.77.

Regarding the time available, 16 people agreed that it was sufficient, while four strongly agreed. On contrary, four disagreed and six strongly disagreed. The mean value was 2.6.

With regard to technical problems, 31 students reported that they had not encountered any difficulties (according to their own interpretation of the term). Four others stated that they had encountered a problem without specifying what it was.

A total of 22 responses were submitted. The distribution of the selected answer options

was as follows: Answer a) was selected six times, answer b) eleven times, answer c) twelve times, and answer d) three times. Only two people correctly selected c) and d). Of the 22 total and two correct responses, eight included a justification. One of these justifications was deemed correct and sufficient and adequately justified the choice of answer.

6.3.10 Fr3x1 - Method Naming

Task Fr3x1 showed very different processing times. The shortest time – for people who had not started the task or had only looked at it briefly – was 2.3 s, while the longest processing time was 767.77 s. The overall average for all participants was 186.88 s (6353.82 s in total). Looking only at the participants who actually completed the task, the average was 203.08 s (5686.28 s across 28 people). Five people stated that they had not started the task.

The assessments of the task were predominantly positive: 15 people agreed that the task was clear and unambiguous, and four people even strongly agreed with this statement. Eight disagreed and one person strongly disagreed. The mean value of the corresponding debriefing question was 2.79.

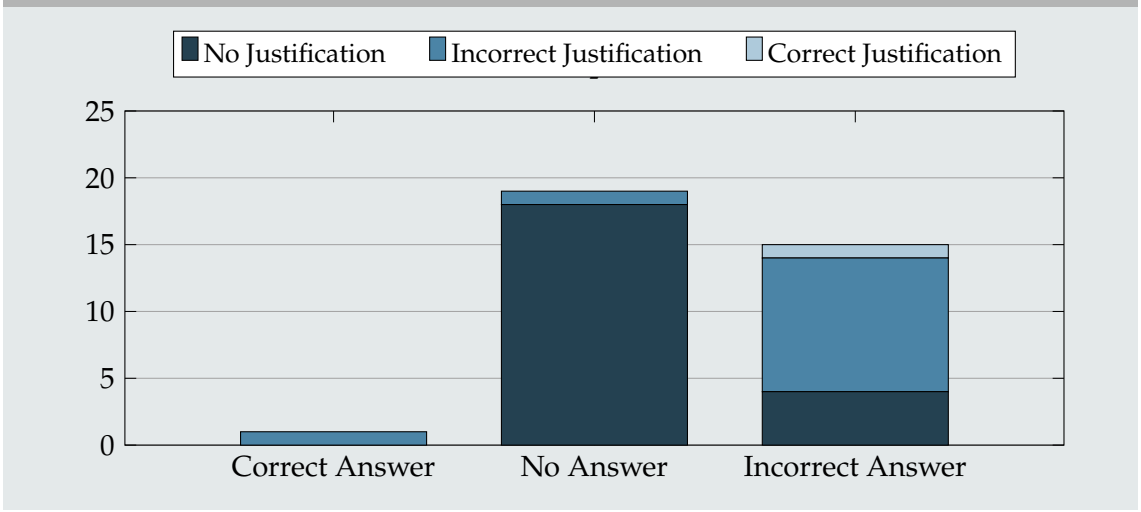
Regarding the level of difficulty, 16 people stated that it corresponded to their knowledge and abilities, while two strongly agreed. Eight disagreed and two strongly disagreed. The mean value here was 2.64.

With regard to the available processing time, 16 people also agreed that it was sufficient; two people strongly agreed. However, seven disagreed and three strongly disagreed. The mean value was 2.61.

Most participants stated that they had not experienced any technical problems: 29 students reported that they had not encountered any difficulties (according to their own definition of the term). One person reported a problem but did not specify what it was. Three others reported difficulties, but these did not correspond to the intended definition of technical problems. These included Python as a programming language (one person), ambiguities in the task description (one person), and problems with time (one person).

A total of 16 answers were submitted. One of these answers was almost correct (balance_values), although the underlying code for this type of task was apparently too complex. Only 13 answers contained a justification, and only one was considered correct, even though it did not justify the correct answer.

Fr3x1



6.3.11 Fr3x2 - Error Detection

For **task Fr3x2**, processing times ranged from 2.3 s (for participants who did not start the task or only looked at it briefly) to 500.71 s. The overall average processing time for all participants was 109.68 s (3729.26 s in total). If only the group that actually worked on the task is taken into account, the average is 141.72 s (3684.71 s for 26 people). Four people stated that they had not started the task.

The assessment of comprehensibility was predominantly positive: 19 participants agreed that the task was clear and unambiguous, five people strongly agreed with this statement, and five disagreed. The average score for the debriefing question was 3.0.

Regarding the level of difficulty, 16 people stated that it corresponded to their knowledge and abilities; four people strongly agreed, eight disagreed, and one person disagreed completely. The average here was 2.79.

With regard to the time available, 14 people agreed that it was sufficient, while six strongly agreed. Six people disagreed and three strongly disagreed. Here, too, the average score was 2.79.

The majority of participants reported no technical problems: 30 students stated that they had not encountered any difficulties (according to their own understanding of the term). One person reported a problem without specifying it. Two others mentioned difficulties that did not correspond to the intended definition, including "couldn't find error" (one person) and time problems (one person).

A total of 22 responses were submitted. The distribution was as follows: response a) was selected four times, response b) 15 times, and response c) nine times. Of these responses, 13 included a justification. Therefore, twelve students chose the correct answer. Three of the justifications were assessed as correct and sufficient; however, one of them led to an incorrect answer.

6.3.12 Fr4x1 - Code Snippet Comparison

The processing times ranged from 2.91 s (for participants who did not start the task or only looked at it briefly) to 514.09 s. The overall average for all participants was 103.09 s (3504.94 s in total). Looking only at those who actually worked on the task, the mean was 129.74 s (3373.24 s across 26 people). Depending on the debriefing questions, six, five, and four people, respectively, stated that they had not started the task.

Regarding the comprehensibility of the task, 19 people agreed that it was clearly and unambiguously formulated; five people even strongly agreed, while two disagreed and one person did not agree at all. The mean value of the corresponding debriefing question was 3.04.

When assessing the level of difficulty, 17 participants stated that it matched their knowledge and abilities; four strongly agreed, six disagreed, and one person disagreed completely. The mean value here was 2.86.

When asked about the amount of time available, 14 people agreed that it was sufficient, while six strongly agreed. Three disagreed and six strongly disagreed. The average score was 2.69.

With regard to technical problems, the picture was largely positive: 30 students stated that they had not encountered any difficulties (according to their own definition of the term). Two people reported a problem without specifying it in detail. Another person reported a difficulty that did not fall under the intended definition of technical problems, namely a problem with the time available.

A total of 15 responses were submitted. The distribution was as follows: Snippet 1 was selected ten times, snippet 2 seven times, and snippet 3 twice. Five participants chose the correct snippets. Of these responses, 14 included a justification. One of these justifications was deemed correct and sufficient and adequately justified the respective response selection.

6.3.13 Fr5x1 - Functionality Check

The processing times for **task Fr5x1** ranged from 4.42 s (for participants who did not start the task or only looked at it briefly) to 513.34 s. The overall average for all participants was 167.44 s (5692.99 s in total). Looking exclusively at those who actually worked on the task, the adjusted mean was 170.81 s (4953.39 s across 29 people). Five people stated that they had not started the task.

Regarding the clarity of the task, 18 participants agreed that it was clearly and unambiguously formulated, and six people strongly agreed. Three disagreed and one person strongly disagreed. The mean value of the corresponding debriefing question was 3.04.

Thirteen people rated the level of difficulty as appropriate for their knowledge and skills, two strongly agreed, twelve disagreed, and one person strongly disagreed; the average was 2.57.

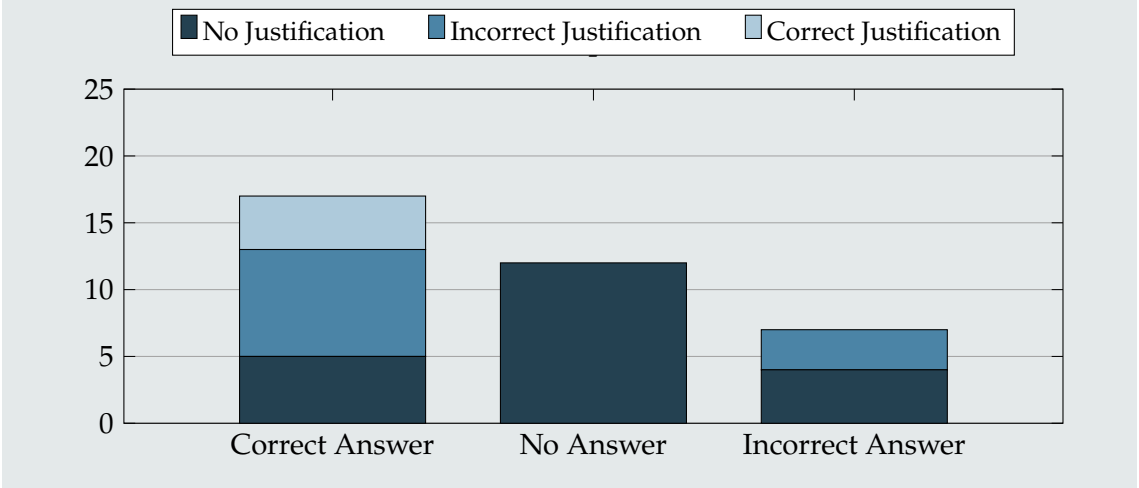
With regard to the processing time available, 14 people agreed that it was sufficient, and

four people strongly agreed. Seven disagreed and three strongly disagreed. The mean value here was 2.68.

With regard to technical problems, 27 students reported no difficulties (according to their own interpretation). Two people stated that they had had a problem but did not specify what it was. Four participants reported problems that did not correspond to the intended definition of technical problems: one each related to time, understanding the task, insufficient skills, and programming knowledge.

A total of 24 answers were submitted. 17 of these were correctly answered with "no." Of the 15 answers that also included a justification, four were classified as correct and sufficient and adequately justified the respective answer choice.

Fr5x1



6.3.14 Fr5x2 - Code Evaluation

The processing times for **task Fr5x2** ranged from 2.63 s (for people who did not start the task or only looked at it briefly) to 398.99 s. The overall average for all participants was 78.29 s (2661.9 s in total). The overall processing time was probably shorter because no justification was required for this task. Looking only at those who actually completed the task, the adjusted average was 120.83 s (2537.37 s across 21 people). Six people stated that they had not started the task.

Regarding the clarity of the task, 18 participants stated that the task was clear and unambiguous, five strongly agreed. Four disagreed and no one strongly disagreed. The mean value of the debriefing question was 3.04.

The level of difficulty was assessed by 19 people as appropriate to their knowledge and abilities, two strongly agreed, five disagreed, and one person strongly disagreed; the average was 2.81.

With regard to the available processing time, 14 people agreed that it was sufficient, four strongly agreed. Six disagreed and three strongly disagreed. The mean value here was 2.70.

Technical problems did not occur for 28 students. Two people reported a problem but did not specify it, and two participants reported problems that did not correspond to the intended definition of technical difficulties: one related to the processing time, the other to programming skills.

A total of 17 answers were submitted. Four answers were completely correct, three were partially correct or incomplete, and ten answers were classified as incorrect or insufficient.

6.3.15 Fr5x3 - Code Improvement

The processing times for **task Fr5x3** ranged from 2.24 s (for participants who did not start the task or only looked at it briefly) to 215.81 s. The overall average for all participants was 65.22 s (2217.55 s in total). The overall processing time was probably shorter because no justification was required. For those who actually completed the task, the adjusted average was 108.02 s (2160.4 s across 20 participants). Six people stated that they had not started the task.

Regarding the clarity of the task, 18 participants rated the task as clear and unambiguous, and six strongly agreed. One person disagreed and two strongly disagreed. The mean value of the corresponding debriefing question was 3.04.

Regarding the level of difficulty, 16 people stated that the task corresponded to their knowledge and abilities; three strongly agreed, five disagreed, and three strongly disagreed. The average was 2.70.

15 participants agreed that the available processing time was sufficient, four strongly agreed, two disagreed, and six strongly disagreed. The mean value here was 2.63.

Technical problems did not occur for 32 students. One person reported a problem but did not specify it, and one reported difficulties that did not correspond to the intended definition of technical problems: processing time.

A total of 17 responses were submitted. Three responses were completely correct, eight were partially correct or incomplete, and six responses were rated as incorrect or insufficient.

Table 6.1: Overview of Task Results

Task	**#Answers**	**#Correct**	**#Justifications**	**#Correct Justifications**
Fr1x1 – Loop Index Understanding	30	11	18	6
Fr1x2 – Edge Case Handling	27	14	18	12
Fr1x3 – Flowchart Comparison	26	20	13	10
Fr2x1 – Detect Output	31	7	18	7
Fr2x2 – Assigning Inputs/Outputs	23	1	11	4
Fr2x3 – Detecting Statements	22	2	8	1
Fr3x1 – Method Naming	16	0*	13	1
Fr3x2 – Error Detection	22	12	13	3
Fr4x1 – Code Snippet Comparison	15	5	14	1
Fr5x1 – Functionality Check	24	17	15	4
Fr5x2 – Code Evaluation	17	4	–	–
Fr5x3 – Code Improvement	17	3	–	–

Chapter 7

Discussion

This chapter builds on the results presented above and focuses on their interpretation. First, the results of the individual tasks are analyzed and key findings are highlighted. Based on this, a critical reflection is made on the extent to which the construction criteria used need to be adapted or refined. Finally, the implications for further research on code comprehension are discussed so that conclusions can be drawn regarding both methodology and content.

Table 7.1 shows an overview of the tested tasks with focus on the three facets of code comprehension — analytical skills, abstraction skills, and critical thinking — and evaluates their effectiveness or suitability. In a nutshell, most of the tasks are rated as suitable for assessing one or more code comprehension facets. Specifically, tasks such as **Fr1x1, Fr1x3, Fr2x1, Fr2x3, Fr3x2, Fr4x1, Fr5x1, Fr5x2**, and **Fr5x3** are mostly suitable, as they provide a valid assessment of the relevant facets. Some tasks, however, need to be adjusted: **Fr2x2,** for example, proved too difficult for many participants, and errors often arose from task complexity rather than a lack of understanding. Finally, a few tasks were not suitable: **Fr1x2** relied on language-specific knowledge, and **Fr3x1** was too complex, making it unsuitable for reliably assessing code comprehension. Overall, the results suggest a need for future task adjustments to cover all facets more comprehensively.

7.1 Task-Specific Analysis

Even though the sub-tasks actually belong together, they will be considered individually below. The following table shows an overview of the tasks and their performances regarding the testing of intended code comprehension facets. The phrase "mostly suitable" in this scope means that the tasks are fundamentally suitable, but should be adapted accordingly for further use.

Table 7.1: Overview of Tasks: Tested Facets of Code Comprehension and Suitability

Task	Analytical Skills	Abstraction Skills	Critical Thinking	Effectiveness/Suitability
Fr1x1 – Loop & Index Understanding	Yes	Partially	Not tested	mostly suitable
Fr1x2 – Edge Case Handling	Yes	Partially	Not tested	not suitable due to required language specific knowledge
Fr1x3 – Flowchart Comparison	Yes	Partially	Not tested	mostly suitable
Fr2x1 – Detect Output	Yes	Partially	Not tested	mostly suitable
Fr2x2 – Assigning Inputs to Outputs	Yes	Yes	Not tested	too difficult for many; errors often due to task complexity rather than lack of understanding
Fr2x3 – Detecting Correct Statements	Partially	Yes	Not tested	mostly suitable
Fr3x1 – Method Naming	Partially	Partially	Not tested	not suitable due to the code being to complex
Fr3x2 – Error Detection	Yes	Partially	Yes	mostly suitable
Fr4x1 – Code Snippet Comparison	Yes	Partially	Partially	mostly suitable
Fr5x1 – Functionality Check	Partially	Partially	Yes	mostly suitable
Fr5x2 – Code Evaluation	Yes	Partially	Yes	mostly suitable
Fr5x3 – Code Improvement	Yes	Yes	Yes	mostly suitable

7.1.1 Fr1x1 - Loop Index Understanding

Results of the debriefing questions show that students viewed the task as clearly and unambiguously phrased, so that misunderstandings due to that factor can be largely ruled out. The difficulty seemed to match the majority of the students' knowledge and skill levels. Regarding the time used to answer the task, a wide range between very short and very long processing times can be seen. Since those differences can only be interpreted to

a limited extent because of the previously discussed reasons, they do not receive much weight here. They can most likely be described by various strategies used by students, organizational elements, such as the tasks' sequence, and less about the tasks themselves. Still, the majority of students perceived the available time as sufficient.

In summary, the task was well-constructed both in terms of content and form: clearly understandable, appropriately difficult, and generally solvable within a reasonable amount of time. Limitations arise only in the interpretation of the completion times, which, given the context, should not be used as a central assessment criterion.

Of all 28 answers, 17 were justified by the students. But only five of those justified the given answers correctly and sufficiently. It is noticeable that four of those justified correct answers, while the other one seemed to be correct in terms of its content but was assigned to an incorrect answer. This may indicate that the distinction between the answer options was not always clearly perceived, and mistakes may not necessarily be due to a lack of understanding but may also be due to incorrect classification or an unclear answer structure. Since only one student mismatched their answer to their justification, there is also the possibility they just drew a wrong conclusion or chose that answer by accident.

By analyzing the justifications, the following difficulties while answering the task could be identified. A lot of participants seemed to have problems differentiating the index and the element itself as well as the modification of an already existing list and the creation of a new one. In addition, the results show a strong focus on single if-conditions and local calculations, while the holistic understanding of the code – i.e., the goal of the calculation and the final return – faded into the background. Partially, the logic of conditions was correctly identified, but the choice of answer was not correctly or consistently justified or even assigned. You can also see that a lot of students took a procedural (step-by-step) approach while working with the code. Therefore, they are often missing an abstracted perspective on the whole concept.

The few correct and sufficient justifications stood out by having a clear and structured depiction of code. Those explanations described the iteration, the different processing of elements at even and odd indices, and the creation of a new list for results as well as the final calculation of the sum. What was crucial here was the fact that students identified the whole purpose of the code and did not just focus on single aspects, such as specific lines and their effects.

The following insights could be gathered by analyzing the results: First of all, the high error rate, despite a clear and unambiguous task, shows that the task itself seemed to be a challenge for a lot of the students. Further, the strong focus on local code structures and conditions suggests that students may have analytical skills, but lack the ability to abstract. The incorrectly assigned answer, in spite of correctly justifying the code, may show that the answer options themselves were not always clearly differentiated from one another. Finally, it becomes clear that the quality of the justifications was overall rather low, which shows that the explanatory component represented an additional difficulty for many participants.

Several adaptations can be derived for further developing the task. As the first option, it

could be designed in a more structured manner, for example, by asking additional guiding questions, such as "Why is the chosen answer correct? Explain in 2-3 sentences what happens to the original list and what exactly is returned.“ This could prevent students from giving step-by-step explanations without comprehending the overall concept. Additionally, questions that explicitly aim at abstract thinking skills could be asked. For example, "How could the code be modified to directly modify the original list?". Still, the unguided version used in this test helped find different and, for students, natural, unforced approaches that may help form a mental model.

From the point of view of the abilities to be tested, it can be said that analytical skills were often visible within the students' answers, in which many participants could describe what happens in specific parts of the code. The ability to abstract, on the contrary, remained weak. The majority got stuck on analyzing local structures without identifying the purpose of the whole code or explaining the logic in a broader context. Therefore, it can be said that the task indeed tests analytical skills properly. Since some students showed a correct sense of abstraction, the ability to abstract is also being tested, but much less effectively.

In the future, similar tasks should therefore place greater emphasis on promoting and testing abstract thinking skills by integrating clear guiding questions and additional levels of reflection into the task.

7.1.2 Fr1x2 - Edge Case Handling

The majority of students rated the task as clear and unambiguous, as the lowest rating (strongly disagree) was chosen by no one and the average value was 3.1. The difficulty was also mostly seen as fitting to the students' skill level and their knowledge. This shows that the task was not made more difficult by unclear wording or by a level of difficulty that was too high or too low. The times needed by students also show a big range, but as already mentioned, this should not be too highly interpreted, as it can just be an effect of the different orders of the task. Accordingly, time is considered here only as an incidental factor. Still, students perceived the given time for that task as sufficient, as only ten students rated it negatively. Technical problems only occurred to a very small extent and were in most cases not attributable to the actual execution of the task, but rather to individual problems regarding the content. The results indicate that the task was well designed regarding the topics ruled out by the debriefing.

Half of the given answers were correct by choosing option b), while a) was also often chosen instead of c). Which seems to support the assumption that the level of difficulty was neither too hard nor too easy.

Analyzing the 16 (seven correct) given justifications makes it clear that a lot of students had problems abstracting the logic as an overall concept. Oftentimes they focused on individual details, such as initializing variables or isolated calculations, while the overall functionality of the code was neglected. Other justifications remained too superficial or were characterized by false assumptions, whose origins may lie in experiences with other programming languages. But looking at the other most used programming languages C, Java and C++, the code would show similar behavior with exactly the same implementation.

Still, it can not be ruled out that misinterpretations may be a sign of a negative impact resulting from previous experiences with other programming languages. In addition, several answers showed that the behavior of the code when given an empty input was not correctly recognized: instead of stating the correct execution without loop iterations and the return of the initial value, some participants incorrectly assumed default values or error messages.

Correct justifications distinguished themselves by a structured approach. They first identified that the input list was empty and therefore the loops were not processed. Based on this, it was correctly derived that the variable never changes its initial value and therefore returns 0. Those explanations proved that students were able to put the individual parts of code into an overall context and to formulate the expected result precisely.

The following conclusions can be derived from the results: Firstly, the comparatively small number of justifications seems to result from a difficulty of expressing the students' insights linguistically. It can also be a sign of insecurities regarding their knowledge or just guessing the answer. Secondly, incorrect justifications may show that the ability to abstract was not sufficiently developed in all cases: some of the students concentrated on local aspects of code, without taking the whole logic into consideration. In the third place, it becomes apparent that the task not only tested abstraction skills but also language-specific knowledge, as incorrect assumptions about Python's behavior played a role.

Regarding the rating of facets of code comprehension, it can be said that a pronounced use of analytical skills was shown in some of the justifications. But correct justifications also showed that students were capable of combining individual steps of code into a bigger picture. At the same time, incorrect or incomplete justifications show that this did not apply to everyone and that some had only a superficial or fragmented understanding.

To improve the task, it seems useful to structure the justification requirement more clearly. More precise guiding questions that not only ask about the result but explicitly about the reason for the absence of errors when the input is empty could help to better distinguish between superficial and deep understanding. This would allow the goal of testing abstract thinking skills in the sense of a holistic understanding of code to be achieved even more clearly.

In conclusion, it can be said that the task efficiently tests analytical skill as well as the ability to abstract, but only to a limited extent. Correct justifications show that students are indeed capable of combining these two skills to correctly justify their choice of answer. On another note, the use of Python-specific behavior posed a problem for students with missing experiences in that programming language. Also, it should be further discussed whether or not language-specific knowledge is an essential part of code comprehension. Therefore, the task may not be as fitting as intended and should be reworked or eliminated.

7.1.3 Fr1x3 - Flowchart Comparison

The results of the debriefing questions show that with an average value of 3.03, students seemed to understand the task clearly, therefore ruling out any misconceptions resulting from ambiguous or confusing phrasing. Also, the level of difficulty was similarly rated as

in the previous tasks and therefore suggests that it was neither too much nor too little of a challenge for the participants. With regard to completion time, a more differentiated picture emerges: Although many stated that they had sufficient time, the significance is limited due to the uneven distribution of tasks within the test. Nearly no technical problems occurred and therefore did not impact the processing of the tasks in a relevant or negative way. The few named problems referred to personal assessments regarding students' knowledge and the limited time, but not to the technical aspect itself. Therefore, it can be said that the task was successful in its phrasing and level of difficulty and could reliably test the intended competencies in the scope of the debriefing.

The task was designed for testing analytical skills as well as the ability to abstract. The students were required to recognize the structure and logic of a given code in a visual representation, such as a flowchart. Results show that the vast majority of participants identified the correct flowchart, which could possibly reveal any analytical abilities. The comparatively small amount of correct and sufficient justifications, by contrast, suggests that although many students were able to select the correct solution, they were not always able to support their decision with sound reasoning. Partially, the correct choice could have been a result of recognizing patterns or even guessing the answer and therefore show less of a deep understanding of code.

The given justifications reveal different strategies: often students chose a pattern- or rule-based approach, where they viewed single, isolated conditions or operations. Others concentrated on comparing single operations between the code and the flowchart (e.g., the correct presentation of multiplication and subtraction). Less commonly, students reconstructed the entire logic of the code, for example, by taking into account the final summation. This indicates that analytical skills were only partially tested in this task, as many participants focused on individual conditions without systematically considering the overall context. The ability to abstract was also only partially visible: only a few explanations showed a holistic understanding of the code and its dynamics.

Therefore, two central findings can be concluded: On the one hand, the task is able to capture basic aspects of code understanding, especially the recognition of conditions and their representation in a flowchart. On the other hand, it shows definite potential for improvement, since the task in its current form tends to encourage rule-based thinking, while deep analysis and abstract thinking are less required. For future references, the task could be expanded by additional versions of the flowchart with different kinds of errors. With that adjustment, students cannot just get the answer by concentrating on single parts of the codes. Also, guided questions could be asked that require consideration of the effects of alternative code changes (e.g., how the output would change if the operation were changed). This would make it possible to differentiate more clearly whether participants understand the code as a whole or merely identify individual patterns.

7.1.4 Fr2x1 - Detect Output

The analysis of the responses to debriefing questions indicates that it was clearly phrased and did not cause any misinterpretations. So, the majority of students stated that they understood the task and found the level of difficulty to be appropriate in relation to

their own knowledge and skills. The processing time provided was also largely rated as sufficient, although individual use of the time varied greatly. For most students no technical issues occurred while working on the task. Only isolated feedback related to aspects such as the programming language used or the perception of one's own level of knowledge. Overall, it can be concluded that the task was carried out under largely optimal conditions and that the measured processing times reflect the individual processing strategies and speeds rather than difficulties due to unclear task definitions or technical limitations. These findings support the interpretation that the task was easily accessible for the majority of participants and that the results primarily allow conclusions to be drawn about the content processing and not about external factors.

Task Fr2x1 was designed to test analytical abilities and the ability to abstract. The task required students to analyze the code and divide it into smaller components to fully understand its function. This also includes identifying the initial value of the iteration variable as well as comprehending how the loops handle which elements. In addition, participants had to analyze the effects of the if-else condition, which treats odd numbers on the one hand and even numbers on the other hand differently, and take into account that each iteration adds a final character "C". The ability to abstract is also partially required, since students had to combine those individual components into one complete construct. This also involved deriving a general rule for string generation, such that each processed numeric value produces either "AC" or "BC" to predict the correct output. The analysis of the responses indicates that a majority used a pattern- or rule-based approach by understanding the condition "odd = A, even = B" but often made mistakes in implementing it. A frequent source for mistakes was an improper use of the derived rule, the overlooking of the starting value for the iteration, or imprecise assumptions about the loops. Students who based their approach on the identification of patterns were able to partially capture the general scheme but neglected details within the iteration logic. Differences between top-down and bottom-up thinking were visible: some abstracted the code globally and described its function on a high level, while others proceeded step-by-step but often did not take into account the starting position or the exact loop logic.

Correct justifications are characterized by a structured analysis of the codes functionality. They clearly explain how the iteration begins at numbers[2], how the strings are generated, and how the condition controls the selection between "A" and "B" before the final "C" is added. The step-by-step derivation of the output demonstrates that a deep understanding of loop and conditional logic is necessary.

The results demonstrate that the task did indeed test the participants' analytical skills, as understanding the conditions and iteration logic was crucial for a correct solution. At the same time, the explanations show that abstract thinking skills were only partially assessed: While some participants recognized general patterns, they often lacked the connection between the rule and the precise iteration logic. This led to incorrect answers, especially when incorrect assumptions about starting values were made, even when the rule itself was correctly understood.

In summary, the task tests both analytical skills and, to a limited extent, the ability to think abstractly. It particularly emphasizes the need to not only analyze code in isolation for

specific conditions but also to consider the entire process. To improve the differentiation between thinking styles and the assessment of abstract thinking skills, future tasks could specifically include comparison or in-depth questions that explicitly require the linking of pieces of information to form an overall understanding.

7.1.5 Fr2x2 - Assigning Inputs to Outputs

The vast majority of students rated the phrasing of the task as clear and unambiguous. The high level of approval (18 students agreed, two strongly agreed) supports that assumption as well as also being reflected in the average value of the corresponding debriefing question. From this it can be concluded that the task was communicated in a comprehensible and unambiguous manner. The difficulty level of **Fr2x2** in comparison to other tasks shows a slight decline in the average mean. This was caused by 14 students disagreeing (two of them strongly disagreeing) that the difficulty matched their own skill level. By contrast, only 16 (two of them strongly agreeing) stated that the required skill matched their own. Since almost half of the students that started that task did not perceive it as fitting difficulty-wise, it should be further assessed, and the main problem should be identified and reworked accordingly. In contrast to the recorded times needed to answer the task, the assessments taken by students allow clearer conclusions to be drawn. Similar to the difficulty of the task, almost half of the students disagreed (six of them strongly disagreeing) that the available time was sufficient. However, this assessment is methodologically difficult to interpret, as the test's time frame was not primarily designed to allow for the thorough completion of all tasks within 30 minutes. Thus, this question has only limited significance for interpreting the results. Still, considering the results of the debriefing question regarding the difficulty, the numbers suggest a task refinement for further testing. Finally, the overwhelming majority of respondents indicated that technical issues did not play a role in their performance in the intended sense. However, the issues they reported support the previously analyzed results from the other debriefing questions and further demonstrate that problems arose in understanding the task itself. In summary, the results indicate that the task may not have been appropriate to the level of difficulty for the majority of students. Accordingly, any errors in completing the task could be attributed to these deficiencies rather than to students' lack of knowledge. However, there is little evidence of interference from technical problems.

We designed the task to test analytical skills and the skill to abstract. The former was required by needing to divide the code into smaller components and comprehend how they work in detail. This included understanding the function generate_string, analyzing the behavior of the loop (starting at i = 1, the incrementation and the ending condition), as well as identifying the condition numbers[i] % 2 != 0 and their impact on the output. Additionally, the ability to abstract was used to combine the individual findings into an overall understanding. In addition, the requirement to simulate different inputs also encouraged systematic testing.

The answers and their justifications show distinct differences regarding the approaches used by students. Of 23 given answers, only two showed the correct order. Ten students completed their answers with a justification, with four of them being correct, although

three of them were linked to incorrect final answers. Evaluating the incorrect justifications showed that a significant part of students used a rule-based and simplified thinking, in which the code logic was reduced to a supposedly general rule. This usually lacks detailed code walkthroughs with concrete values. Many students instead relied on a superficial pattern-matching method, where recurring sequences were detected in the outputs but incorrectly transferred to the positions in the result. That approach often leads to an incorrectly identified starting point and iteration logic. From that, an existence of analytical thinking could be derived, but the approaches lacked a consequent implementation that was often replaced by a heuristic approach.

Correct justifications, on the contrary, showed a clear structure and overall understanding of the code's logic. They all had precise descriptions of the iteration process over the list, the checking of the parity of the numbers, and the resulting generation of "A" for even and "B" for odd numbers, followed by the systematic addition of the character "C" in common. They only differed regarding the set focus on execution: While one answer emphasized the process in terms of a sequence of actions, another focused more on the correctness of the mapping between inputs and outputs. Another focused on the later course of the iteration but demonstrated the same basic logic.

Regarding the code comprehension target criteria, it can be stated that the successful justifications demonstrate a high level of analytical skills: The participants were able to correctly identify and explain the loop logic, the condition, and the structure of the result string. At the same time, the ability to abstract is also demonstrated, as the participants not only described the functionality for a single case but were also able to transfer the logic to different inputs and outputs and recognize structural similarities.

The overall small amount of correct answers and justifications demonstrated that the task was too complex for a majority of the students, as also previously seen in the debriefing. The tendency to identify patterns and the simplified heuristic approach, in particular, show that the intended competencies were addressed but not fully activated in the scope of a too complex task with a confusing visualization. For future testing, a reworked difficulty level, phrasing, and answer mechanic are recommended to enable greater performances and reduce misinterpretations.

7.1.6 Fr2x3 - Detecting Correct Statements on Behavior

Looking at the results of DeFr2x3.1 it becomes clear that students predominantly perceived the task as clearly and unambiguously phrased. Almost the entirety of the student agreed to that in regard to the wording so that misunderstandings regarding the work assignment can be largely ruled out. It can be therefore assumed that the content was communicated precisely and transparently. The difficulty level of the task was also mostly perceived as appropriate. Both in regard to their skill level and their level of knowledge, students rated the task as neither too hard nor too easy. This suggests that the task development has successfully reached a level that allows for fair processing. Even though the mean value of DeFr2x3.3 was lower than the other debriefing questions of task **Fr2x3**, the overall rating suggests a fair and sufficient amount of available time, with two thirds agreeing to the statement "I felt I had sufficient time to complete the task without feeling rushed." Beyond

that, it could be ruled out that technical problems (in the sense of the intended meaning and definition of the term) caused any trouble while answering the task, and therefore the test environment was stable and external factors did not significantly influence the results. Participants who stated they had had a problem did not mention what kind of problem it was. In conclusion, it can be said that we designed the task to be answered under favorable conditions: it was clearly phrased, appropriately difficult, and mostly free of any technical problems. Only the time assessment needs to be considered with some nuance, as its significance is limited due to the structure of the test. Nevertheless, the concrete time figures speak for the design of the task.

The task was also focused on examining the students' abilities to analyze and abstract. Again, students had to dissect the whole code into smaller parts and thoroughly understand the way those work on their own. E.g., analyzing the starting value of i, the condition if numbers[i] % 2 != 0, the behavior of the loop with regard to starting from the second element and ending before the last element, as well as the effects of the loop condition on short input lists. By combining those smaller code elements into an overall understanding, which included both the iteration process and the transformation of the numbers into the output patterns, students had to apply the ability to abstract.

However, the evaluation of the results shows that this goal was only partially reached. Answers distributed between different options, without a clear majority emerging for the right solutions. Even though a lot of students chose at least one correct statement, only a fragment answered correctly by choosing both. Especially answer d) was chosen rarely. This may indicate that the task or its answer option was not fully understood by most of the students. It was particularly striking that answer options that contained code-specific details were chosen less than others. This suggests that central subtleties of the loop logic – such as the starting index i = 1 or the condition i < len(numbers) - 1 – were not taken into consideration by many students.

The given justifications determine this finding. The explanations and justifications mostly stayed on a superficial, basic level: the process of iterating was described in a general manner without taking into consideration concrete values or the role of individual variables. Therefore, a step-by-step evaluation of code was missing in multiple cases, even though this is a crucial step for understanding and analyzing. Although a basic mental model was evident in many answers, it was often incomplete or overgeneralized. So analytical thinking was recognizable in its early stages; however, it was not implemented consistently.

The few correct explanations, however, exemplify the level of code comprehension that can be achieved through the task. Those examples show how students comprehended the omission of the first element and the special treatment of lists with fewer than three elements. This indicates that a precise dissection of code in combination with a step-by-step understanding of iteration was possible.

Regarding the abstraction of code, the results show a mixed picture. A lot of students were able to name the main function of code but displayed difficulties combining this abstraction with specific details of iteration or logic. This led to justifications that described the main function, but the specific interactions between starting index, condition and output pattern were inadequately explained.

To conclude the assessment of **Fr2x3**, the task was fundamentally suitable to address the intended facets of code comprehension. However, the analytical level of detail was not fully apparent in many students. For future use, it appears to make sense to more strongly shift the focus onto a step-by-step execution of code. This could be achieved by adding guiding questions that require students to further document concrete values and conditions. In this way, the task could promote not only abstract logic but also the precise analysis of individual steps and thus provide a more comprehensive picture of the students' code comprehension.

7.1.7 Fr3x1 - Method Naming

The analysis of the results suggests that the task was predominantly perceived as clear and understandable by participants. The vast majority agreed with the statement that the task was clearly formulated, largely eliminating misunderstandings regarding the task. This suggests that the instruction was precise in content and enabled fair completion. The assessments of the difficulty level also confirm the appropriateness of the task. Students largely rated the requirements as compatible with their own level of knowledge and skills. This indicates that the task was neither too challenging nor too easy but rather met the intended target group's skills. Feedback on available time also shows a predominantly positive rating. While most respondents stated that the time available was sufficient, this assessment is only open to limited interpretation. Since the test tasks were not designed to be completed within 30 minutes, and some students only received the task at the end, statements regarding the time dimension should be viewed with caution. Nevertheless, the overwhelmingly positive assessment suggests that the test was completed under realistic and fair conditions. Furthermore, technical problems played an insignificant role. Almost all participants reported no disruptions. The few problems that were mentioned either referred to aspects that did not meet the intended definition of technical difficulties (e.g., language or task clarity) or remained unspecific. This underscores that external technical barriers did not influence the processing or results. Even though two students used that question to report unclear instructions and insufficient time, this is not worth mentioning, as this opinion is most likely reflected in the previously discussed figures, and it concerns only a small fraction of participants. In summary, the task was clearly formulated, of appropriate difficulty, and took place under largely uninterrupted conditions. The only limitations are in the interpretation of the completion times, as these are of limited significance due to the test structure. Overall, however, the results indicate a high level of task validity.

We designed Task **Fr3x1** to examine the students' abilities to analyze and to abstract. The former was required by systematically dividing the code into smaller fractions and understanding how the indices i and j handle the loop and how the conditions (if v[i] < v[j] and related structures) affect the values and under what requirements the loop ends. Without a step-by-step analysis like this, it is difficult to determine the functionality of the code correctly. The ability to abstract should be demonstrated by deriving an overall understanding from the individual operations: The algorithm changes the values in the beginning and the end of the list iteratively; larger values are reduced by smaller ones so that the final values are approximated. On this basis, participants should be able to give

the function a suitable, abstract name.

But the results show a different picture, in which this goal is very rarely achieved. Only one of the given answers was correct and reflected the purpose of the code. The small amount of justifications also did not show a correct derivation in terms of content. A key observation is that many participants used mental models from known algorithms - e.g., for example, similarities to the Euclidean algorithm (GCD) through repeated subtraction. This indicates that students tried to close any gaps within their knowledge by using the knowledge they had that seemed to be relevant for that case. Also, those used models were often utilized incorrectly or inaccurately, which also led to wrong conclusions. Other justifications merely focused on the end result instead of the mechanism that leads to that result. This resulted in generalized statements such as "the code sorts the list" or "it always creates an array of ones" that describe patterns but miss the underlying logic. Those tendencies indicate a rather superficial analysis of students, without dissecting the code more precisely. Detail errors in the interpretation of index movements or termination conditions demonstrate that the mechanism was not systematically reproduced. While many participants recognized recurring patterns, they lacked the ability to connect these observations to the specific operations of the code. This means that existing analytical skills were used only to a limited extent.

The ability to abstract was also less pronounced than expected. While participants recognized repeated subtractions or structural similarities to known algorithms, they generally derived incorrect generalizations from them. Instead of an abstract description of the actual function, they relied on intuitive but incorrect assumptions. This demonstrates that while the task was fundamentally suitable for testing abstraction, it also allowed participants to resort to superficial heuristics.

Overall, the results indicate that the task was more cognitively demanding than originally expected. However, the majority of participants found the wording clear and the difficulty appropriate, so any comprehension issues are not due to the task itself. Rather, the problem seems to lie in the fact that the task allowed for too much surface-level processing rather than consistently forcing students to engage in detailed analysis.

For further development, it would be advisable to adapt the task so that it requires more step-by-step understanding. This could be achieved, for example, through intermediate questions that ask about the value of individual variables in different iterations, or through incomplete pseudo-code that students must complete. It could also be useful to modify the code so that it is less likely to be confused with known algorithms. However, this could be discussed further, as proximity to known algorithms can also be helpful in determining whether students have truly understood the code in full detail and are not just looking for known patterns. In this way, the goal of testing both analytical skills and the ability to think abstractly in a more differentiated manner could be more clearly achieved.

7.1.8 Fr3x2 - Error Detection

Analyzing the needed times shows clear differences. While some students quit the task after a short amount of time or did not even start it, others took a lot longer. This suggests

that the task was perceived heterogeneously and that different approaches to its processing existed. Since the test was not designed to be completed within a fixed time frame, simply looking at the time values is of limited value. Rather, the results illustrate that students worked at different speeds and with different task orders, without necessarily allowing conclusions to be drawn about the quality of their understanding. Particularly noteworthy is the comparatively high average value for DeFr3x2.1 which is a result of 24 students agreeing that the task was phrased clearly and unambiguously (five of them strongly agreed), while five students disagreed. This indicates that difficulties in understanding do not result from unclear instructions but rather from the content of the task itself. Question DeFr3x2.2 had a rating that was not as high but still fairly good, which indicates that the task was appropriate to the students' prior knowledge and skills. It was therefore neither overwhelming nor too trivial but presented a challenge that was appropriate in terms of content. Regarding the available time, it can be said that the majority of students stated that they had had a sufficient amount of time. The technical implementation of the task did not seem to have posed any problems to students. Only two participants reported that they encountered problems. However, these did not match the intended definition of technical issue, instead they concerned either a problem with understanding the code or the given time. Therefore, it can be said that potential barriers in the test environment played little role and the results were not distorted by external factors. Overall, it can be concluded that, from the participants' perspective, the task was clearly understandable, appropriately difficult, and technically feasible. Difficulties apparent in the completion times are therefore more likely to be due to individual strategies and content-related engagement than to external factors such as time pressure, unclear task definitions, or technical issues.

Task Fr3x2 was one of the few questions that did not only test analytical skills and the ability to abstract but also critical thinking. On an analytical level, students had to be capable of dissecting the given code step-by-step and understanding the workings of the loop. This included interpreting the condition i < j as well as the various branches that lead to modifications of the list elements v[i] and v[j]. Crucially, it had to be recognized that the list elements are mutually modified until the indices meet or overtake each other.

Alongside the analyses of individual code lines, the ability to abstract was tested again by having students recognize the underlying logic of the algorithm and not just react to specific inputs. That included the identification of a superior pattern that appears similar to the Euclidean algorithm regarding its structure.

In addition to those two facets, critical thinking was tested, because participants had to rate the behavior of code with specific inputs and identify the cases that uncovered any issues within the code. Especially answer c) posed a challenge since it could potentially lead to endless loops that may endanger the stability of the code.

Answers showed clear differences in the depth of the approaches. A lot of students mostly relied on intuitive assumptions or processes of elimination rather than systematically working through the code. A common pattern was the focus on individual structures, such as the condition i < j, without understanding the interactions of the different parts of code and their effects. That isolated view led to misconceptions and wrong assumptions, such as

that the endless loop arises solely from the condition, without taking into account the lack of change in the variables in the overall context. Additionally, a lot of justifications showed pattern-based thinking that relied on general programming experiences and knowledge but was not secured by concrete tests with input values. Because of that, analysis often remained purely theoretical, without actually comprehending the execution of code. Only a few participants were able to identify the causes of problematic behavior. Those answers were characterized by a differentiated analysis that not only explained the effect of a zero in the input list but also the lack of increase in the index variables in their interaction.

Regarding the intended facets of code comprehension to be tested, a mixed picture emerges. Students did show analytical behavior but mostly on individual parts of the code. The ability to abstract, on the contrary, was only shown a few times, since most students did not identify a superior pattern for the algorithm. In comparison, critical thinking was captured better, as some responses at least identified the faulty conditions symptomatically, even if the causes were not always correctly identified.

Several implications for the optimization of future tasks can be derived from these results. In order to promote abstract thinking skills more specifically, tasks should include explicit questions about the general functioning of the algorithm. In addition, mandatory simulation of the code with concrete inputs ensures that participants not only apply theoretical rules but also understand the actual behavior. A stronger focus on the cause of issues rather than just looking at symptoms, for example, by asking questions, such as "Why does this input lead to an error?" or "What role do the variables i and j play during execution?" Considering additional edge cases could also help to test critical thinking more broadly. Overall, the results indicate that the task primarily addresses analytical skills and critical thinking, while only inadequately testing the ability to abstract. A further development of the task format could therefore contribute significantly to capturing the different dimensions of code comprehension in a more balanced way and thus enable more differentiated insights into the cognitive strategies of the participants.

7.1.9 Fr4x1 - Code Snippet Comparison

The instructions of task **Fr4x1** were rated to be clear and unambiguous by the vast majority of students (24 students either agree or strongly agree). So the responses and the comparatively high average value of 3.04 suggest that any misconceptions that emerge while answering the question are not the result of unclear or confusing wording. While the average value of DeFr4x1.2 is not as high as for the previous question, the individual ratings of students state that their skill and knowledge level mostly fit the difficulty level required by the task. This indicates a successful balance between cognitive demands and individual performance. Regarding the given time students had to answer the task, a mixed picture emerges. While 20 students agreed (six strongly agreed) that the time was sufficient, nine students did not share that opinion. Six of them even strongly disagreed with the corresponding statement. Still, the majority perceived the time as sufficient, and therefore it can be said that time was not a main factor regarding any problems with the task. Especially since the meaningfulness of that debriefing question is limited. Another central aspect of the debriefing is the technical framework, which did not cause

any problems regarding the given responses. Instead, students either did not mention their problems or, again, named the limited time as a source for problems. It can therefore be assumed that external disturbances caused by technical factors had little influence on the processing. To conclude the debriefing section of **task Fr4x1**, we can say that the task was rated predominantly positively in terms of its wording and difficulty and thus appears to be successful both in terms of content and teaching methodology. Differences in completion times are more likely due to individual processing strategies and the organizational test structure than to deficiencies in the task itself.

The task pursued the goal of testing students' analytical skills, their ability to abstract, and their critical thinking. For that, the participants had to compare three functionally similar code snippets that provided differently implemented solutions for processing lists.

By looking at the results regarding analytical skills, it is apparent that the task tests the fundamental competence of comprehending code line-by-line. Most students were able to recognize the use of a loop combined with a condition that identifies negative values and replaces them with zero. They were also able to differentiate between the handling of a negative and positive value. This demonstrates that the conditional logic has been understood and a functional model for processing the input data has been created. However, those correct analyses remained mostly on a sequential level: many answers were based on a step-by-step execution, without reflecting the superior differences within the implemented strategy.

The ability to abstract was only tested to some extent. While students were capable of understanding individual code components, they often lacked the ability to put them into a bigger overall context and systematically compare them between different code snippets. For example, it was often overlooked that Snippet 1 modifies the original list, while Snippet 2 creates a new list, even though both lead to the same functional result. As well as disregarding the fact that snippet 3 varies from the given specification by solely changing positive values. This shows that the ability to abstract overarching patterns and differences was only inadequately addressed by the task. The task was also only partially suitable with regard to critical thinking, since most of the justifications were limited to the correctness of the condition test. Aspects such as efficiency or comprehensibility of the different implementations received little attention. Instead, simple "if-then" rules and a selective analysis of individual snippets dominated, without a systematic comparison between the solution approaches.

Overall, the results show that the task does indeed test fundamental aspects of code comprehension, especially the processing of conditions and loops; however, it only contributes limitedly to the promotion of higher code comprehension facets. Analytical skills were demonstrated in a basic form, whereas abstract thinking skills and critical thinking were only inadequately assessed.

For future references, it would be advisable to specifically incorporate comparative elements that encourage participants to systematically compare different implementations. Questions that explicitly direct to efficiency, comprehensibility, and more could contribute to strengthening critical thinking. Likewise, more in-depth reflection tasks could be introduced to promote the ability to place individual logic in a larger conceptual context.

Such an extension would enable a more differentiated and comprehensive understanding of the three facets of code comprehension.

7.1.10 Fr5x1 - Functionality Check

Analyzing the responses of the debriefing data showed that the task was perceived as easy to understand and appropriately phrased. Even though five students stated to not have started the task, the rest of the responses suggest that the instruction was phrased clearly and unambiguously, which suggests a high quality of the wording. With regard to the difficulty of the task, a more heterogeneous picture emerges. 13 students agreed and two strongly agreed that the task was suitable for their own skill and knowledge level. But with 13 students, almost as many stated that there was a noticeable difference between the required level and their own. Which indicates that compared to other presented tasks in this thesis, **Fr5x1** seems to be more challenging and that problems emerging while answering may stem from missing knowledge. Looking at DeFr5x1.3 a similar but slightly better rating is visible. Even though ten students in total stated not to have had enough time to answer the task, 18 students contradicted this statement. Therefore, the majority of students rated this task as successfully designed regarding the available time. The technical framework was mostly positively rated. 27 students did not state to have had any problems while answering the questions. Two of the six students who encountered problems did not specify which problem they had. While one student mentioned the used programming language as a problem, three students each mentioned the topics of the previous debriefing questions as problematic. This suggests that the test environment was stable and processing was not affected by external technical factors. In summary, the task was largely successful with regard to the debriefing questions. Only the difficulty of the task could be discussed again.

The examined task aimed at testing students' ability to analyze, abstract, and use their critical thinking. They, again, had to dissect the code into smaller pieces, comprehend their functionality, and identify any errors. Along with that, they had to combine those smaller components into one big concept and rate its correctness.

The evaluation of the responses indicates that many students intuitively recognized the implementation as faulty. Therefore, the majority of students correctly chose answer b). However, the quality of the given justification differed vastly. While some precisely identified the faulty handling of negative differences or redundant loops, others remained on a superficial level of explanation. Oftentimes they only implied the faultiness of code without specifying the cause or the impact the errors had on the behavior of code. This may indicate a sense for mistakes, but they have difficulty translating their intuition into precise analysis and reasoning. Also, it could be an indicator of a guessed answer.

In terms of content, different approaches from the students can be seen. Some viewed the code from a functional perspective and recognized the purpose of code fragments, calculating differences between consecutive numbers without consistently considering the overarching goal – determining the largest difference. Others viewed the code from a mathematical perspective and pointed out the need to work with absolute differences, which reveals a deeper reflection on the intended goal of the code. Some focused only on

structural aspects, such as redundant loops or unnecessary variables. Finally, part of the answers explicitly identified errors, albeit at varying levels of depth – from simply naming them to concrete suggestions for improvement.

These findings demonstrate that the task indeed activates various thought processes related to code comprehension. Analytical skills were demonstrated by students examining individual components of the code and identifying errors such as the incorrect handling of negative differences. However, the analysis often remained fragmented and did not always capture the consequences for the overall behavior. The ability to abstract was particularly evident when students identified the goal of the calculation as determining the largest difference or emphasized the importance of the absolute value. However, even here, some of the answers remained strongly focused on detailed errors without reflecting on the purpose of the code as a whole. Critical thinking was finally evident in the examination of efficiency and correctness, for example, by identifying unnecessary loops or by suggesting improvements. Nevertheless, only a small proportion of students provided a complete, well-founded explanation with a clear perspective on a solution.

Overall the results point to a possible gap in the knowledge or experience of students. Many students are able to identify errors, but few can explain them precisely and place them in the context of the overall code intent. This suggests that a deeper understanding of code must be trained not only through intuitive error detection but also through systematic reflection and reasoning. To more specifically capture and promote these skills, the question could be adjusted in the future. Instead of simply asking about the correctness of the code, more detailed tasks would be useful, such as identifying specific errors, correcting the code, or comparing it with a correct solution. This would not only test whether students recognize errors but also how they can analytically classify, abstract, and critically evaluate them. This confirms that the task is fundamentally suitable for assessing code comprehension, as it activates various thought processes. However, the analysis shows that more precise differentiation and a stronger demand for justification are necessary to obtain a complete picture of the existing competencies.

7.1.11 Fr5x2 - Code Evaluation

With 23 people agreeing on the statement that the task was phrased clearly and unambiguously and only four people disagreeing, we can say that the instructions did not pose a problem while answering the questions and therefore, did not represent a significant barrier to understanding. Similarly, the responses of DeFr5x2.2 indicate that the perceived level of difficulty also proved to be appropriate. The majority of students stated that the task corresponded to their level of knowledge and was considered neither too demanding nor too trivial. This suggests that the task was appropriately designed to test the intended competencies without being expected to be either over- or under-challenging. Regarding the available time, a mixed picture emerged. On the one hand, most participants stated that they had sufficient time. On the other hand, this assessment is qualified by the fact that the task was distributed within a test setting in which not all students completed it at the same time or within the same time frame. This makes it difficult to draw a clear conclusion about the actual time appropriateness. Nevertheless, the overall trend suggests

that time pressure was not perceived as a key obstacle. Technical difficulties played a virtually insignificant role. Almost all students reported no problems related to completing the task or the software used. Individual feedback listed as "technical problems" turned out, upon closer examination, to relate not to technical aspects but to content-related aspects, such as understanding the programming language. This underscores that the overall framework of the task was stable and trouble-free. In summary, the task examined was successful in several aspects: it was clearly formulated, appropriate in its difficulty level, feasible under acceptable time constraints, and free of technical barriers. Thus, it provides a solid basis for the intended assessment of the relevant competencies.

We designed the task to assess key facets of code comprehension, particularly critical thinking skills related to the correctness, efficiency, readability, and understandability of program code. Students were asked to evaluate a flawed Python code designed to calculate the maximum difference between two consecutive numbers. However, the code contained both logical errors and efficiency and understandability issues, requiring a more nuanced analysis.

The results show that while the students were able to identify individual aspects, only a small proportion were able to provide a comprehensive and accurate analysis. Of the 18 answers submitted, only four were completely correct, while eleven were inadequate or incorrect. This finding indicates a clear discrepancy between the participants' subjective impression of understanding the task and the actual analytical depth of their answers. Many contributions focused on obvious problems such as inefficient loops or unclear variable names, while more complex logical errors — such as the incorrect handling of negative differences — were frequently overlooked. In most cases, the efficiency assessment was superficial. While many respondents identified unnecessary operations and redundant structures, a more in-depth complexity analysis was largely lacking. Readability was also assessed ambivalently: While some participants criticized the poor naming of variables and the lack of comments, others rated the same code as sufficiently understandable. This discrepancy highlights that readability is highly dependent on subjective assessments and therefore should not be used exclusively as a measure of code comprehension.

The logical errors in the code were only partially identified. While some participants noted that the initialization of variables and the treatment of negative differences were problematic, a systematic analysis was lacking in most cases. This suggests that the ability to critically examine semantic correctness is not yet sufficiently developed for many students. Instead, they often limited themselves to identifying surface features such as unnecessary calculations.

These results suggest that code comprehension is a multi-stage process that requires diverse skills and cannot be adequately captured by focusing on individual dimensions. Many participants appear to have followed a "best-effort" strategy: as soon as they identified a glaring error, they ended their analysis without systematically examining further aspects. This provides a methodological clue for future research: While open-ended response formats encourage differentiated feedback, the results are highly dependent on the individual thoroughness of the participants.

To more accurately capture the various dimensions of code comprehension, it would

be useful to structure the task more closely. Instead of an open analysis, for example, multiple-choice sub-questions could be used that explicitly address logical errors, efficiency issues, and readability aspects. This would increase both the comparability of the answers and improve the validity of the results.

Overall, the study shows that the task is fundamentally suitable for stimulating critical thinking in the sense of code comprehension. However, it is clear that not all intended skills were assessed equally. While efficiency issues were relatively easily identified, the identification of logical errors and in-depth reflection on readability and comprehensibility often remained incomplete. This underscores the need to further develop task formats to enable a more systematic and differentiated assessment of competencies.

7.1.12 Fr5x3 - Code Improvement

The analysis of the accompanying survey results indicates that the task was overall clearly understood. Even though some students did not even start to work on the task, the responses show that this is less due to ambiguities in the task but rather to individual factors such as time management or test strategy. The high level of agreement regarding the clarity and unambiguity of the task shows that misunderstandings in the formulation can be ruled out. In addition to the clarity of the phrasing, the difficulty seemed to match the students' level of knowledge and competence, since the majority either agreed or strongly agreed to the given statement. This suggests that the task was neither too demanding nor too little demanding and is therefore, in principle, suitable for assessing the students' intended abilities. The rating of DeFr5x1.3 showed similar results and therefore seems to have been appropriate in regard to available time. Also, the small number of technical issues makes it clear that external conditions hardly played a role. The one student that further specified their problem claimed to have had trouble with the given time. But it is likely that their opinion is already represented in the formerly discussed rating. This means that it can be ruled out that technical hurdles had a systematic influence on the results. Overall, the feedback suggests that the task was carried out under methodologically favorable conditions: it was clearly formulated, appropriately difficult, and not distorted by external factors such as technical problems or time pressure.

The task aimed at testing the critical thinking of students, which was further specified to rating the efficiency and code readability. For this, students were given a logically flawed, inefficient, and hard-to-understand program. The former task tested identification of errors and was then expanded by requiring the development of suggestions for improvement and optimization of the code in terms of comprehensibility and robustness.

A central element was examining the logical correctness. The code contained incorrect handling of negative differences, which illogically set variable values. Furthermore, a redundant loop was implemented that had no influence on the result. These items required participants to identify logical errors and critically examine inefficient structures. Furthermore, the use of vague variable names made readability difficult, meaning the task also tested the ability to conceptually understand the code and formulate suggestions for improvements in understandability.

The analysis of the answers shows that the students recognized these intended aspects to varying degrees. Of the 17 submitted solutions, three were completely correct, eight were partially correct or incomplete, and six were inadequate or incorrect. Core problems, such as unnecessary loops, faulty handling of negative differences, and lack of readability, were addressed by a high number of students. Students particularly frequently suggested removing redundant structures, using absolute differences, and using more descriptive variable names. In some cases, the use of comments was also recommended.

The analysis of the partially correct answers reveals that basic approaches were captured but not always fully developed. For example, while removing the redundant loop was often suggested, the logical problems associated with negative differences sometimes remained unacknowledged or were only superficially addressed. This heterogeneity suggests that the task required not only superficial but also in-depth critical thinking and addressed different thinking styles.

Overall, the results confirm that the task covers essential facets of code comprehension. It promotes the ability to analyze errors, identify inefficient structures, and improve readability—central components of critical thinking in a programming context. At the same time, the responses show that the students were able to recognize these aspects and derive constructive suggestions for improvement to varying degrees.

From a didactic perspective, several opportunities for improvement can be identified. For example, clearer instructions could help focus attention more specifically on the aspects of logic, efficiency, and readability. Furthermore, the integration of additional test cases with varying levels of difficulty would further challenge critical thinking and test the robustness of proposed solutions. An interactive debriefing session, in which students reflect on and justify their considerations, could also deepen the assessment of thought processes. Finally, the task could be expanded by scaling it to include several smaller errors or optimization tasks to enable a broader representation of competencies in the area of code comprehension.

In summary, the evaluation shows that the task was generally suitable for assessing the intended facets of code comprehension by encouraging students to analyze errors and propose improvements in efficiency and readability. However, the results also demonstrate that some students identified only superficial problems, while others identified deeper logical and structural aspects. This confirms that the task is a suitable tool for differentiating between different levels of competence in the areas of critical thinking and code comprehension.

7.2 Key Findings and Interpretation

In the following, the findings already presented in detail are summarized and classified with regard to test design and code comprehension.

7.2.1 Data and Framework

Time
Looking at the recorded completion times, the large range between the shortest and longest times required is particularly striking. Since some of the responses come from participants who, according to their own statements, did not begin the task, the unadjusted mean is only of limited significance. The adjusted average, which takes into account responses for which definitive completion seems plausible, provides a more realistic picture. Nevertheless, it can be stated that the time values as a whole should not be used in this study to evaluate task quality, since the limited maximum time of the test design was poorly designed for uniform completion time, and the different task distribution for the students is sometimes responsible for such large differences. However, since most tasks were rated relatively positively with regard to the time available, it can be assumed that the few negative ratings came from students who only received the task at the end. Tasks at the end of the processing time can therefore only be completed to a limited extent and not with equal quality, which means that answers/reasonings are not fully comparable.

Issues while conducting the test or unforeseen outcomes
During the test administration and ultimately the data analysis, we encountered unexpected difficulties. Firstly, despite the introduction at the beginning, some things were misunderstood or overlooked by the students. The task instructions were often not read correctly. For example, students stated in the debriefing questions that they had not started the task, even though their answers actually indicated something else. This could possibly have been used to mask a lack of knowledge, perhaps because they didn't know the correct answer or abandoned the task as a result. Some students also did not read the personality question (**Question P2**) completely. The instructions state that students should list any programming languages they have learned besides Python, since Python was already covered in **Question P1.3**. Nevertheless, some students still listed Python here. In some cases, no answer was submitted in the main task, but an explanation was provided, some of which were even correct. In these cases, students may have simply forgotten to give an answer. The problem then lay in the evaluation and the appropriate handling of such results, which was not defined beforehand.

Although the instructions provided information about permitted resources, students still used unauthorized resources. During the feedback sessions, it was admitted that ChatGPT had been used to have the code explained. This student's answers therefore actually contaminate the evaluation, as they do not provide a reliable picture of the student's understanding of the code. Furthermore, there was one submission (unclear whether it was from the same person) where the answers also indicated ChatGPT. The student stated that he had a mediocre experience in programming in general and a rather poor experience with Python and the logical programming paradigm. Nevertheless, they achieved comparatively good results (seven correct answers without explanation and three with explanation). Furthermore, the explanations were mostly very detailed and grammatically and spelling-wise correct, which was an exception among the other students' explanations. Here, too, the handling of the results is questionable, but they were included for the time being. This ultimately indicates a superficial reading or misinterpretation of the instructions.

The findings in this section demonstrate that data cleansing is essential for processing and interpreting the results. Furthermore, the findings point to problems with (understanding) the instructions.

7.2.2 Answer and Justification Rate

The relatively low response and explanation rate could be due to several factors. First, it was mentioned several times that Python as the chosen programming language was a problem. However, the results of the personality questions show a less negative, mixed picture. There were certainly some students who considered themselves experienced with Python. Accordingly, the reason for incorrect and few answers/explanations is only partly due to this. By contrast, the difficulty of the tasks themselves could also be a factor in the poor ratings. There were certainly some tasks that were considered more difficult (e.g., **Fr2x2,** partly **Fr2x3**), but their proportion of the total tasks was not particularly large. Therefore, this is not the sole reason for the results. Some students stated in their feedback that the focus for them was on the answers and therefore deliberately ignored or neglected the explanations. Furthermore, the time factor may have led to fewer explanations being given, as these were more time-consuming than answering the questions alone. Accordingly, omitting the justifications might be a strategic behavior ("only the result counts"). Overall, one could say that the justification rate also points to additional cognitive load or instructional misunderstandings.

Quality of Justifications
Good justifications were usually characterized by a structured approach that analyzed the code step-by-step and derived higher abstractions from the underlying details and circumstances. Poor justifications, on the contrary, were usually very superficial and rather fragmented in their information and the aspects considered. They often relied on rule-based insights without deeper understanding.

However, these justifications also provide insights into the students' thought processes and may reveal gaps in their knowledge, their analytical understanding of details, or their ability to abstract.

7.2.3 Task Characteristics

Suitability of the Tasks
One question that arose during the design of the tasks is whether a task is ultimately appropriate for testing code comprehension or not. Due to the factors already mentioned, little can be said about the measurement object. However, this can be assessed based on the context of the tasks.

Based on the debriefing questions, it was clear that most of the tasks were quite appropriate in terms of comprehensibility. However, there were a few exceptions in this regard. For example, **task Fr2x2** and **Fr5x1** were rated as rather incomprehensible.

The same applies to task **Fr3x1**. This was suitable in its basic structure in the term paper Bartl, 2024, so perhaps the additional material, i.e., the code, was the problem here. This code should therefore possibly be adjusted to be more suitable. This could potentially

reduce the test validity. The ultimate reason for this should be discussed. Was the difficulty level too difficult, or was it due to other contextual factors? Are some difficult tasks still useful in order to differentiate between very good performance in the tasks?

Task Design
A large portion of the tasks focused on analytical skills and the ability to think abstractly. Critical thinking was tested far less frequently and in fewer facets. Accordingly, a greater variety of tasks should be created in the future regarding the facets to be tested.

Python as a Programming Language
When answering the tasks, it was noticeable that programming language-specific aspects were still being tested, which is fundamentally unsuitable, as these gaps are likely to be filled with knowledge from other programming languages. Also, students who have claimed not to have any experience with Python might have experienced learned helplessness. An effect where repeated failure - or in this case the apparent certainty that the task cannot be solved due to a lack of knowledge - may lead to less motivation or a negative result (Seligman, 1972).

7.2.4 Typical Errors and Strategies

The analysis of the responses shows that typical error sources and processing strategies can be identified. A common pattern was the excessive use of heuristic or pattern-based strategies. Students focused on superficial similarities in the code without systematically examining the underlying logic. This led to misinterpretations, especially in more complex tasks.

Another key error area concerned iteration and syntax details. Misunderstandings regarding starting values, loop conditions, or the exact execution steps occurred regularly. Such details appear to pose a particular hurdle for many participants and illustrate that not only abstract understanding but also precise, detailed knowledge is required.

Furthermore, the influence of other programming languages became apparent. Students with previous experience in languages with different conventions (e.g., regarding indexing or syntax) transferred their prior knowledge to Python, which sometimes led to misinterpretations. This highlights a central fairness issue: language-specific peculiarities influence understanding and can distort the results. Overall, these observations also reflect differences in the cognitive facets of code comprehension. While analytical skills—such as recognizing local structures — were used comparatively frequently, deficits were evident in abstraction, i.e., linking individual details to a holistic understanding.

7.2.5 Cross-Task Findings

The results suggest that the participants' strengths lay primarily in analytical detail. Many were able to examine individual code components in isolation and understand their function. The analysis of the results shows that the participants demonstrated strengths primarily in local code analysis and analytical detail. Individual statements or smaller code fragments could usually be correctly understood. However, significant weaknesses were evident in the area of abstraction: recognizing higher-level structures and intentions

of the code was rarely achieved. This limited the holistic understanding of program intent. Many students had difficulty grasping the program's higher-level intentions or relating multiple code components to one another. Misunderstandings were particularly common in the area of iterations and loop conditions, for example, by overlooking starting values or misinterpreting termination conditions.

Furthermore, the results suggest that previous experience with different programming languages influenced the quality of the answers. Participants who were already familiar with languages such as Python often interpreted specific structures more easily, while students with other language backgrounds were more prone to misunderstandings. This suggests that code comprehension is not only a universal cognitive ability but is also strongly influenced by language-specific experiences. For the evaluation and design of assessments, this means that differences in programming background can potentially lead to bias and should be considered accordingly.

A tendency toward heuristic strategies was also evident: instead of following code step by step, many resorted to superficial "pattern matching." This sometimes led to correct answers, but often also to incomplete or incorrect answers. Furthermore, the data show that cognitive processes are closely linked to motivational factors. For example, a correlation was found between processing time, motivation, and error tolerance: Longer processing times did not necessarily lead to better results but did indicate greater perseverance or willingness to learn. Feedback could enhance this learning effect but was not systematically used in the current design. For future assessments, this results in the implication of using feedback mechanisms more specifically to support learning processes during test completion.

7.2.6 Conclusion

These insights are a good starting point. On the one hand, they provide initial insights into the thought processes, etc., and on the other, they allow us to assess the quality of the test conducted and ultimately identify and implement necessary changes.

7.3 Refinement of the Criteria Catalog Based on Findings

The criteria are still sound in theory, but they require further refinement and stronger operationalization in light of practical data. One may argue that the test's first version was exploratory and therefore the subsequent version needs to be built using criteria and hypotheses.

7.3.1 Measured Variable (Code Comprehension)

During the survey, a methodological problem emerged regarding the multidimensionality of the competencies assessed. Although the dimensions of analysis, abstraction, and critical thinking were theoretically defined, they were not considered with equal intensity in the practical implementation. In particular, the facet of abstraction was assessed significantly less intensively than the detailed analysis. This created an imbalance in the measurement, which could lead to a distortion of the results. To optimize the instrument, a targeted

adjustment is therefore necessary. In the future, the items should represent the various facets more evenly. In particular, this means establishing a balance between micro-level tasks ("What does this line do?") and macro-level tasks ("What is the purpose of the code as a whole?"). This balance can ensure that both detailed analysis and abstraction are adequately measured and that the intended multidimensionality of the construct is methodically and reliably represented. This could be further supported by the use of guided questions, which aim to further direct the explanations toward the ability to abstract.

Additionally, the different manifestations among students suggest that the concept of code understanding exists at different levels. Models such as the SOLO taxonomy could therefore be used to further define the definition in order to clarify the various gradations and to better differentiate the evaluation of the answers.

7.3.2 Participants

One of the seemingly biggest problems within this thesis was the heterogeneous programming experience, especially language skills, among the students. Therefore, a preliminary survey regarding the participants' programming language experience is recommended, along with offering the option to choose between several languages. Accordingly, a higher degree of homogeneity within the participant group in the relevant factors would be appropriate for a repeat study.

7.3.3 Choice of Programming Language

As already stated, Python, a supposedly neutral language, turned out not to be entirely neutral (influences of other languages, misunderstandings in Python syntax). Therefore, it should be ensured that each participant was able to gain a certain level of experience in the tested programming language. Alternatively, several languages could be offered, but here too, one should ensure that each participant can choose a language they have already worked with. Ensure tasks don't rely on language-specific peculiarities (fairness issue).

7.3.4 Test Duration

Generally, a longer total time would be advantageous. Even though we gave the instruction not to rush while working on the task, students neglected that note and therefore probably did not take the full time they needed to adequately answer the tasks. So in the future, perhaps even no time limit would be best in order to truly see how long some students need to complete the tasks and to achieve truly meaningful and comparable results. The different orders of tasks used in this study would therefore also be irrelevant, and the tasks could, for example, be sorted by difficulty or randomized completely or in blocks.

7.3.5 Validity

A central problem arose with regard to the content validity of the instrument. In particular, there was no sufficiently clear distinction between items that primarily aimed to test analytical ability and those that aimed at abstraction. This vagueness carries the risk that

the intended cognitive facets will not be reliably captured in a differentiated manner, thus limiting the meaningfulness of the results.

Several measures are planned to improve content validity. First, prior validation by experts should be conducted to ensure the content fit of the items with the theoretically defined facets (content validity). Second, a clear mapping should be established that explicitly indicates which item addresses which dimension of code comprehension. Third, subsequent item analyses (e.g., difficulty index, discriminatory power) should be systematically integrated into the evaluation process to empirically verify the quality of the items and make adjustments if necessary.

7.3.6 Complexity

Another problem arose in the categorization of the tasks according to their complexity. It was difficult to assess the complexity during task design and therefore only became apparent ex post based on the debriefing questions and the results. As a result, some tasks appeared too difficult to the students, which could have negatively impacted their motivation and willingness to respond. To avoid this in the future, the findings from this study could be used. Furthermore, item difficulty could be classified ex ante using expert ratings and explicitly named (e.g., easy, medium, difficult). Tasks used here that were considered too difficult could also be adjusted accordingly. Rework or replace overly complex/confusing tasks (**Fr2x2**, possibly **Fr1x2**).

7.3.7 Feedback

Since the feedback function was not used in this study, possibly due to low extrinsic motivation, individual feedback could be more integrated (e.g., direct debriefing after a task block). This could be supported by immediate feedback after answering. However, direct negative feedback could also lead to a loss of motivation. Adaptive elements such as comparison with the average and explanations could provide further inspiration.

7.3.8 Task Types

Even though the design of this study avoided open-ended formats due to the scope and assessment effort, these could provide further insights into students' thought processes. However, the frequently used combination of multiple-choice and required reasoning could be expanded to include additional guided questions to further specify the purpose of the task. This could reduce random aspects of the answers and achieve more consistent data. Encourage step-by-step analysis to complement abstract reasoning. Use guiding questions to elicit both step-by-step and holistic reasoning.

To go deeper into the skills, guided questions may be useful to direct the students' answers and thinking to specific aspects of code comprehension. However, it was also quite beneficial because the unguided version offered some preliminary insights into the pupils' mental processes. Thus, the questions concerning the logic should be broadened and more focused to test more precisely.

7.3.9 Contextualization

Another problem arose with regard to the design of the context. In the previous design, this was considered secondary, potentially impairing participants' motivation. The lack of authenticity and relevance of the tasks may have led to the items being perceived as less appealing or practical, which in turn may have negatively impacted the willingness to complete the tasks and the validity of the results. For optimization purposes, authentic contexts will be given greater consideration in the future. This includes, in particular, the use of realistic code examples and small, practice-oriented problems. Furthermore, it is planned to more clearly emphasize the connection to the course content in order to increase perceived relevance and strengthen participants' motivation.

7.3.10 Response Format

Since a mix of digital and paper formats was only intended as an emergency, this was not systematically considered. Therefore, the comparability and fairness of the completion and responses from the two formats can be considered questionable. A uniform format would therefore be recommended, but students would need to be informed of this in advance so that everyone can make appropriate preparations and a certain degree of fairness can be achieved. However, this could reduce fairness in terms of the availability of appropriate equipment. If different formats are used, systematic recoding and comparability should be ensured.

7.3.11 Motivation

The conduct of this study offered little extrinsic incentive for participation, which may have had a negative impact on the answers, the justifications, and the respective quality of those answers. Therefore, appropriate incentives, such as participation points, bonus questions, certificates, or similar, should be offered in the future. Gamification of the test or a practical scenario could also increase intrinsic motivation.

7.3.12 Additional Materials

The additional materials used here were strongly related to the respective tasks. Codes were primarily used, and explanations or visual representations were provided where necessary. However, these caused problems in the paper formats because they were not transferred one-to-one and were therefore unusable. This should be taken into account in the future.

7.3.13 Error Tolerance and Assessment

A methodological problem arose in the initial grading rule, according to which an answer was only considered correct if an adequate justification was also provided. However, this strict requirement proved unsustainable in practice, as it excluded potentially valid answers and thus led to a bias in the results. Furthermore, very few justifications were provided overall.

For the further development of the assessment model, a distinction is therefore planned between the quality of answers and justifications. Furthermore, the use of established taxonomies (e.g., the SOLO taxonomy, Bloom's taxonomy) is recommended to more precisely classify the cognitive demand levels of the items and thus enable a more refined gradation of the performance assessment.

7.3.14 Objectivity

Since this study is the first of its kind and was only conducted once, generalizability cannot be assumed. Accordingly, the study should first be adapted and then replicated with other cohorts.

7.3.15 Fairness

With regard to linguistic fairness, it was shown that the use of English was largely unproblematic for the participants. However, the question of programming language fairness is more critical. Different prior experiences with certain languages or paradigms can create systematic advantages or disadvantages and thus impair the comparability of the results. Ensure tasks don't rely on language-specific peculiarities. To ensure fairness, it is therefore planned to explicitly expand the term beyond the linguistic level to include the participants' coding background. This requires item design that ensures that different prior experiences do not lead to systematic biases. In addition, the format issue presents a further challenge: Furthermore, fairness could not be ensured for students who used the paper format compared to the rest of the students, as interaction options, presentation, and editing contexts varied.

7.3.16 Falsification

Another problem concerns the potential use of external AI tools (e.g., ChatGPT) during the course of the exam. Since their use was not explicitly excluded, cases in the survey indicated the possible use of such tools. This calls into question the validity of the results, as it remains unclear whether the performance corresponds to the participants' actual competencies.

Several adaptation measures are planned. First, clear instructions should be formulated that unambiguously specify which tools are permitted during the exam. Second, technical safeguards such as browser lock systems or individual tests can be used to prevent unauthorized assistance. Third, the use of complementary detection heuristics seems sensible, for example, by analyzing response style, unusual linguistic perfection, or conspicuous orthography to identify potentially AI-generated posts.

7.3.17 Reliability

Another methodological problem was the limited reliability of the instrument. This was negatively influenced by several factors: the heterogeneity of the participants, an overall low justification rate, and the inconsistent or incorrect use of the "Task Not Started" category.

This weakened the internal consistency of the measure and reduced the comparability of the results.

Several adjustments are planned to improve reliability. First, the instructions should be more standardized, particularly by precisely defining when the "Task Not Started" category should be applied. A demonstration of correct answering at the beginning could also be helpful. Second, a systematic item analysis (e.g., calculation of Cronbach's alpha) can be integrated into the evaluation process to empirically verify the quality of individual items and examine their contribution to the overall reliability of the instrument.

7.4 Implications for Code Comprehension Assessment

In the following, the implications of the results are presented explicitly for the area of code comprehension.

7.4.1 Multidimensionality of Code Comprehension

The results show that the participating students demonstrated strong competencies in detailed analysis, while their overall abstraction skills were weaker. This reinforces the assumption that code comprehension should not be tested in a single dimension (e.g., "What does line X do?"). Instead, it requires tasks that measure both local analysis (such as syntax, individual lines, or parts of the code) and global abstractions (the purpose and overall structure of the code). These levels (along with critical thinking) should therefore be tested in a more balanced way in the future.

7.4.2 Role of Justifications

Without explicitly requested justifications, many participants tend to simply provide answers. However, since the provided justifications provide good insights into students' thought processes and approaches (e.g., analytical vs. heuristic), they prove helpful in making statements about potential mental models and revealing cognitive strategies. Accordingly, the evaluation of these justifications should be further systematized and specified in order to be able to validly measure and evaluate their quality (e.g., using taxonomies).

7.4.3 Task characteristics and Level of Difficulty

Some of the tasks posed greater challenges for students than others due to their structure or requirements. One might assume that the difficulty is related to the level of thoroughness required. Accordingly, the difficulty level should not arise randomly but rather be deliberately modeled in advance using expert ratings, piloting, or item analysis. Research could therefore more specifically address the question of how to reliably predict difficulty levels in code comprehension.

7.4.4 Influence of Programming Languages

The results indicated that language-specific prior experience partially influenced students' understanding and response quality. This could indicate that code comprehension, as defined here, is not language-neutral. Further investigations could therefore examine the ultimate influence of other programming languages and, based on this, develop cross-linguistic, fairer task formats (e.g., pseudo-code, visual representations).

7.4.5 Error and Strategy Analysis

The typical errors (heuristic pattern recognition, ignoring loop details, transferring rules from other languages) indicate a multitude of recurring errors, which can be just as useful for deriving mental models as positive ones. To better utilize these, it is recommended to develop error categories and strategy models for more in-depth analysis.

7.4.6 Methodological Standards

The analysis shows that problems such as the occurrence of "task not started" or uncertainties in assessment can create systematic biases in the research context. Such findings clearly demonstrate that current methodological approaches are insufficient to ensure robust and comparable results. This implies that research urgently needs common methodological standards. A framework for test design – comparable to the established procedures in intelligence or literacy tests – could remedy this situation. Such a framework would need to contain clear guidelines, for example, on the definition and design of task formats, on uniform assessment systems, and on difficulty modeling. Only through this standardization can the comparability of results be increased, methodological artifacts reduced, and the long-term validity of empirical findings ensured.

7.4.7 Role of Artificial Intelligence

With the widespread adoption of large language models and AI, a study like this one carries the risk of distorting results due to improper use. Exceptional cases in this study already indicated such use. Accordingly, future code comprehension assessments should consider AI influences and address the recognition of these responses. Furthermore, one could discuss whether the ability to critically use AI could become a part of code comprehension.

7.4.8 Implications for Test Design and Code Comprehension

For test design, this requires more structured tasks through guiding questions, more careful planning of the placement of individual items in the test, and ensuring fair difficulty grading. This can reduce bias and enable valid conclusions about participants' abilities. For research, this means that code comprehension should not be viewed as an isolated skill. Rather, it should be understood as an interplay of analytical thinking, abstract reasoning, and critical judgment. Only by considering these dimensions together can an adequate picture of the underlying cognitive processes be gained.

7.4.9 Aim of the Research: A Common Concept of Code Comprehension

Since the test is only an exploratory attempt with a previously defined definition of code comprehension, the problem remains that there is still no uniform definition, and multidimensional approaches are necessary. Therefore, research based on this study could further focus on theory building. This includes questions such as "Which dimensions does code comprehension encompass?" "How are analysis, abstraction, experience, language, etc. related?" or "Which assessment formats best represent these dimensions?"

7.4.10 Definition of Code Comprehension

At the beginning of this thesis, a possible definition of code comprehension was presented, which can now be further refined based on the findings after conducting the test.

To further refine the definition theoretically, it would be advisable to define the previously rather abstract phrase "analyze, interpret, and evaluate" more precisely. In particular, the term "interpret" could be defined more clearly, for example, as "to discover the meaning of code elements in their respective context." This would make the content dimension clearer and more easily delineated.

Furthermore, the three facets of the definition can be systematically linked to the goals of code comprehension:

> Analytical thinking is required to understand what the code actually does.
>
> Abstraction makes it possible to recognize the structure and intention of the code.
>
> Finally, critical thinking is necessary to assess whether the code actually achieves what it is intended to achieve.

This more precise connection between cognitive facets and goals creates a more consistent understanding of the term. At the same time, the transfer to research becomes clearer: Such a refined definition provides a robust basis for the development of testing instruments as well as for the empirical investigation of code comprehension as a complex, multidimensional process.

Furthermore, the definition could be further differentiated from similar concepts such as "code reading" (purely syntactic) or "program comprehension" (often broader, including design or architectural aspects). This could demonstrate that the definition of code comprehension used is precisely tailored to the understanding of source code. Based on the findings, the following revised definition of code comprehension now emerges:

Code Comprehension *describes the process(es) by which people analyze, interpret, and evaluate source code. Interpretation particularly involves deriving the meaning of code elements in their respective context. The goal of this process is to:*

1 *understand what the code does*

   (includes retracing the functions, processes, and results of the code),

2 *assess whether the code does what it is supposed to do*

(includes checking whether the code conforms to the stated requirements or its functionality),

3 *recognize its structure and intention*

(includes checking the clarity, readability, and syntax of the code).

These goals are closely linked to three cognitive facets:

Analytical thinking is required to understand what the code actually does.

The ability to abstract enables the recognition of the structure and intention of the code.

Finally, critical thinking is necessary to assess whether the code actually does what it is supposed to do.

This defines ***code comprehension*** *as a multidimensional cognitive process that can be distinguished from related concepts: In contrast to code reading (which is primarily syntactically oriented) or program comprehension (which also includes design and architectural aspects), the definition used here focuses precisely on understanding the source code itself.*

Chapter 8

Limitations of the Study and this Thesis

We encountered several key limitations while conducting this study that we wish to note: A central problem seemed to be the use of digital and paper versions. Since the paper version only served as an emergency plan for students without digital devices, the execution and evaluation were not planned out well enough. In addition to the question of comparability, the presentation of some tasks posed problems because not everything was displayed correctly as in the digital version. This led to wrong or incomplete answers and therefore, distorted results.

In addition, the test structure showed unclear instructions in several places, which ultimately led to incorrect processing by the students and problems during the evaluation. Similar to this, some terms that were used in the task proved to be interpreted in many different ways and therefore led to students answering questions in terms of their own definition. Consequently, it can not be guaranteed that some aspects (e.g., technical problems) were tested and ruled out in terms of their intended definition.

Beyond that, the choice of Python did not prove to be as neutral as assumed while designing the test. Many students stated that they had problems regarding the programming language, which ultimately led to knowledge gaps that do not have their roots in understanding the code but rather in having trouble with the language. Additionally, it can not be ruled out that the use of a programming language that someone has not worked with, led to a decrease in motivation and maybe even learned helplessness (Seligman, 1972).

Another methodological problem arose in the debriefing. Since the study used a mostly exploratory approach, it was not determined in advance which values of the debriefing questions should be classified as “good” or “negative”. Besides that, there was no defined procedure for students that stated, on the one hand, not to have started the task. But on the other hand, they still rated the debriefing questions of other aspects. A clear definition of assessment criteria — such as thresholds for average scores or a classification of tasks according to difficulty levels based on debriefing scores — would have been useful here.

Instead, a more exploratory approach was chosen, in which observations were first made and conclusions were subsequently drawn.

Regarding the content of the test, it can be said that it mostly tested analytical skills as well as the ability to abstract, while only partially testing critical thinking. A higher level of diversity within task types and their aims seems necessary to evenly test all of the facets of the proposed code comprehension definition. Also, the unexpectedly low number of answers and justifications provided only limited insights into the reasoning. One might assume that even more detailed explanations and responses would suggest other results regarding the facet testing, or that a higher number of explanations would either expand, confirm, or perhaps even falsify the findings.

Finally, it must be taken into account that the approach of this study was strongly heuristic. Furthermore, this was the first attempt of its kind based on the chosen definition. Whether it actually captures all aspects of code comprehension remains an open question. A second, adapted test would have offered the opportunity to correct design weaknesses and increase the reliability of the results. However, due to time and organizational constraints, a repeat test was not possible. External factors such as clues from participant feedback or potential cheating, which is difficult to prove, also pose additional uncertainties that could be enlightened in another execution.

Chapter 9

Conclusion and Future Work

9.1 Summary of Contributions

This study makes several contributions to research and practice in the field of code comprehension, as well as to the further development of suitable test designs.

A key contribution lies in the theoretical foundation and clarification of the term code comprehension. Based on the empirical findings, the definition proposed at the beginning could be refined and terminologically differentiated from related concepts such as code reading or program comprehension. Code comprehension was conceptualized as a multidimensional process that encompasses analysis, abstraction, and critical thinking. This differentiation contributes to a more consistent understanding of the term and provides a basis for future research and test developments.

Furthermore, the study provides empirical insights into the students' strengths, weaknesses, and strategies. While comparatively solid performance in the local, analytical examination of code was evident, significant deficits were evident in the areas of abstraction and global understanding. In addition, typical error sources and strategies were identified, such as the reliance on heuristic pattern recognition, the transfer of concepts from other programming languages, or difficulties in dealing with iterations and loops. These results enable the categorizing of typical error and strategy patterns and thus provide an empirical basis for differentiated diagnostic approaches.

A further contribution concerns the role of justifications in diagnostics. The analysis illustrates that justifications provide valuable insights into cognitive processes and mental models – regardless of whether the underlying answer was correct. While superficial justifications primarily reflect rule applications, systematic analyses and abstractions can be recognized in deep justifications. This attributes diagnostic potential to justifications that goes beyond the mere evaluation of results and necessitates standardization of their analysis and evaluation.

The work also highlights important implications for test design and fairness. Problematic task formats (e.g., **Fr2x2**, **Fr3x1**) and language-specific effects highlight the need for clear

instructions, greater task variety, and the development of language-independent formats, for example, through the use of pseudo-code or visual representations. This contributes to the discussion of fairness and comparability in code comprehension assessments.

In addition to content-related insights, methodological contributions are also highlighted. The study highlights the importance of data cleansing, instructional clarity, time management, and the reflective use of external tools such as ChatGPT. These aspects underscore the need for common methodological standards to ensure the validity and comparability of future studies. Furthermore, relationships between motivation and response behavior are demonstrated. Initial evidence suggests that motivation and perseverance — such as longer response times — can influence response behavior without necessarily leading to better results. Thus, code comprehension is understood not only as a cognitive but also as a motivation driven process.

Finally, the study also contributes to the practical development of assessments. The study provides concrete starting points for optimizing tasks, integrating critical thinking more strongly, using feedback mechanisms, and adapting the test structure. Also, the role of AI systems such as large language models as a potential source of bias is shortly addressed. Thus, the study may open up another field of research and simultaneously highlights the need to consider the critical use of AI as an integral component of code comprehension tests.

All things considered, it is evident that the study not only advances the construct's theoretical understanding and empirical groundwork, but also offers real support for the advancement of methods and real-world application of evaluations. As a result, it provides a wide range of opportunities for code comprehension research and growth.

9.2 Conclusion

The aim of this study was to test a previously conceived definition of code comprehension and to gain methodologically sound insights into how it can be measured. The following conclusions can be drawn from the results:

9.2.1 Code Comprehension and its Cognitive Facets (R1)

The study was supposed to test code comprehension as a combination of analyzing, interpreting, and evaluating in the proposed facets. While empirically it did show that participants were equipped with strong skills to analyze, the ability to abstract was significantly weaker. Critical thinking was hardly visible. It also became clear that "interpretation" was vaguely defined.

Even though the previously established definition of code comprehension could be consolidated in its core statement, code comprehension is now defined as a multidimensional process with three distinct facets: analytical thinking, abstraction, and critical thinking. The role of "interpretation" is specified as deriving the meaning of code elements in context.

9.2.2 Designing Tasks and Tests (R2)

The item pool in Python was supposed to cover both local and global aspects of code comprehension skills, within a specified time frame and using a standardized response format. Results show that abstraction was not tested well enough, while analyzing tasks prevailed. Task difficulties became apparent after the execution and could not be planned for. Heterogeneous knowledge (e.g., programming languages, syntax) led to biases. Answers were impacted by the limited time, lack of motivation, and unclear instructions. Assessment rules ("answer only valid with justification") proved to be too strict. Results indicate that valid and reliable measuring tasks of code comprehension have to meet the following requirements:

Task Balance: The combination of micro (e.g., analyzing single lines) and macro (e.g., comprehending the overall structure) tasks.

Structured Tasks: The use of guiding questions to support step-by-step analysis and/or abstract thinking.

Difficulty and Validity: The level of difficulties should be purposefully planned (e.g., through expert ratings, pilot studies, and item analysis) and clear and unambiguous assignments of items and the cognitive facets.

Reliability and Fairness: Standardized instructions, uniform formats, consideration of programming language experience, and AI usage.

Motivation and Contextualization: Authentic and practical tasks for a higher motivation and therefore better quality of results.

Assessment: Better defined parameters and standardized systems. Possibly using other taxonomies for support (e.g., SOLO-Taxonomy).

Those results show that measuring code comprehension without careful test planning would be biased and incomplete.

9.2.3 Suitability of the tasks used in the paper (R3)

Tasks were designed to show whether the proposed measuring instrument is practical and can provide differentiated information about code comprehension.

Results indicate that although the tasks provided insights into thought processes, the level of abstraction remained underexposed. Some tasks were too complex or unfair due to linguistic/syntactic peculiarities. Motivation and contextualization were insufficient, which had a negative impact on the results. At the same time, tasks allowed for the identification of typical error or strategy patterns, which enable conclusions about mental models and cognitive processes. The following adjustments should thus be taken into consideration: Revise tasks in a targeted manner (simplify or replace items that are too complex). Incorporate more realistic, practical contexts. Systematically use error and strategy models to analyze mental models. Pilot tasks in the future to check their level of difficulty and validity ex ante.

9.2.4 Further Insights

Code comprehension is not completely language-neutral; prior knowledge of certain programming languages might affect performance. The duration of the test, format, feedback, and motivation have a decisive impact on the quality of the results. The use of LLMs may affect the validity and therefore has to be taken into consideration for future studies. Methodological standardization is necessary to ensure comparability, reliability, and long-term validity of empirical studies.

9.2.5 Summary

With this thesis we planned to develop and test an initial instrument for measuring code comprehension with a precise, theoretically sound definition of code comprehension. In the end, strengths (insight into thought processes, identification of facets) and weaknesses (imbalance, fairness issues, validity restrictions) clearly emerged. It also highlights the need for multidimensional test tasks and identifies key methodological and practical criteria that influence measurement. It thus lays the foundation for further research and standardized testing procedures that enable code comprehension to be assessed in a reliable, valid, and differentiated manner.

9.3 Outlook for Future Research

This study represents a first step in the systematic assessment of code comprehension and, at the same time, makes it clear that this field of research is still in the beginning. The findings provide several perspectives for future work.

First, it seems crucial to repeat the test under adapted conditions in order to verify the effectiveness of the changes made and to compare the results with the findings reported here. This study can be understood as a kind of manual for subsequent studies, providing not only concrete suggestions for test design but also pointing out potential sources of error. A repetition with adapted tasks and validated debriefing questions could also reduce the frequency of follow-up questions in the test, as the task constructions would then already have been tested and the burden on participants could be reduced. Student feedback should also be systematically incorporated to increase the acceptance and fairness of the instrument.

Furthermore, it appears that a methodological expansion is needed. For example, think-aloud protocols or screen recordings could be used to gain deeper insights into mental models and strategies. Since this involves considerable effort, it is advisable to use a sample or combine it with post-task interviews to better relate processing strategies to the learners' prior knowledge. This should critically reflect on the extent to which think-aloud is actually representative of thought processes. A comparison of novices and experts could provide additional insights into differences in code comprehension.

Another important perspective concerns the language dimension of code comprehension. Future studies should examine whether tasks can also be provided in multiple languages

or in language-neutral formats such as pseudo-code to avoid biases caused by language-specific prior knowledge. Also, tasks could be improved by adding random information or varying levels of difficulty to make the measurement more accurate.

Furthermore, specific research questions arise:

> To what extent do previously learned programming languages influence the processing of code comprehension tasks by novices and experts?
>
> How can existing models, such as the SOLO taxonomy, be meaningfully combined with code comprehension tests?
>
> What role do motivation, contextual factors, and the use of AI systems (e.g., large language models) play in the processing and interpretation of tasks?

In conclusion, future research on code comprehension must consider code comprehension in a more multidimensional, cross-linguistic, and methodologically standardized way. Validation and evaluation approaches — especially for justifications — should be further developed. Likewise, motivational influencing factors and the potential effects of AI must be taken into account.

This work has created a conceptual and empirical framework that not only contributes to the practical development of test designs but also opens up a research agenda. Future studies are invited to further examine, supplement, and refine the definition developed here and the proposed instruments — thus systematically advancing research on code comprehension.

Bibliography

Ackerman, P. L., & Kanfer, R. (2009). Test length and cognitive fatigue: An empirical examination of effects on performance and test-taker reactions. *Journal of Experimental Psychology: Applied*, *15*(2), 163–181. https://doi.org/10.1037/a0015719

Asenov, D., Hilliges, O., & Müller, P. (2016). The effect of richer visualizations on code comprehension. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 5040–5045. https://doi.org/10.1145/2858036.2858372

Baecker, R. (1988). Enhancing program readability and comprehensibility with tools for program visualization. *Proceedings of the 11th International Conference on Software Engineering*, 356–366. https://doi.org/10.1109/ICSE.1988.93716

Bartl, L. (2024, November). *Evaluierung von code comprehension tasks zur objektiven und zuverlässigen messung der fähigkeiten von studierenden anhand der solo-taxonomie* [Term Paper, Faculty of Computer Science, Technical University of Chemnitz, Unpublished].

Bartl, L. (2025). Appendix of "designing and validating code comprehension tests: An empirical study on task design". https://doi.org/10.5281/zenodo.17152476

Beddow, P. A. (2018). Cognitive load theory for test design. In *Handbook of accessible instruction and testing practices* (pp. 199–211). Springer International Publishing. https://doi.org/10.1007/978-3-319-71126-3_13

Bednarik, R., Schulte, C., Budde, L., Heinemann, B., & Vrzakova, H. (2018). Eye-movement modeling examples in source code comprehension: A classroom study. *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, 2. https://doi.org/10.1145/3279720.3279722

Benander, A. C., Benander, B. A., & Pu, H. (1996). Recursion vs. iteration: An empirical study of comprehension. *Journal of Systems and Software*, *32*(1), 73–82. https://doi.org/10.1016/0164-1212(95)00043-7

Börstler, J., & Paech, B. (2016). The role of method chains and comments in software readability and comprehension—an experiment. *IEEE Transactions on Software Engineering*, *42*(9), 886–898. https://doi.org/10.1109/TSE.2016.2527791

Calvin, W. H. (1994). The emergence of intelligence. *Scientific American*, *271*(4), 100–107. http://www.jstor.org/stable/24942875

Cordova, D. I., & Lepper, M. R. (1996). Intrinsic motivation and the process of learning: Beneficial effects of contextualization, personalization, and choice. *Journal of Educational Psychology*, *88*(4), 715–730. https://doi.org/10.1037/0022-0663.88.4.715

Costa, J. A. S. D., & Gheyi, R. (2023). Evaluating the code comprehension of novices with eye tracking. *Proceedings of the XXII Brazilian Symposium on Software Quality*, 332–341. https://doi.org/10.1145/3629479.3629490

Denny, P., David H. Smith, I., Fowler, M., Prather, J., Becker, B. A., & Leinonen, J. (2024). Explaining code with a purpose: An integrated approach for developing code comprehension and prompting skills. *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, 283–289. https://doi.org/10.1145/3649217.3653587

Goodfellow, M., Booth, R., Fagan, A., & Lambert, A. (2025). Testing code comprehension using genai. *Proceedings of the 2025 Conference on UK and Ireland Computing Education Research*. https://doi.org/10.1145/3754508.3754532

Haiduc, S., Aponte, J., & Marcus, A. (2010). Supporting program comprehension with source code summarization. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, 223–226. https://doi.org/10.1145/1810295.1810335

Hattie, J., & Timperley, H. (2007). The power of feedback. *Review of Educational Research*, *77*(1), 81–112. https://doi.org/10.3102/003465430298487

Izu, C., & Mirolo, C. (2020). Comparing small programs for equivalence: A code comprehension task for novice programmers. *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, 466–472. https://doi.org/10.1145/3341525.3387425

Lewis, C. M. (2023). Examples of unsuccessful use of code comprehension strategies: A resource for developing code comprehension pedagogy. *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, 15–28. https://doi.org/10.1145/3568813.3600116

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 118–122. https://doi.org/10.1145/1140124.1140157

Mahdavi, M., & Abedjan, Z. (2020). Baran: Effective error correction via a unified context representation and transfer learning. *Proc. VLDB Endow.*, *13*(12), 1948–1961. https://doi.org/10.14778/3407790.3407801

Masters, G. N. (1982). A rasch model for partial credit scoring. *Psychometrika*, *47*(2), 149–174. https://doi.org/10.1007/BF02296272

Minelli, R., Mocci, A., & Lanza, M. (2015). I know what you did last summer: An investigation of how developers spend their time. *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, 25–35.

Moosbrugger, H., & Kelava, A. (Eds.). (2020). *Testtheorie und fragebogenkonstruktion* (3rd ed.). Springer Berlin Heidelberg.

Nawas, A. (2018). Contextual teaching and learning (ctl) approach through react strategies on improving the students' critical thinking in writing. *International Journal of Management and Applied Science*, *4*(7), 46.

Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., & Paterson, J. H. (2010). An introduction to program comprehension for computer science educators. *Proceedings of the 2010 ITiCSE Working Group Reports*, 65–86. https://doi.org/10.1145/1971681.1971687

Seligman, M. E. P. (1972). Learned helplessness. *Annual Review of Medicine*, *23*(1), 407–412.

Smith IV, D. H., Fowler, M., Denny, P., & Zilles, C. (2025a). Counting the trees in the forest: Evaluating prompt segmentation for classifying code comprehension level. *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 37–43. https://doi.org/10.1145/3724363.3729045

Smith IV, D. H., Fowler, M., Denny, P., & Zilles, C. (2025b). Redefining code comprehension: Function naming as a mechanism for evaluating code comprehension. *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, 44–50. https://doi.org/10.1145/3724363.3729097

Stapleton, S., Gambhir, Y., LeClair, A., Eberhart, Z., Weimer, W., Leach, K., & Huang, Y. (2020). A human study of comprehension and code summarization. *Proceedings of the 28th International Conference on Program Comprehension*, 2–13. https://doi.org/10.1145/3387904.3389258

Sweller, J. (2010). Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, *22*(2), 123–138. https://doi.org/10.1007/s10648-010-9128-5

Turner, R., Falcone, M., Sharif, B., & Lazar, A. (2014). An eye-tracking study assessing the comprehension of c++ and python source code. *Proceedings of the Symposium on Eye Tracking Research and Applications*, 231–234. https://doi.org/10.1145/2578153.2578218

Urhahne, D., & Wijnia, L. (2023). Theories of motivation in education: An integrative framework. *Educational Psychology Review*, *35*, 45. https://doi.org/https://doi.org/10.1007/s10648-023-09767-9

Vielsack, A. (2024). Meaningful highlighting - improving educational ides to enhance code comprehension for programming novices. *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 2*, 846–847. https://doi.org/10.1145/3649405.3659480

Wyrich, M. (2023). Source code comprehension: A contemporary definition and conceptual model for empirical investigation. https://doi.org/10.48550/arXiv.2310.11301

Wyrich, M., Bogner, J., & Wagner, S. (2023). 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv.*, *56*(4), 1–42. https://doi.org/10.1145/3626522

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, *44*(10), 951–976. https://doi.org/10.1109/TSE.2017.2734091

Appendices

Appendix A

Personality-Related Questions

Please respond to the following statements according to your experience.

There are no right or wrong answers - the important thing is to answer honestly..

Questions P1

Question 1.1

How do you rate yourself in terms of your programming experience compared to your fellow students?

very inexperienced				very experienced
☐	☐	☐	☐	☐

Question 1.2

How experienced are you with the logical programming paradigm?

very inexperienced				very experienced
☐	☐	☐	☐	☐

Question 1.3

How would you rate yourself in terms of your programming experience with the Python programming language?

very inexperienced				very experienced
☐	☐	☐	☐	☐

Question P2

Question

Which other programming languages do you consider yourself to have at least an intermediate level of proficiency in?

If it is more than 3, please specify only your 3 most familiar languages then add only the number of the remaining languages.

Appendix B

Item Pool

Item Fr1x1

Type: Bound Format: Multiple Choice with Justification
Ability to be tested: analytical skills and ability to abstract

Task

Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

```python
def process_data(data):
    result = [0] * len(data)
    for i in range(len(data)):
        if i % 2 == 0:
            result[i] = data[i] * 2
        else:
            result[i] = data[i] - 1
    final_value = 0
    for value in result:
        final_value += value
    return final_value


data_input = [3, 7, 2, 8, 5]
output = process_data(data_input)
print("Output:", output)
```

Options:
Select all that apply

a) It doubles the even-indexed elements of the list and decreases the odd-indexed elements by 1, then returns the sum of the modified list.
b) It creates a new list where even-indexed elements are doubled and odd-

indexed elements are decreased by one, then returns the modified list.

c) It modifies the input list by multiplying elements at even indices by 2 and subtracting 1 from odd-indexed elements, then calculates and returns the sum of that list.

Justification:

Correct Answer: a)

Possible Justification:

a) "It doubles the even-indexed elements of the list and decreases the odd-indexed elements by 1, then returns the sum of the modified list."

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr1x2

Type: Bound Format: Multiple Choice with Justification
Ability to be tested: ability to abstract

Task

How does the code behave if the input list `data_input` is empty?
Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

The code is the same as the code for task 1.1.

```python
def process_data(data):
    result = [0] * len(data)
    for i in range(len(data)):
        if i % 2 == 0:
            result[i] = data[i] * 2
        else:
            result[i] = data[i] - 1
    final_value = 0
    for value in result:
        final_value += value
    return final_value


data_input = [3, 7, 2, 8, 5]
output = process_data(data_input)
print("Output:", output)
```

Options:
Select all that apply

a) The code returns an error message.
b) The code returns 0.
c) The code returns None.

Justification:

Correct Answer: b)

Possible Justification:

b) "The code returns 0 when the input list is empty because no iterations occur, and the initialized `final_value` of 0 is returned unchanged."

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr1x3

Type: Bound Format: dichotomous task
Ability to be tested: analytical skills and ability to abstract

Task

Please select the flowchart that reflects the code.
Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

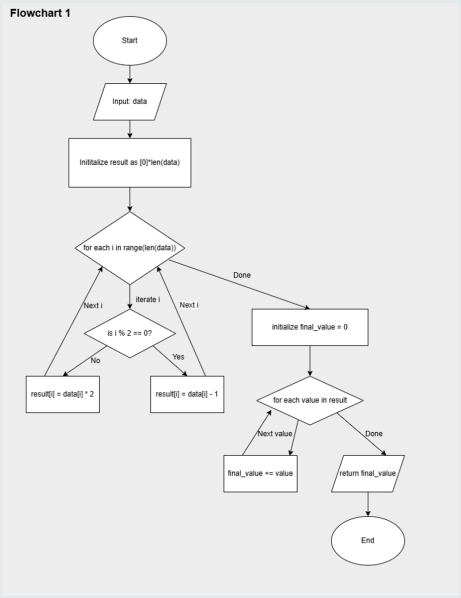
The code is the same as the code for task 1.1.

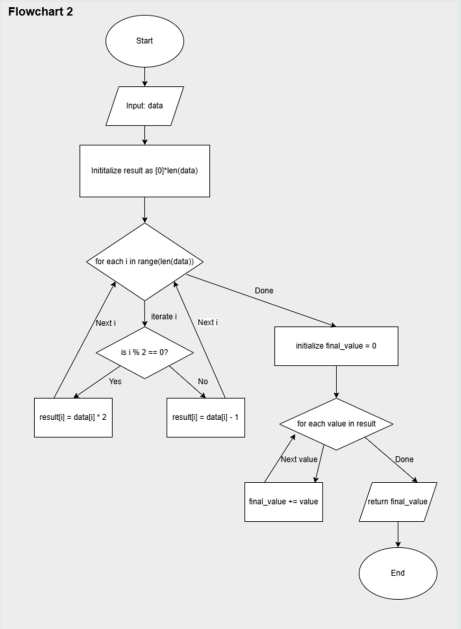
```
def process_data(data):
    result = [0] * len(data)
    for i in range(len(data)):
        if i % 2 == 0:
            result[i] = data[i] * 2
        else:
            result[i] = data[i] - 1
    final_value = 0
    for value in result:
        final_value += value
    return final_value


data_input = [3, 7, 2, 8, 5]
output = process_data(data_input)
print("Output:", output)
```

A flowchart for code is a visual representation of a program's logic, structure, and flow. It illustrates how a program processes input, performs operations, makes decisions, and produces output. It uses standardized symbols to represent various programming constructs like loops, conditions, and functions.

Options:
Choose one of the following answers





Flowchart 1
Flowchart 2

Justification:

Correct Answer: Flowchart 2

Possible Justification:

"Flowchart 2 is correct because it applies the condition if i % 2 == 0 to double the even-indexed values and subtract 1 from the odd-indexed values, which matches the code. Flowchart 1 is incorrect because it swaps the actions for the even and odd cases."

Evaluation Scheme: binary scoring

Answer	**Points**
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr2x1

Type: Bound Format: Multiple Choice with Justification
Ability to be tested: analytical skills and ability to abstract

Task

What is the output of the following code? Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

```python
def generate_string(numbers):
    result = ""
    i = 1
    while i < len(numbers) - 1:
        i += 1
        if numbers[i] % 2 != 0:
            result += "A"
        else:
            result += "B"
        result += "C"
    return result

numbers = [4, 7, 2, 5, 8, 4, 9]
output = generate_string(numbers)
print("Output:", output)
```

Options:
Select all that apply

a) BCACBCBCAC
b) BCBCACBCAC
c) CACBCACBC

Justification:

Correct Answer: a)

Possible Justification:

a) “The loop starts at i = 1 and increments i by 1, so itprocesses elements starting from numbers[2] (index 2). The numbers processed are 2 (even), 5 (odd), 8 (even), 4 (even), and 9 (odd)." Even numbers result in the addition of "B" and odd numbers in the addition of

"A". After every loop a "C" is added."

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr2x2

Type: Bound Format: Matching task
Ability to be tested: analytical skills and ability to abstract

Task

Please assign the following inputs to their corresponding outputs.
Please justify in 1-3 sentences why your answer is correct. You can only get one point, when everything is assigned correctly and justified.

The code is the same as the code for task 2.1.

```
def generate_string(numbers):
    result = ""
    i = 1
    while i < len(numbers) - 1:
        i += 1
        if numbers[i] % 2 != 0:
            result += "A"
        else:
            result += "B"
        result += "C"
    return result

numbers = [4, 7, 2, 5, 8, 4, 9]
output = generate_string(numbers)
print("Output:", output)
```

Inputs	Outputs

`A = [4, 7, 2, 5, 8, 4]`
`B = [6, 3, 5, 8, 2, 4]`
`C = [7, 9, 1, 3, 8, 6]`
`D = [3, 6, 2, 1, 7, 8]`
	`1 = "BCACACBC"`
	`2 = "ACACBCBC"`
	`3 = "BCACBCBC"`
	`4 = "ACBCBCBC"`

Justification:

Correct Answer:

A = 3
B = 4
C = 2
D = 1

Possible Justification:

a) “The loop starts at i = 1 and increments i by 1, so itprocesses elements starting from numbers[2] (index 2). The numbers processed are 2 (even), 5 (odd), 8 (even), 4 (even), and 9 (odd)." Even numbers result in the addition of "B" and odd numbers in the addition of "A". After every loop a "C" is added.”

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr2x3

Type: Bound Format: Multiple Choice with Justification
Ability to be tested: analytical skills and ability to abstract

Task

Which of the following statements about the function is/are true?
Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

The code is the same as the code for task 2.1.

```python
def generate_string(numbers):
    result = ""
    i = 1
    while i < len(numbers) - 1:
        i += 1
        if numbers[i] % 2 != 0:
            result += "A"
        else:
            result += "B"
        result += "C"
    return result

numbers = [4, 7, 2, 5, 8, 4, 9]
output = generate_string(numbers)
print("Output:", output)
```

Options:
Select all that apply

a) The function processes all elements in the input list.
b) The function always adds the same number of "C" characters as there are numbers in the input list.
c) The function skips the first element and only processes numbers starting from the second element.
d) The function correctly handles an input list with fewer than three elements by returning an empty string.

Justification:

Correct Answer: c) / d)

Possible Justification:

c) "The function skips the first element and only processes numbers starting from the second element."

d) “The function correctly handles an input list with fewer than three elements by returning an empty string.”

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr3x1

Type: Free Format: Short essay
Ability to be tested: analytical skills and ability to abstract

Task

Give the method a meaningful name that more clearly describes what it does.

Please justify your answer in 1-3 sentences. You will only get a point for the answer and the justification.

```
1  def process_input(v):
2      i = 0
3      j = len(v) - 1
4
5      while i < j:
6          if v[i] < v[j]:
7              v[j] = v[j] - v[i]
8          elif v[i] > v[j]:
9              v[i] = v[i] - v[j]
10         else:
11             i = i + 1
12     return v
```

Name:

Justification:

Correct Answer:

"reduceArrayValues"
"iterativeValueReduction"
"convergeArrayEnds"
"pairwiseValueAdjustment"
"equalizeArrayEnds"

Possible Justification:

"The method repeatedly compares the values at the two ends of the array and reduces the larger one by the smaller until they converge or become equal, which explains names like "reduceArrayValues" or "equalizeArrayEnds."

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr3x2

Type: Bound Format: Multiple Choice with Justification
Ability to be tested: analytical skills, ability to abstract and critical thinking

Task

Which of the tests uncovers an issue in the source code?
Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

The code is the same as the code for task 3.1.

```python
def process_input(v):
    i = 0
    j = len(v) - 1

    while i < j:
        if v[i] < v[j]:
            v[j] = v[j] - v[i]
        elif v[i] > v[j]:
            v[i] = v[i] - v[j]
        else:
            i = i + 1
    return v
```

Options:
Select all that apply

a) Input: v = [14, 1, 42]
b) Input: v = [1, 0, 18]
c) Input: v = [1, 13, 1]

Justification:

Correct Answer: b)

Possible Justification:

b) "b, because 0 is not processed."

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr4x1

Type: Bound Format: Multiple Choice with Justification
Ability to be tested: analytical skills, ability to abstract and critical thinking

Task

Which of the three code snippets fulfills the following specification?

The function should replace all negative numbers in a list with the value 0. All positive numbers and 0 remain unchanged.

Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

```python
#Snippet 1
def replace_negatives_v1(lst):
    for i in range(len(lst)):
        if lst[i] < 0:
            lst[i] = 0
    return lst


#Snippet 2
def replace_negatives_v2(lst):
    result = [0] * len(lst)
    for i in range(len(lst)):
        if lst[i] < 0:
            result[i] = 0
        else:
            result[i] = lst[i]
    return result


#Snippet 3
def replace_negatives_v3(lst):
    for i in range(len(lst)):
        if lst[i] > 0:
            lst[i] = lst[i] + 1
    return lst
```

Options:
Select all that apply

a) Snippet 1
b) Snippet 2
c) Snippet 3

Justification:

Correct Answer: Snippet 1 & Snippet 2

Possible Justification:

a) "Snippet 1: This function goes through each element of the list and replaces it with 0 if it is negative. It fulfills the specification because all negative numbers are replaced by 0 and all other numbers remain unchanged."

"Snippet 2: Here, too, negative numbers are replaced by 0, which meets the specification."

"Snippet 3: In this version, only the positive numbers are changed by incrementing them by 1. It does not correspond to the specification, as only positive numbers are changed and not the negative ones, as required in the specification."

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr5x1

Type: Bound Format: Multiple Choice with Justification
Ability to be tested: analytical skills, ability to abstract and critical thinking

Task

The code is supposed to do the following:

Calculate the largest difference between consecutive numbers in a list.

Does it fulfill this function?

Please justify in 1-3 sentences why your answer is correct and why the others are not. You will only get a point for the answer and the justification.

```
def max(numbers):
    m = 0
    total_elements = len(numbers)

    for i in range(total_elements - 1):
        d = numbers[i + 1] - numbers[i]
        for j in range(1):
            temp_d = d * 1
        if d > m:
            m = d
        elif d < 0:
            m = d
    return m

numbers = [1, 5, 3, 9, 2]
output = max(numbers)
print("Max difference:", output)
```

Options:
Choose one of the following answers

a) Yes, it does.
b) No, it does not.

Justification:

Correct Answer: b)

Possible Justification:

a) "No, the code does not fulfill its intended function because it incorrectly sets `max_diff` to negative differences, which leads to an incorrect result."

Evaluation Scheme: binary scoring

Answer	Points
Wrong answer with wrong or without justification	0
Wrong answer with right justification	0
Correct answer with wrong or without justification	0
Correct answer with plausible justification	1

Item Fr5x2

Type: Free Format: short essay
Ability to be tested: critical thinking

Task

How would you evaluate the efficiency and readability of the code?

The code is the same as the code for task 5.1.

```
def max(numbers):
    m = 0
    total_elements = len(numbers)

    for i in range(total_elements - 1):
        d = numbers[i + 1] - numbers[i]
        for j in range(1):
            temp_d = d * 1
        if d > m:
            m = d
        elif d < 0:
            m = d
    return m

numbers = [1, 5, 3, 9, 2]
output = max(numbers)
print("Max difference:", output)
```

Evaluation:

Possible evaluation:

"The code is inefficient due to unnecessary calculations, which slow down the process. The code is harder to read due to unnecessary operations as well as inadequate or inappropriate variable names.Additionally, it contains a logical error in handling negative differences. Furthermore, it contains a logical error in handling negative differences. *Also no comments were used (optional).*"

Evaluation Scheme: partial credit scoring

Answer	Points
Incorrect or none of the mentioned issues	0
Some but not all of the mentioned issues	1
All of the mentioned issues	2

Item Fr5x3

Type: Free Format: short essay
Ability to be tested: critical thinking

Task

How could the code be improved regarding its readability and efficiency?

The code is the same as the code for task 5.1.

```
def max(numbers):
    m = 0
    total_elements = len(numbers)

    for i in range(total_elements - 1):
        d = numbers[i + 1] - numbers[i]
        for j in range(1):
            temp_d = d * 1
        if d > m:
            m = d
        elif d < 0:
            m = d
    return m

numbers = [1, 5, 3, 9, 2]
output = max(numbers)
print("Max difference:", output)
```

Evaluation:

Possible evaluation:

"It would benefit from a clearer and more concise structure, eliminating unnecessary steps. Variables should be given descriptive and meaningful names. *The logical error of incorrectly setting* `max_diff` *to negative difference in line 12 should be corrected and the use of comments could improve the readability (optional).*"

Evaluation Scheme: partial credit scoring

Answer	**Points**
Incorrect or none of the mentioned issues	0
Some but not all of the mentioned issues	1
All of the mentioned issues	2

Appendix C

Debriefing Questions

Please respond to the following statements according to your experience. There are no right or wrong answers - the important thing is to answer honestly and spontaneously.

Question D1

Question					
The task was clear and unambiguous.					
Strongly disagree	Disagree	Agree	Strongly agree	Task started	not
☐	☐	☐	☐	☐	

Disruptive Factor to be identified: Unclear or incomplete instructions

Question D2

Question					
The difficulty level of the task matched my skill and experience.					
Strongly disagree	Disagree	Agree	Strongly agree	Task started	not
☐	☐	☐	☐	☐	

Disruptive Factor to be identified: Mismatch between task difficulty and skill level

Question D3

Question

I felt I had sufficient time to complete the task without feeling rushed.

Strongly disagree	Disagree	Agree	Strongly agree	Task not started
☐	☐	☐	☐	☐

Disruptive Factor to be identified: Time

Question D4

Question

Did you experience any technical difficulties while completing the task?

Yes, I did.
No, I did not.

What was the problem?

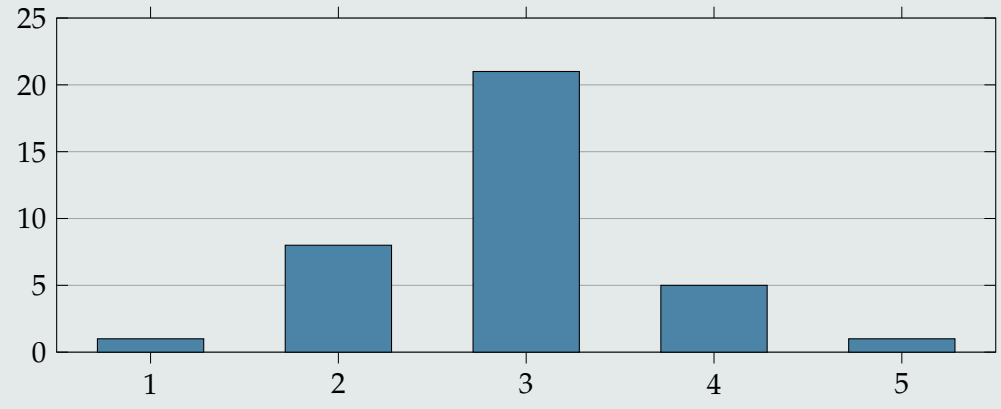
Disruptive Factor to be identified: Technical difficulties (hardware/software/network issues)

Appendix D

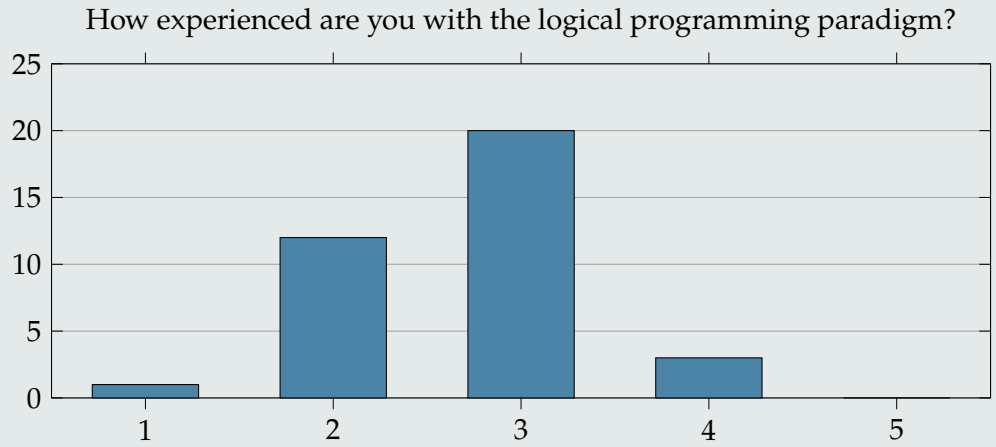
Figures

D.1 Personality-Related Questions

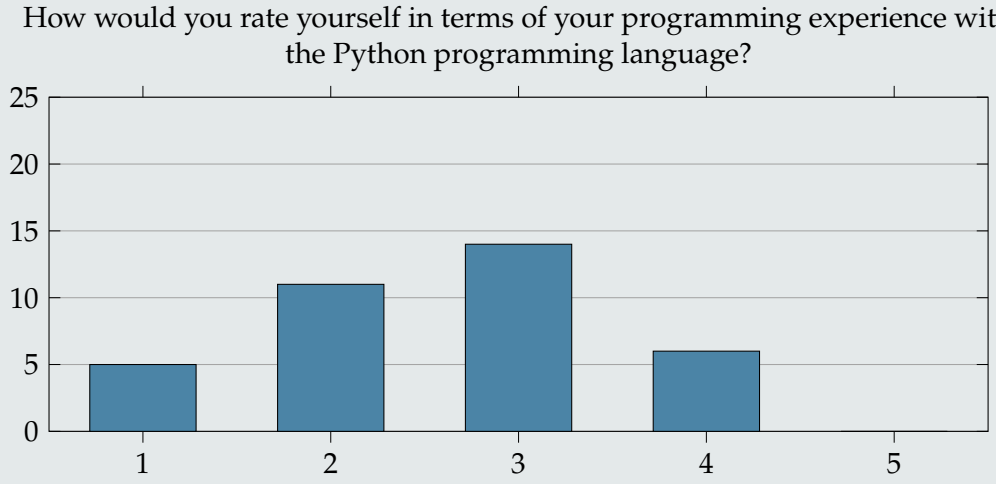
Question P1.1



Question P1.2

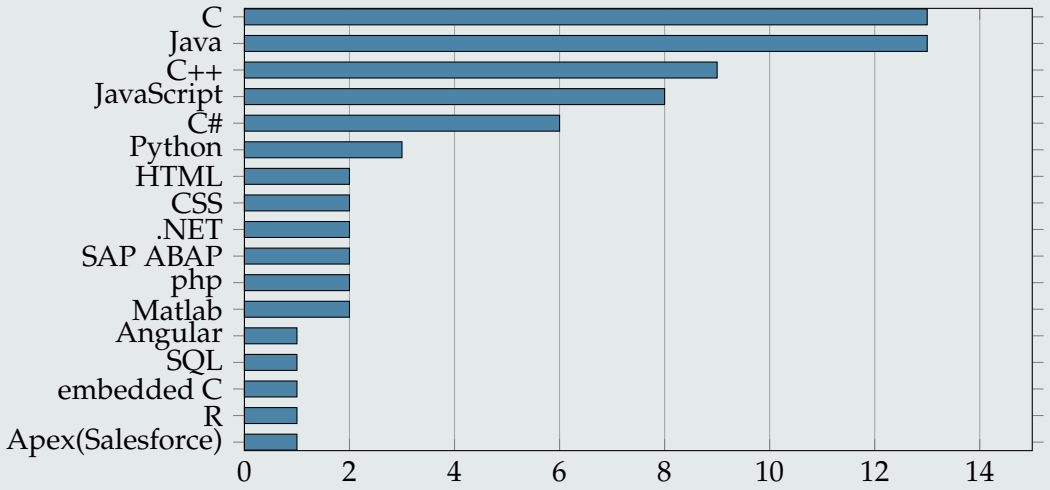


Question P1.3



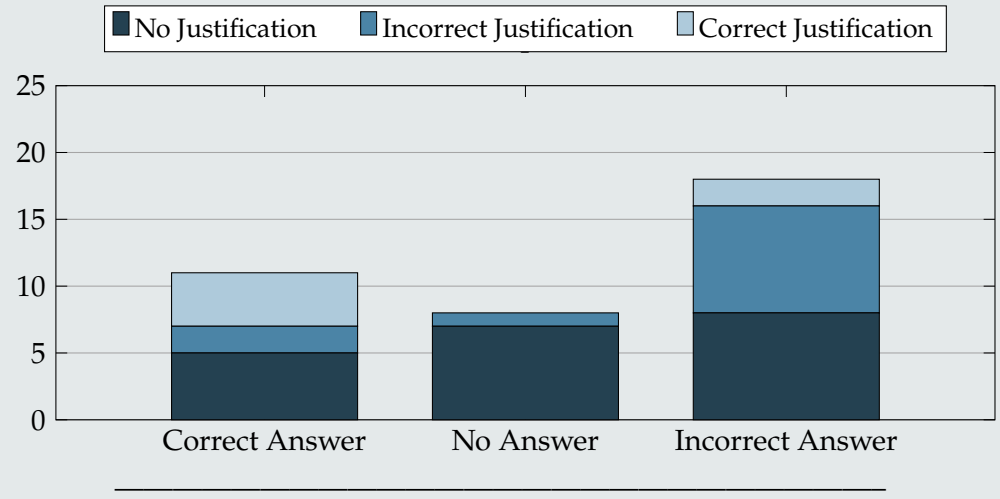
Question P2

Which other programming languages do you consider yourself to have at least an intermediate level of proficiency in? If it is more than 3, please specify only your 3 most familiar languages and the number of the remaining languages.

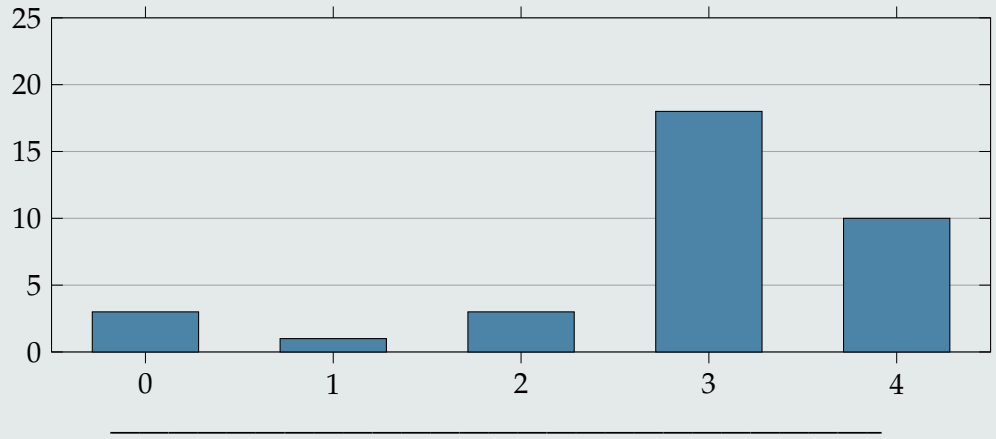


D.2 Tasks with Debriefing

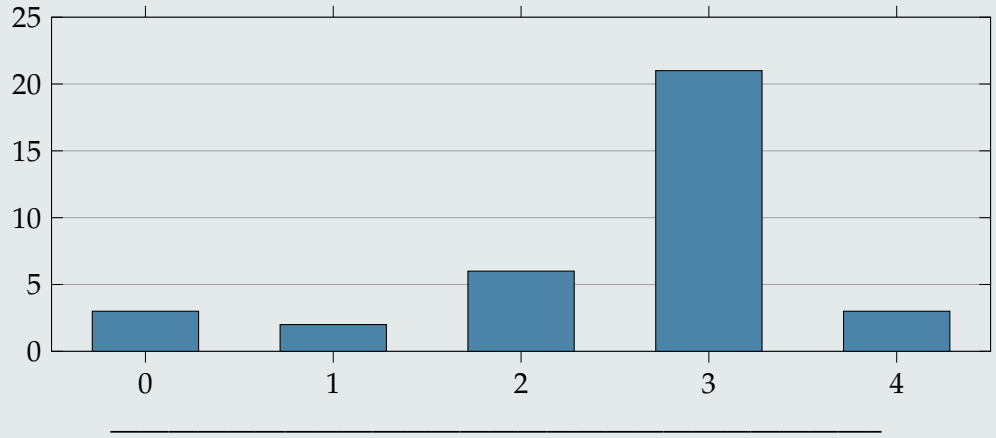
Fr1x1



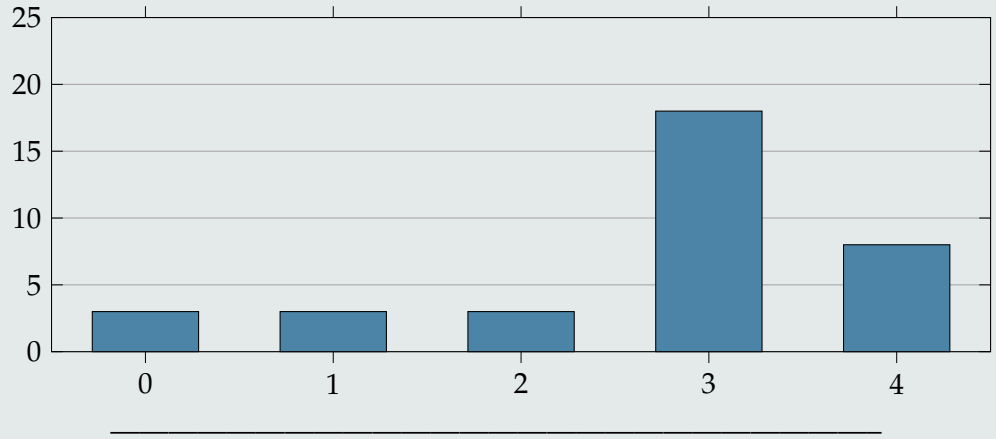
DeFr1x1.1
The task was clear and unambiguous.

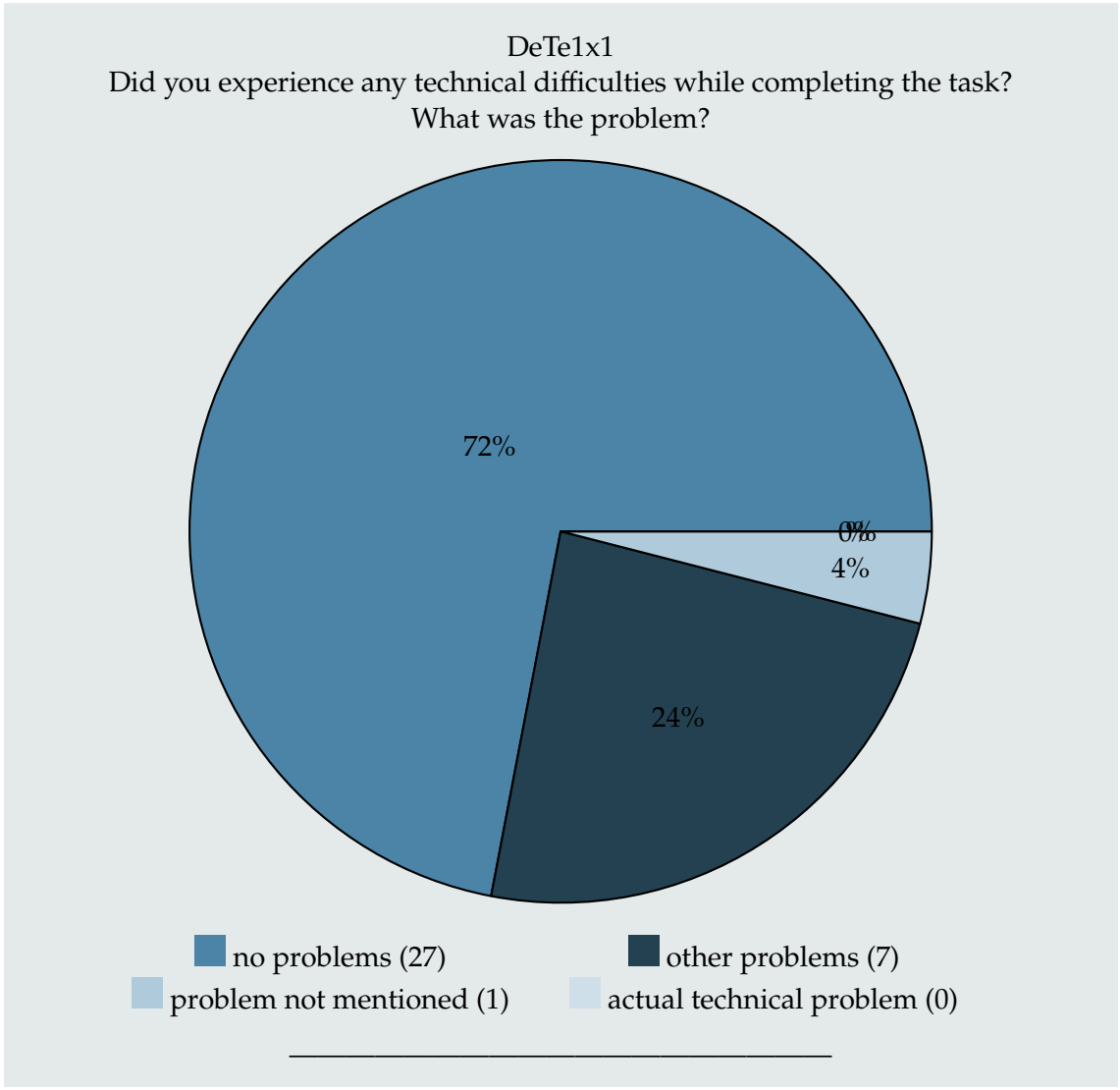


DeFr1x1.2
The difficulty level of the task matched my skill and experience.

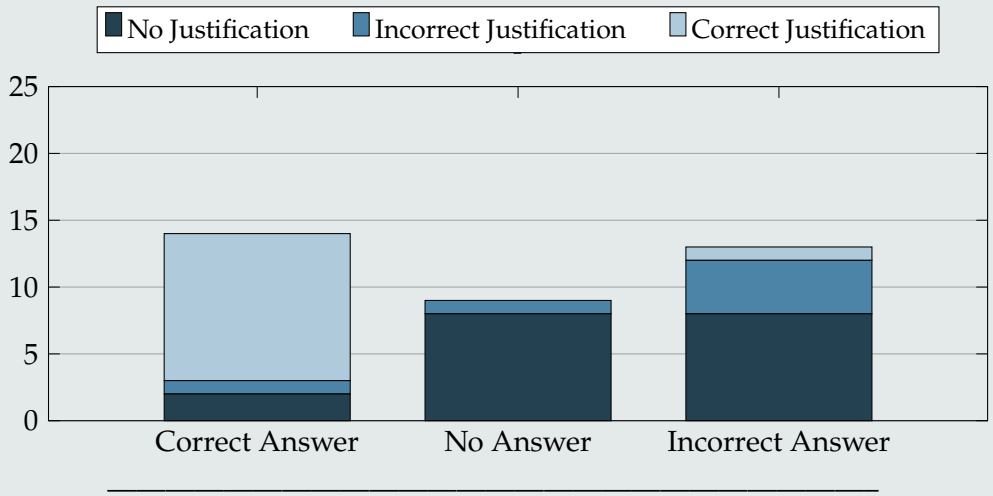


DeFr1x1.3
I felt I had sufficient time to complete the task without feeling rushed.

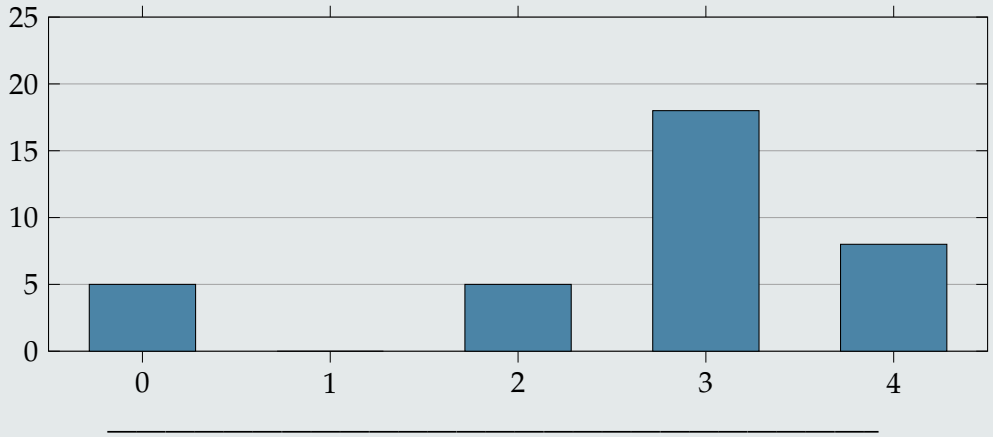



DeTe1x1
Did you experience any technical difficulties while completing the task?
What was the problem?
72%
0%
4%
24%
no problems (27)
other problems (7)
problem not mentioned (1)
actual technical problem (0)

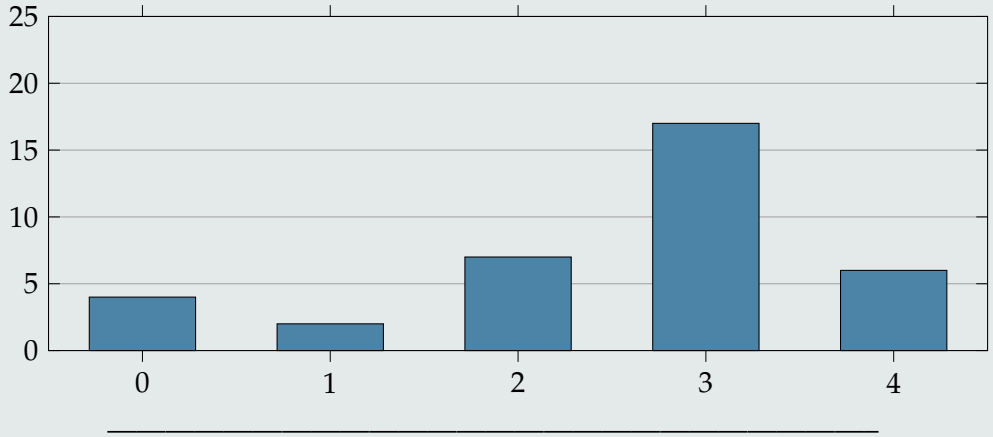
Fr1x2



DeFr1x2.1
The task was clear and unambiguous.

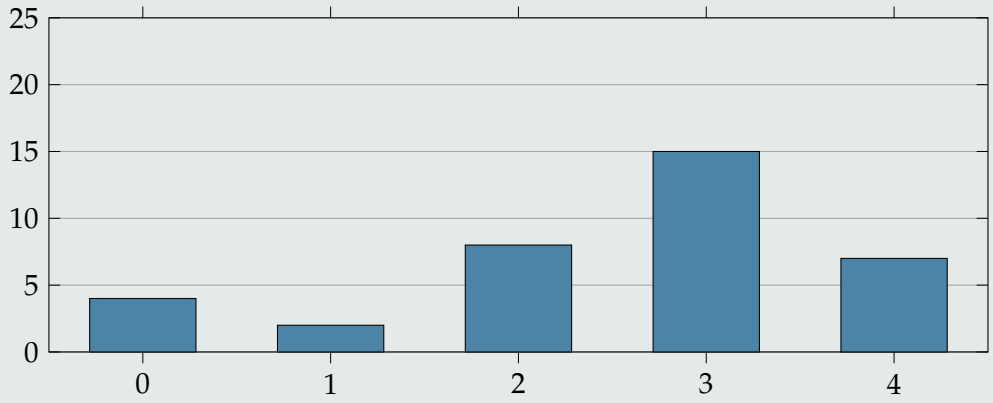


DeFr1x2.2
The difficulty level of the task matched my skill and experience.



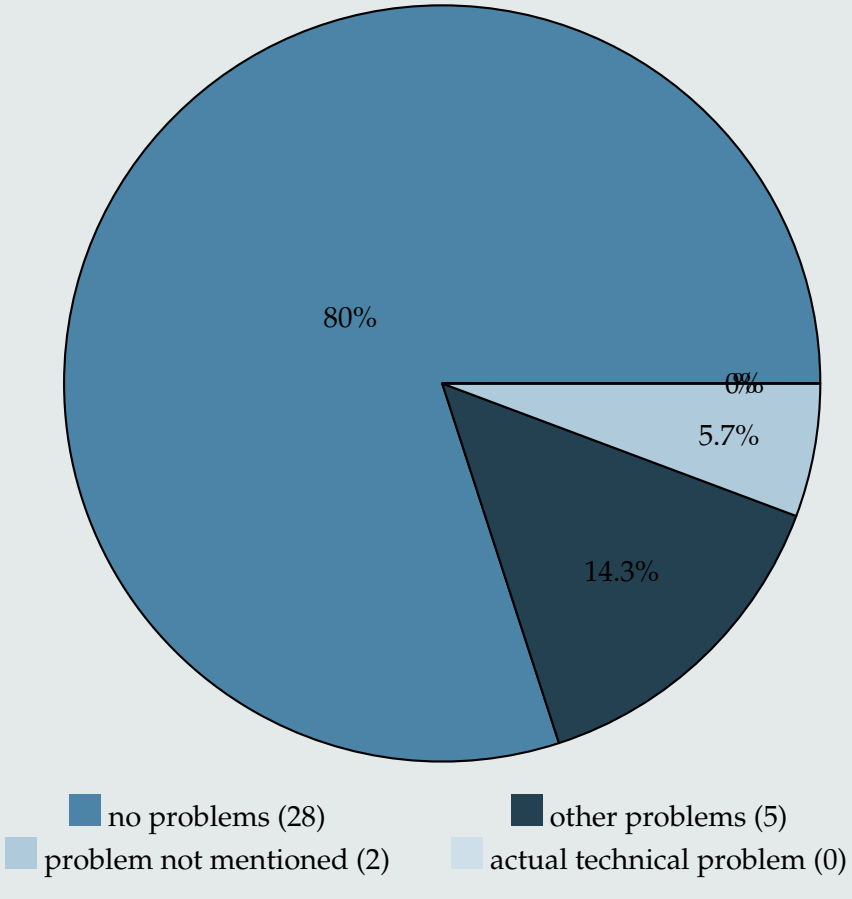
DeFr1x2.3

I felt I had sufficient time to complete the task without feeling rushed.

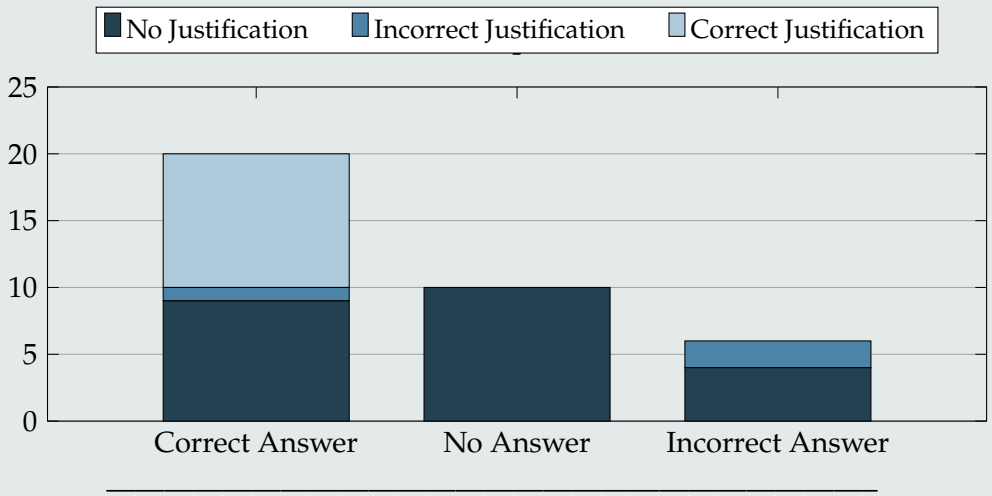


DeTe1x2

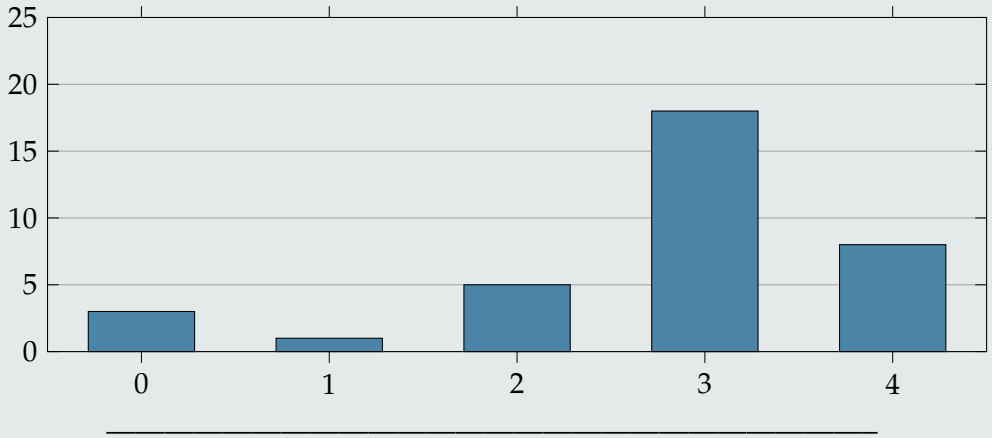
Did you experience any technical difficulties while completing the task?
What was the problem?



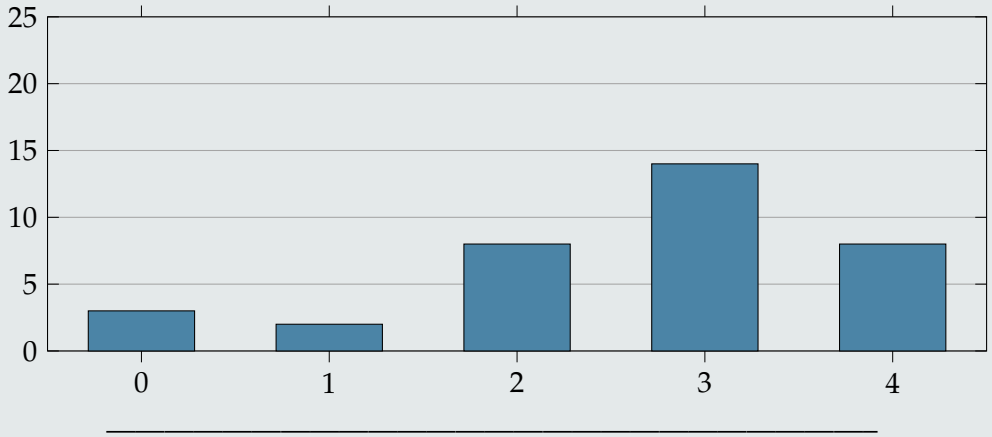
Fr1x3



DeFr1x3.1
The task was clear and unambiguous.

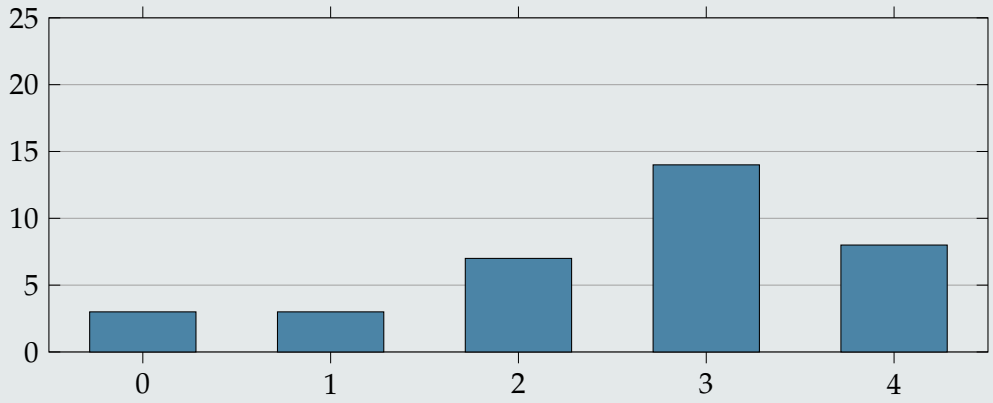


DeFr1x3.2
The difficulty level of the task matched my skill and experience.



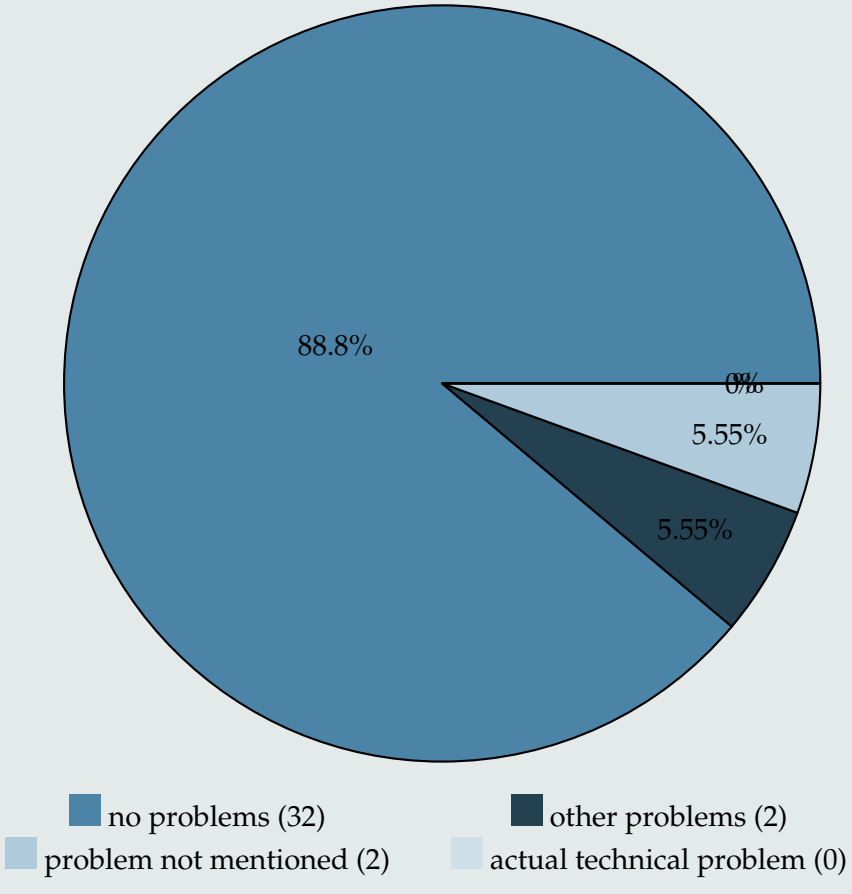
DeFr1x3.3

I felt I had sufficient time to complete the task without feeling rushed.

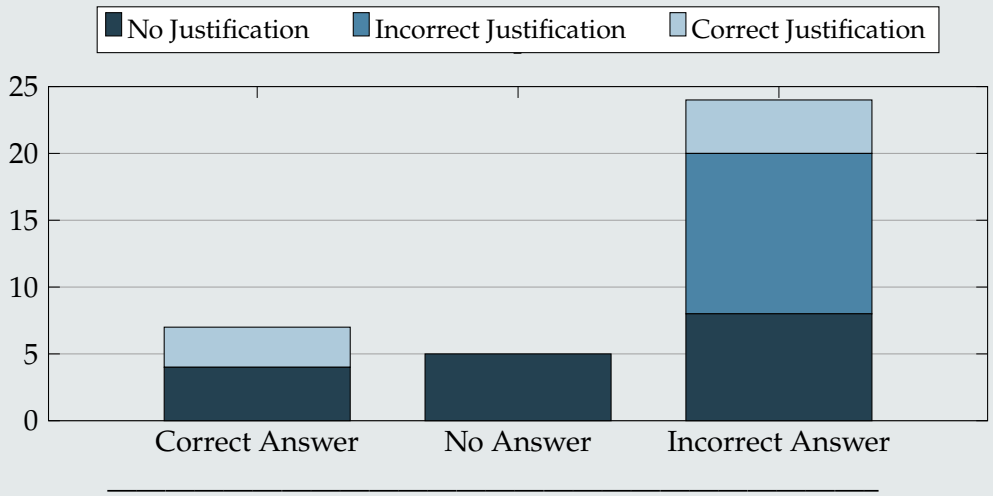


DeTe1x3

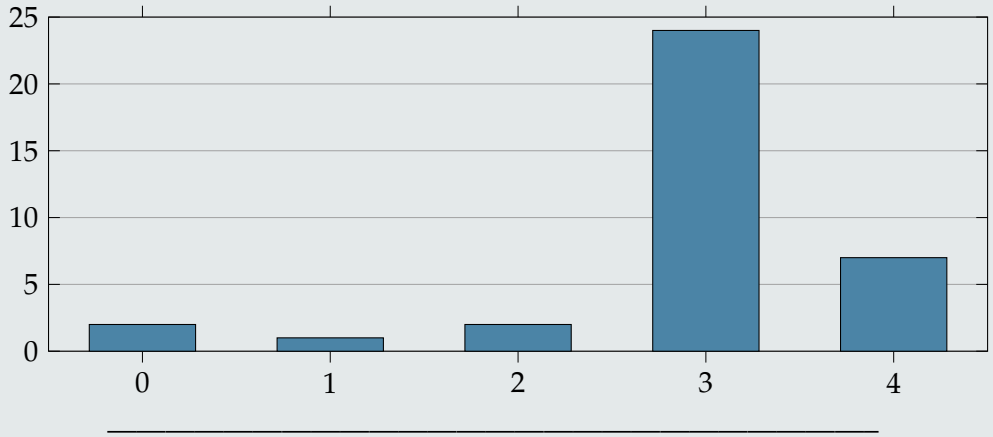
Did you experience any technical difficulties while completing the task?
What was the problem?



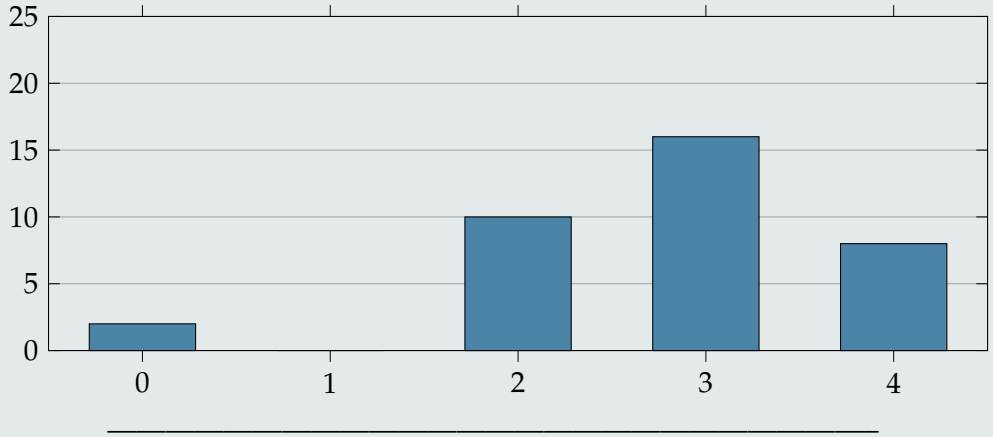
Fr2x1



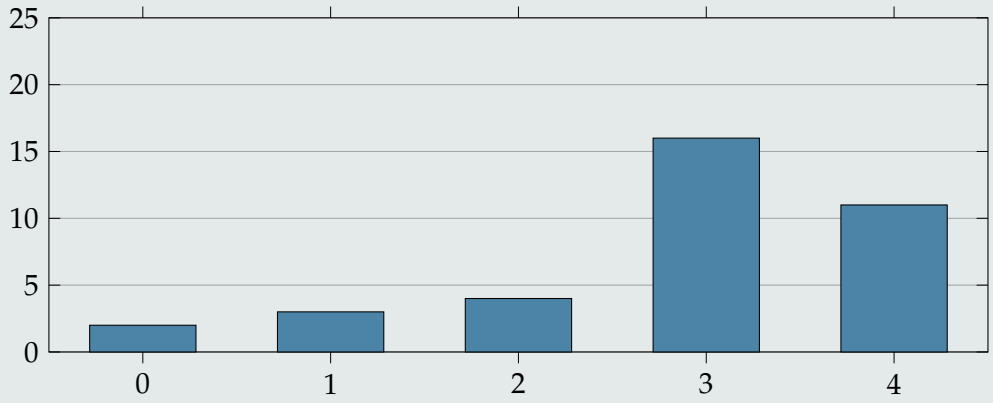
DeFr2x1.1
The task was clear and unambiguous.



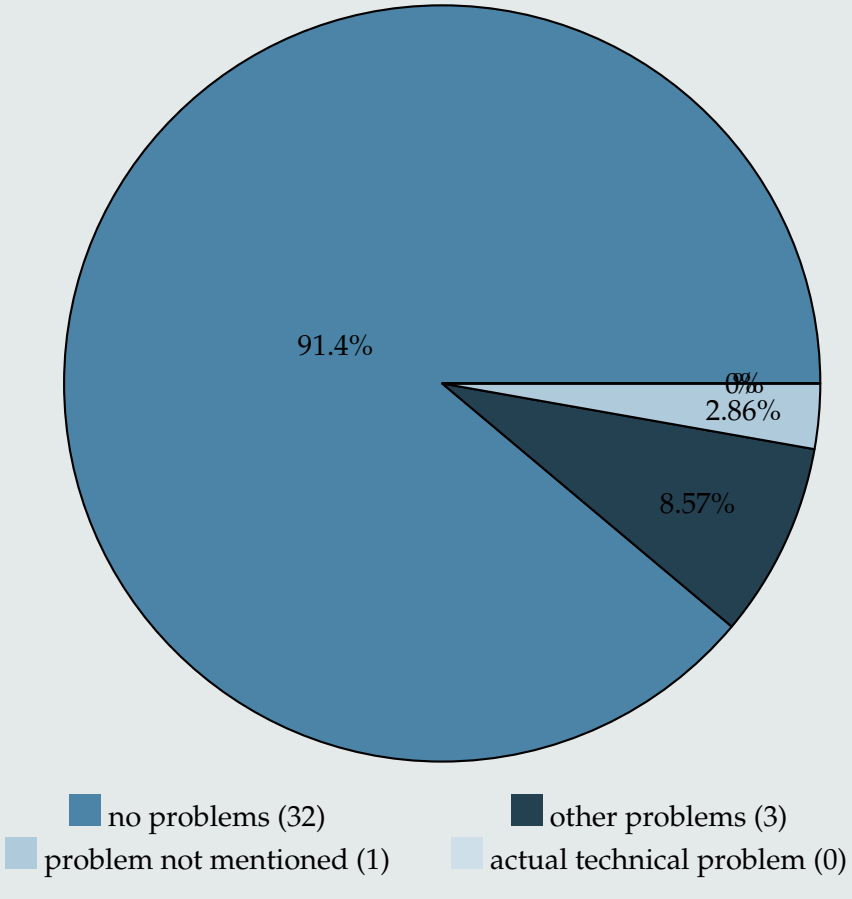
DeFr2x1.2
The difficulty level of the task matched my skill and experience.



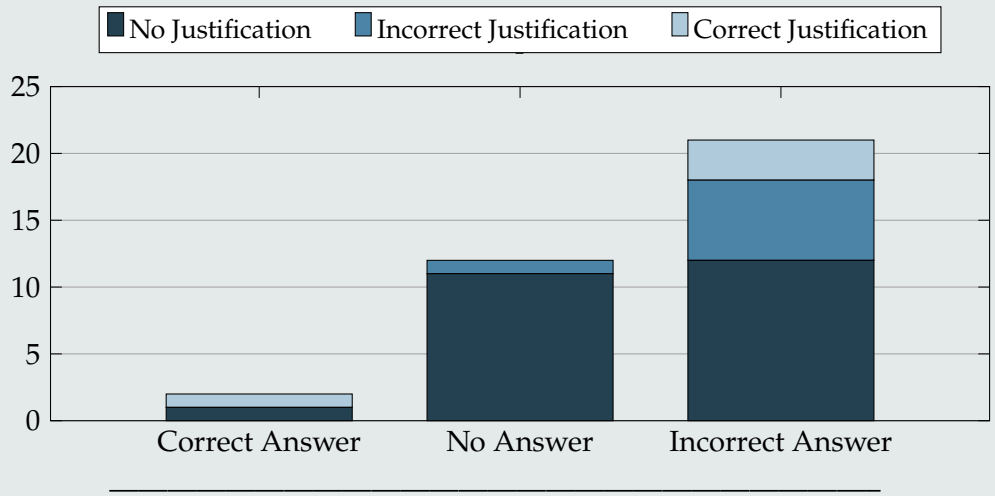
DeFr2x1.3
I felt I had sufficient time to complete the task without feeling rushed.



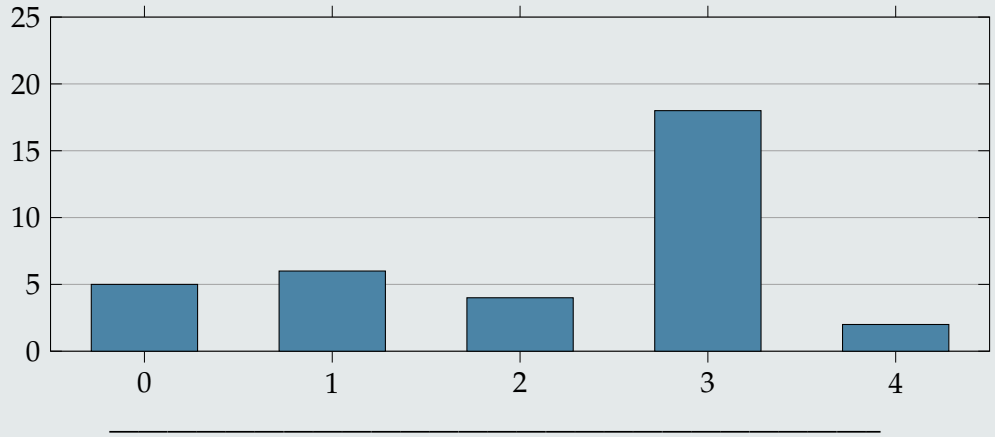
DeTe2x1
Did you experience any technical difficulties while completing the task?
What was the problem?



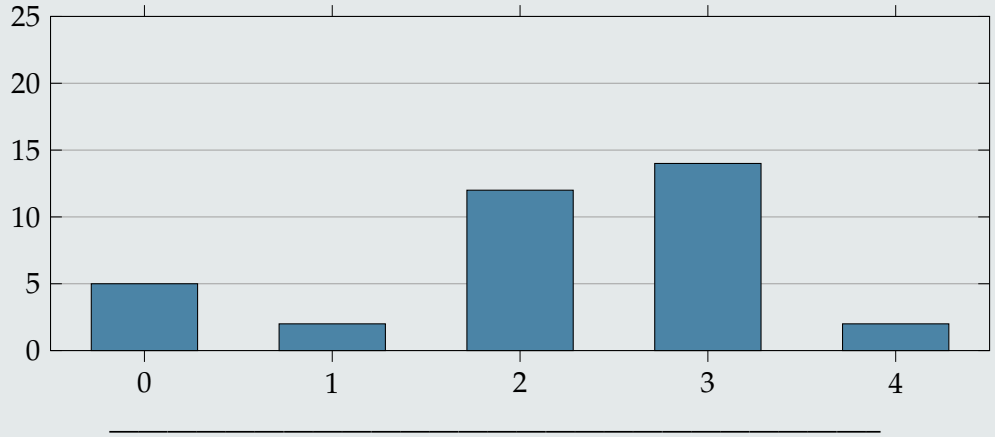
Fr2x2



DeFr2x2.1
The task was clear and unambiguous.

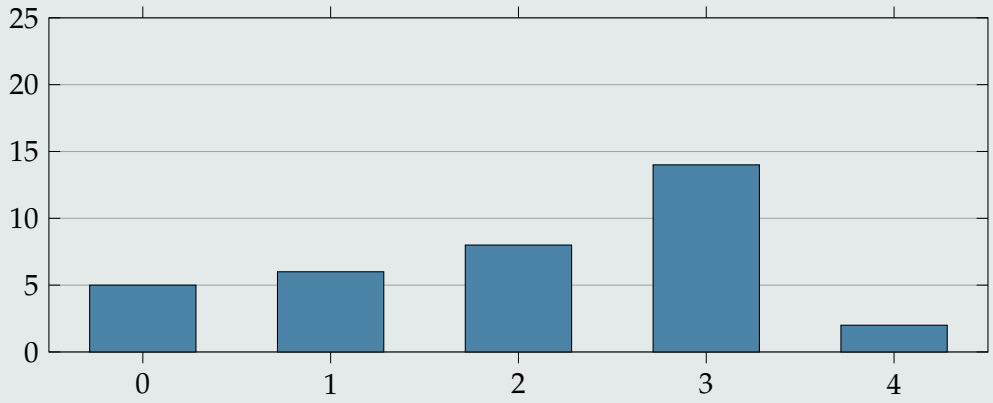


DeFr2x2.2
The difficulty level of the task matched my skill and experience.



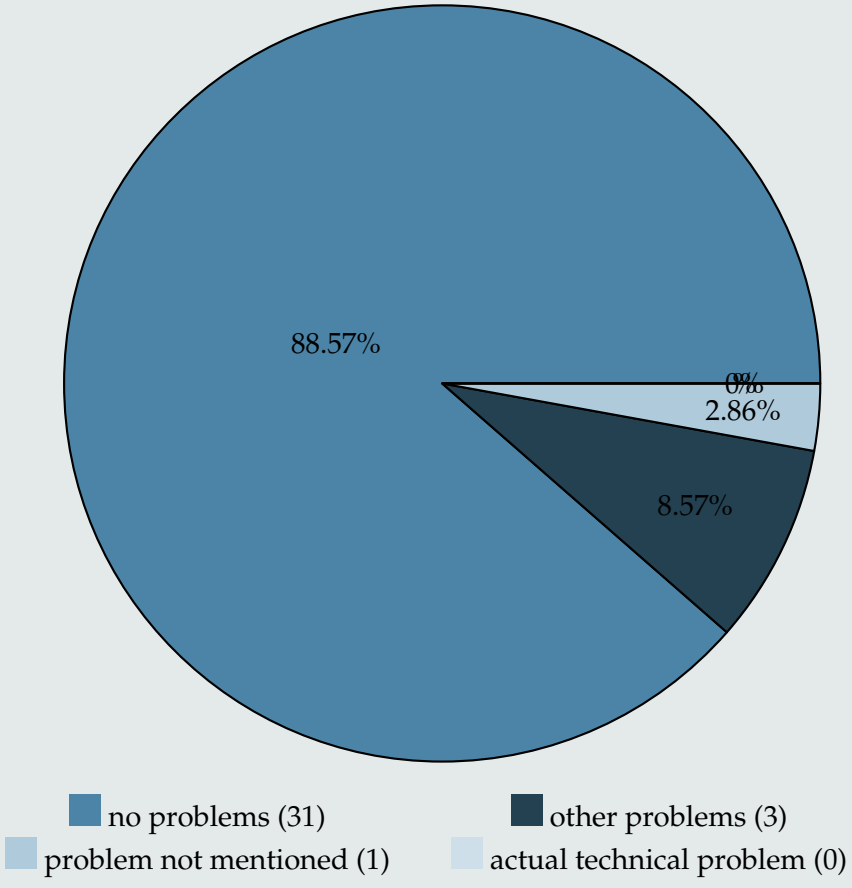
DeFr2x2.3

I felt I had sufficient time to complete the task without feeling rushed.

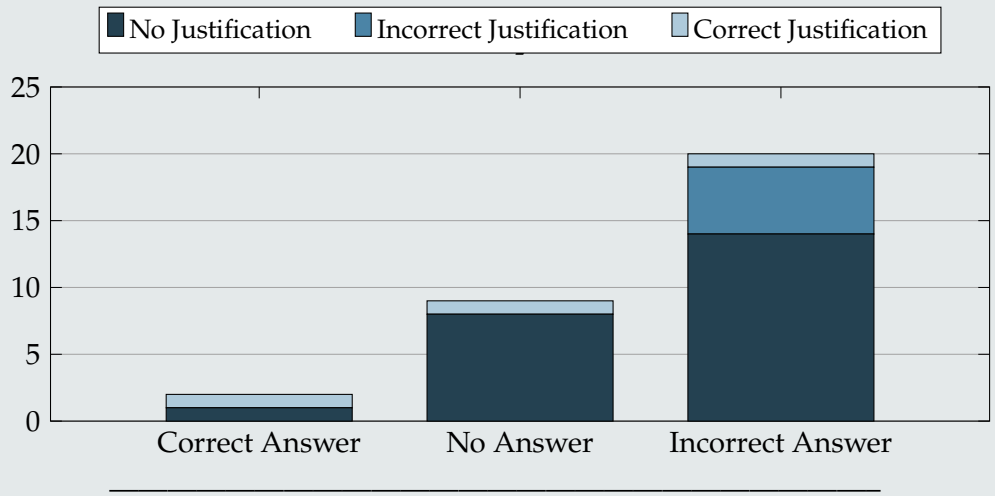


DeTe1x3

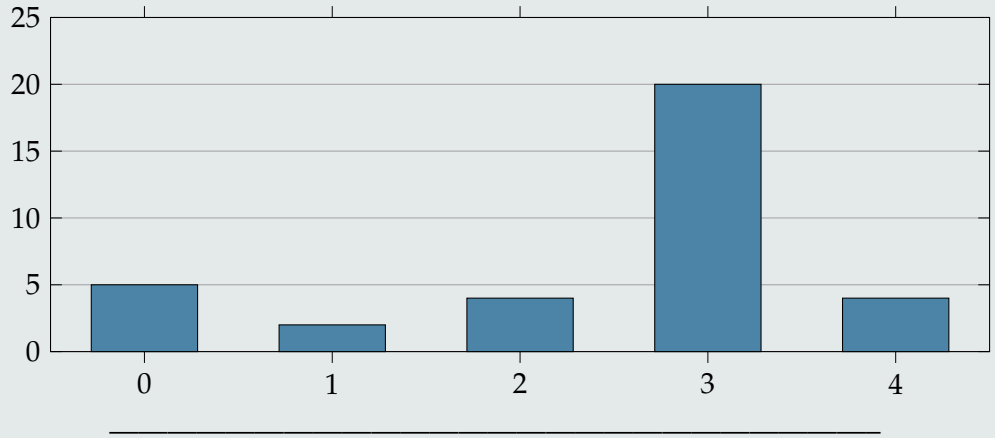
Did you experience any technical difficulties while completing the task?
What was the problem?



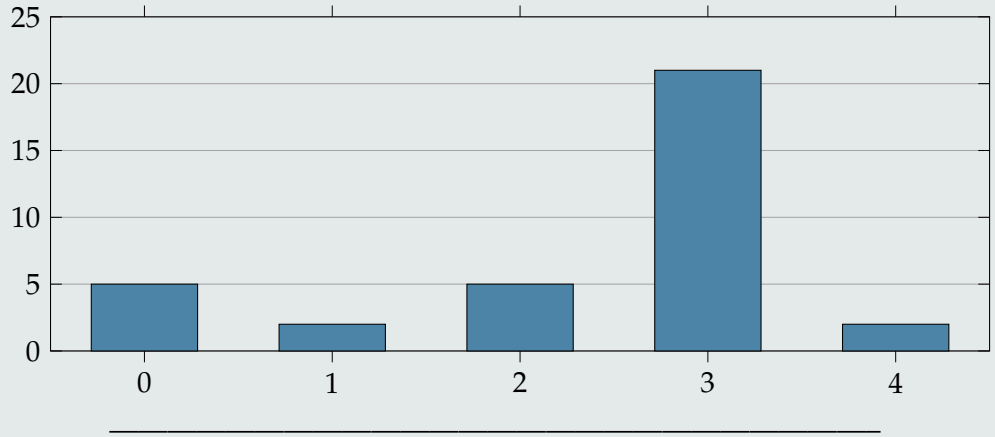
Fr2x3



DeFr2x3.1
The task was clear and unambiguous.

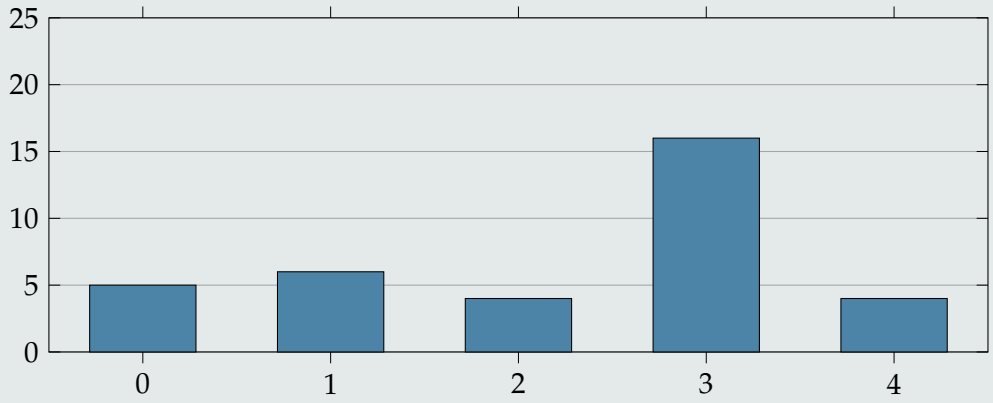


DeFr2x3.2
The difficulty level of the task matched my skill and experience.



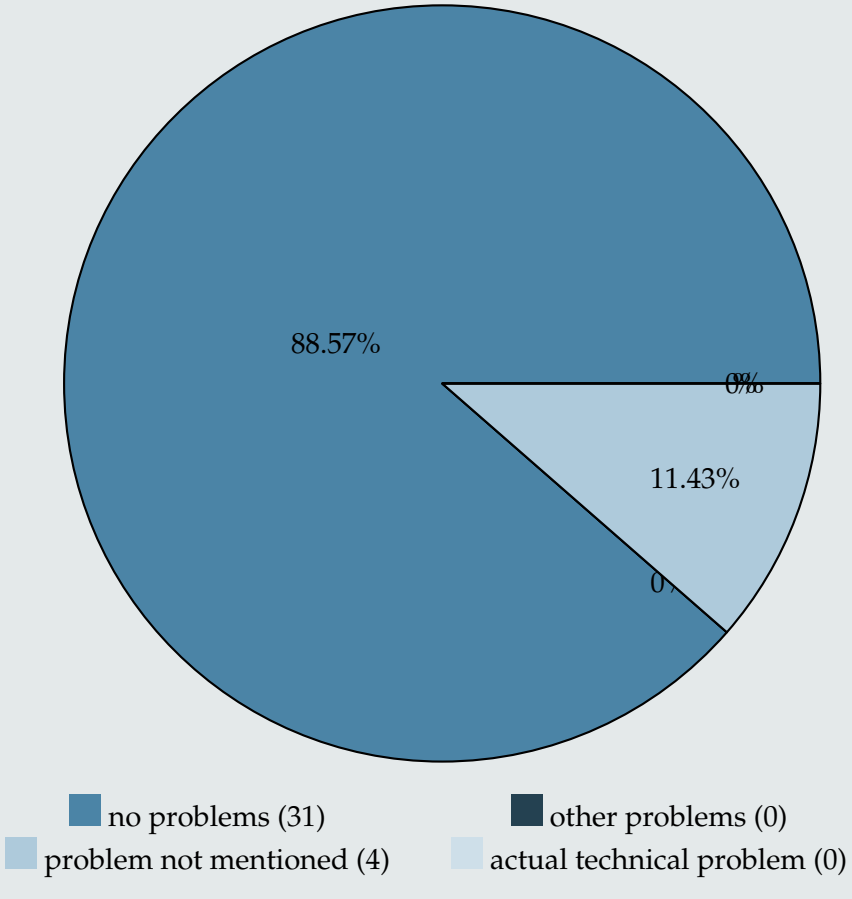
DeFr2x3.3

I felt I had sufficient time to complete the task without feeling rushed.

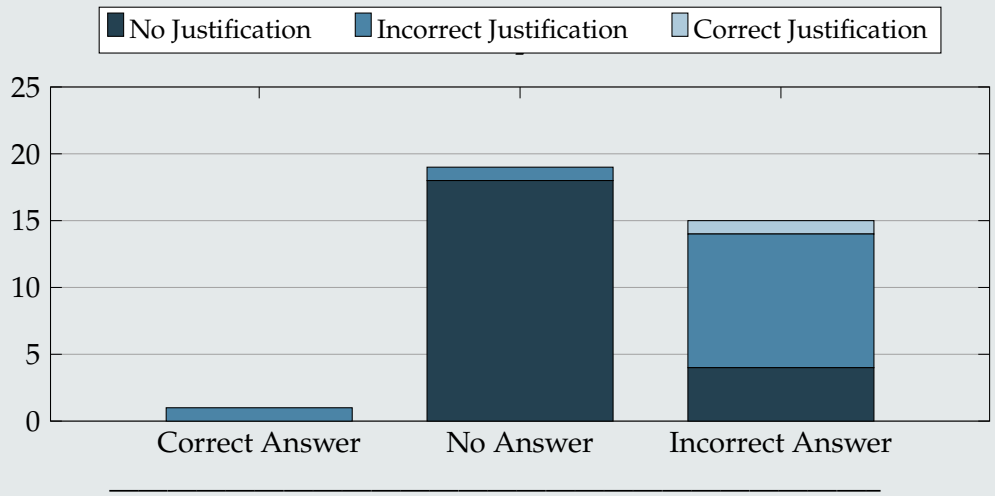


DeTe1x3

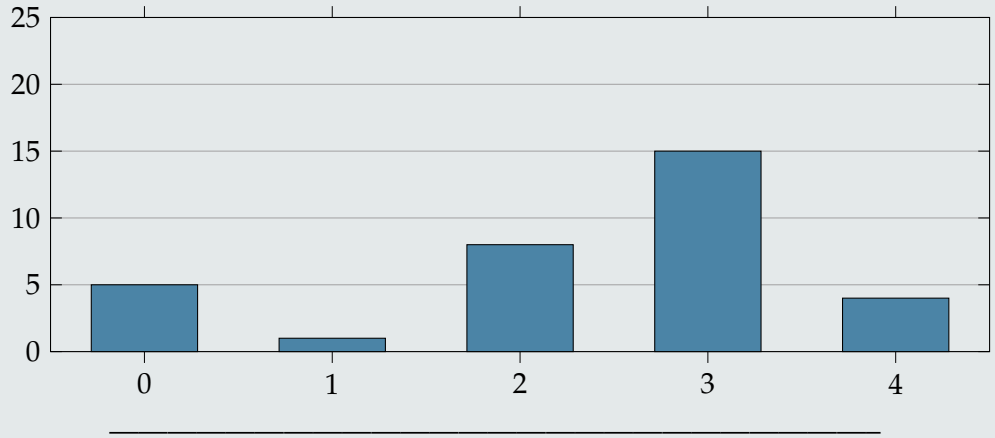
Did you experience any technical difficulties while completing the task?
What was the problem?



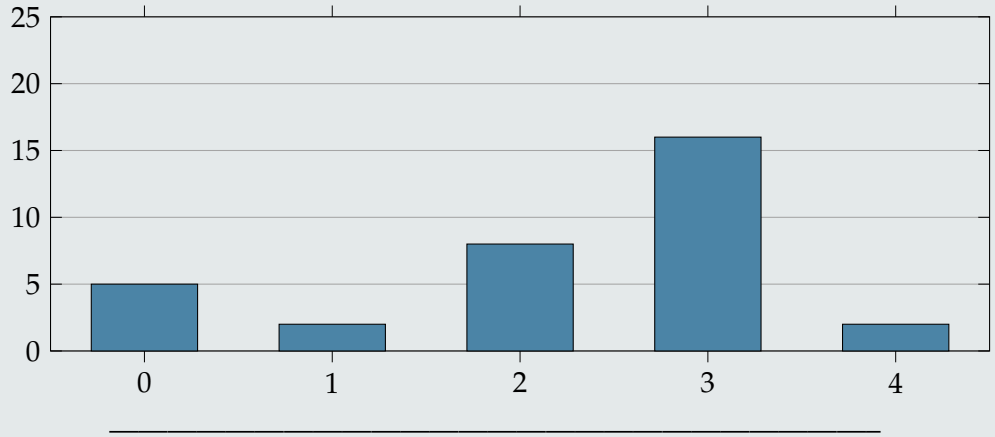
Fr3x1



DeFr3x1.1
The task was clear and unambiguous.

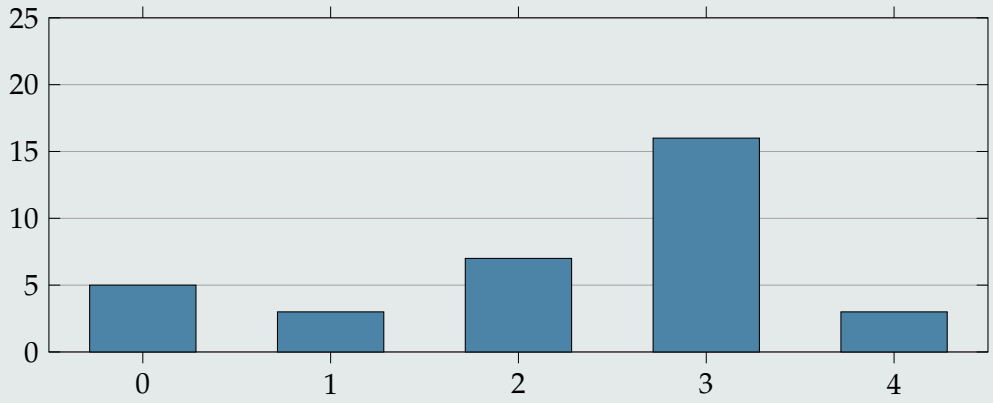


DeFr3x1.2
The difficulty level of the task matched my skill and experience.



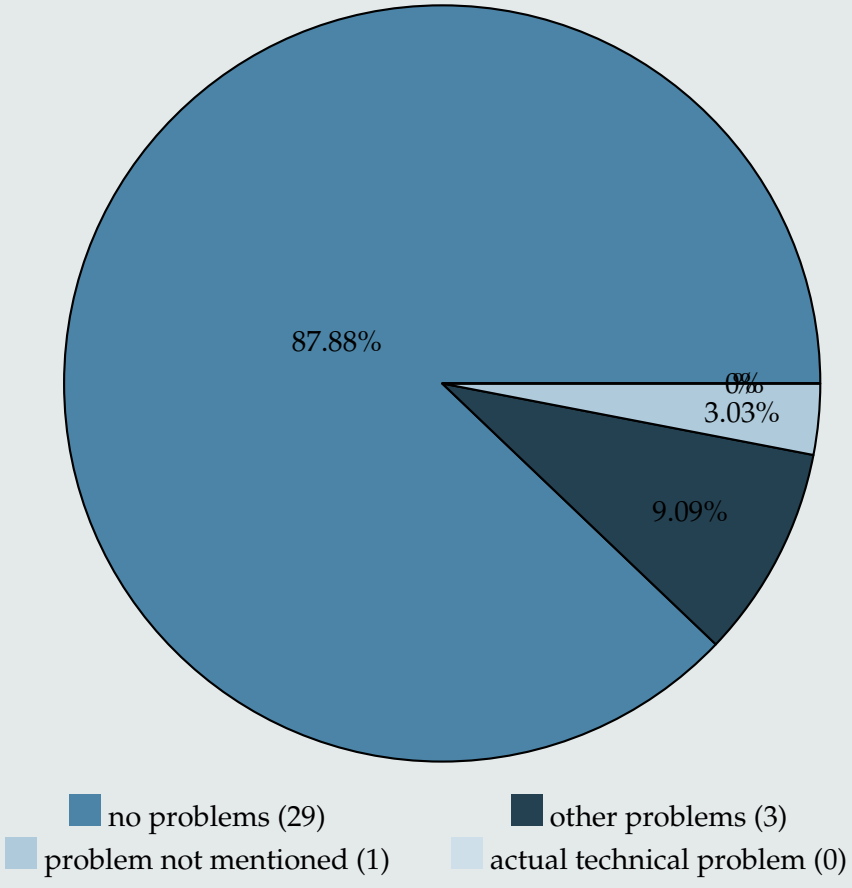
DeFr3x1.3

I felt I had sufficient time to complete the task without feeling rushed.

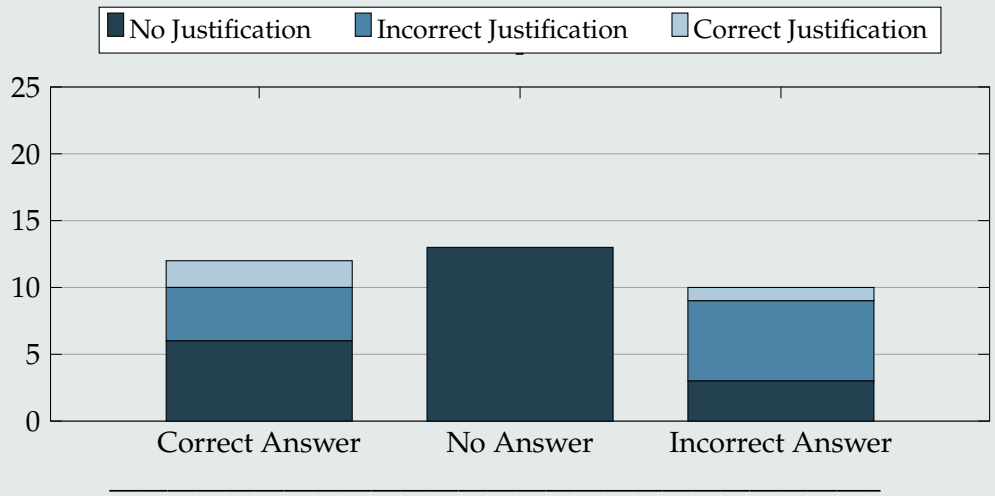


DeTe2x1

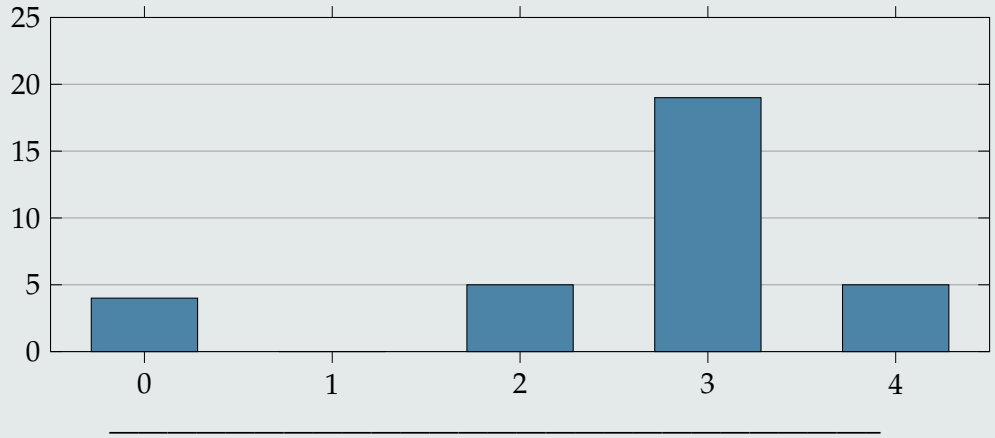
Did you experience any technical difficulties while completing the task?
What was the problem?



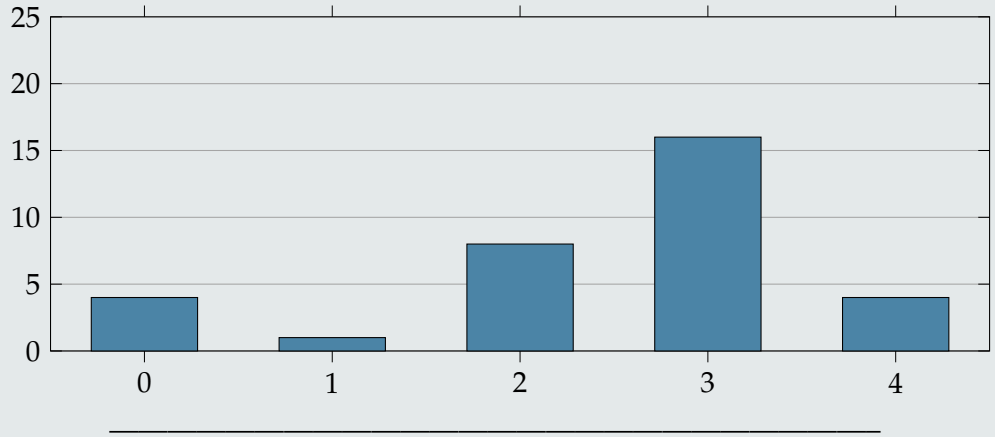
Fr3x2



DeFr3x2.1
The task was clear and unambiguous.

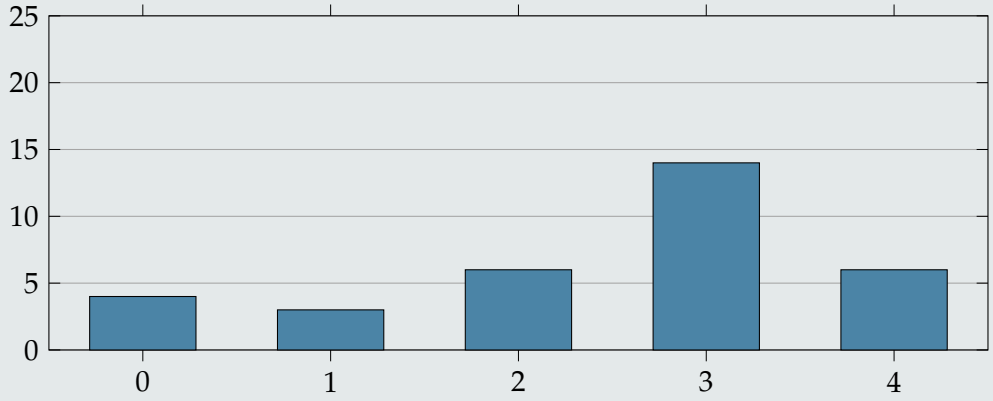


DeFr3x2.2
The difficulty level of the task matched my skill and experience.



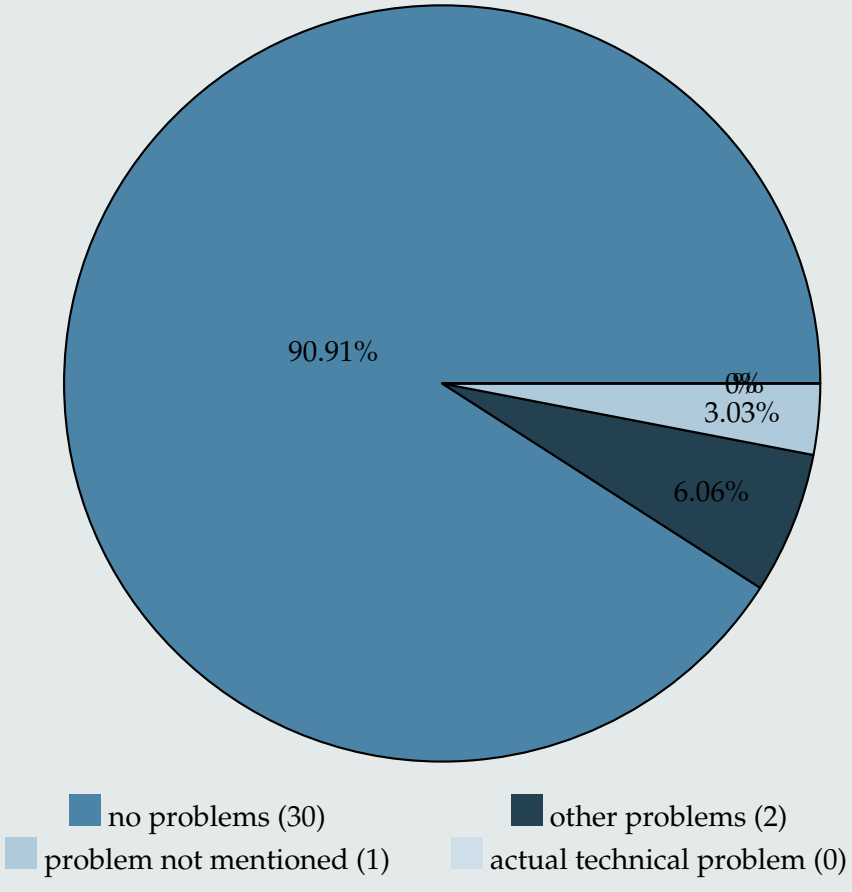
DeFr3x2.3

I felt I had sufficient time to complete the task without feeling rushed.

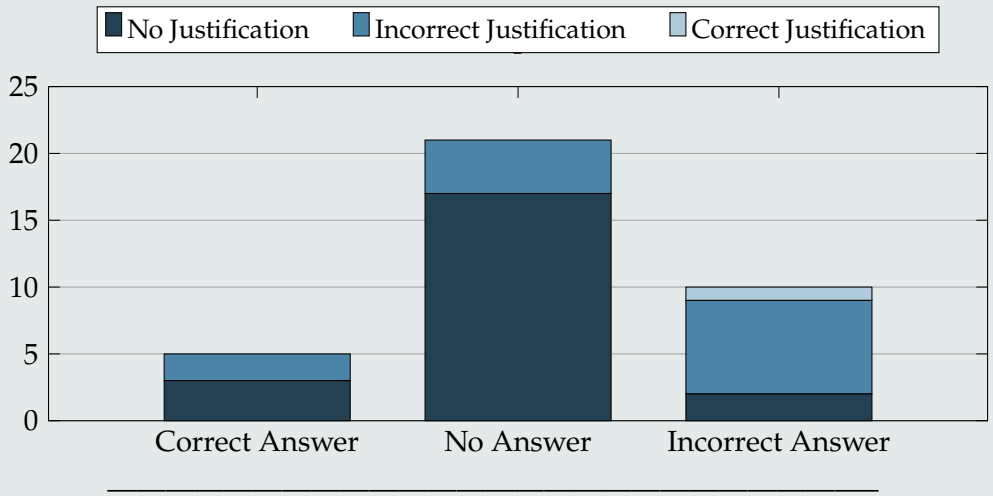


DeTe1x3

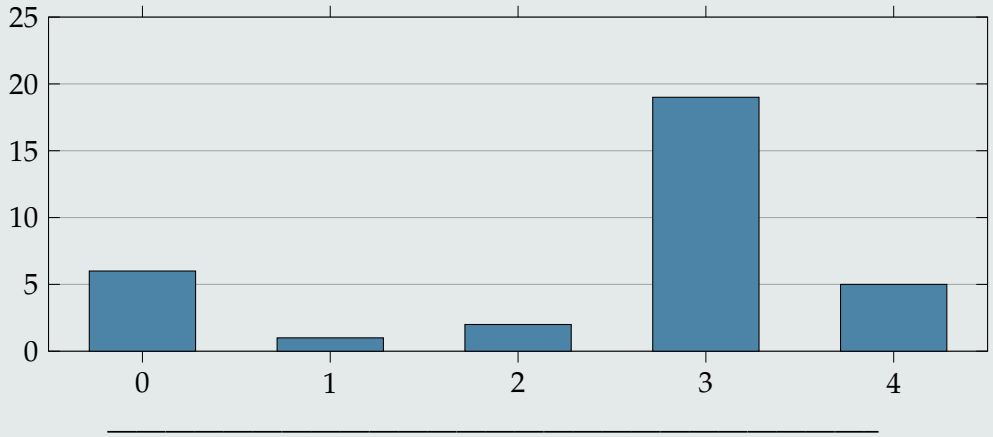
Did you experience any technical difficulties while completing the task?
What was the problem?



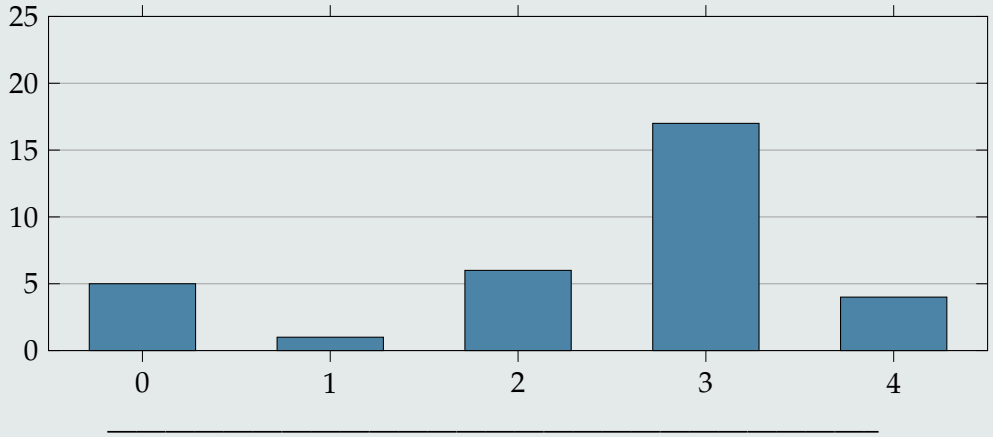
Fr4x1



DeFr4x1.1
The task was clear and unambiguous.

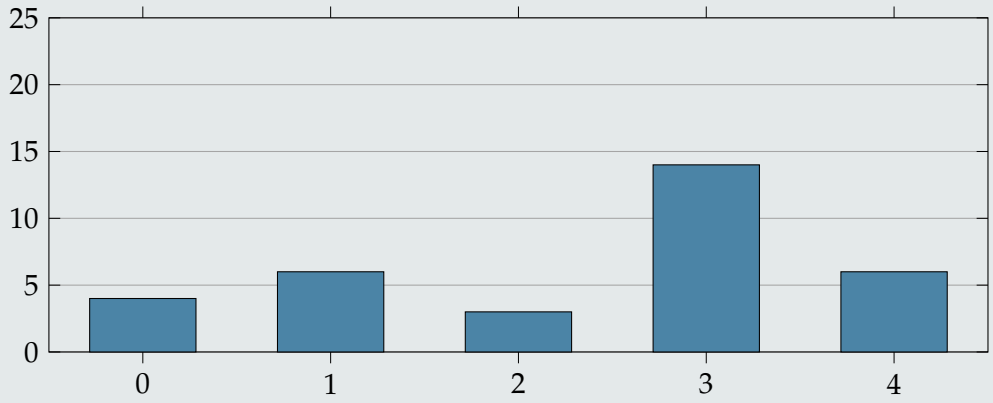


DeFr4x1.2
The difficulty level of the task matched my skill and experience.



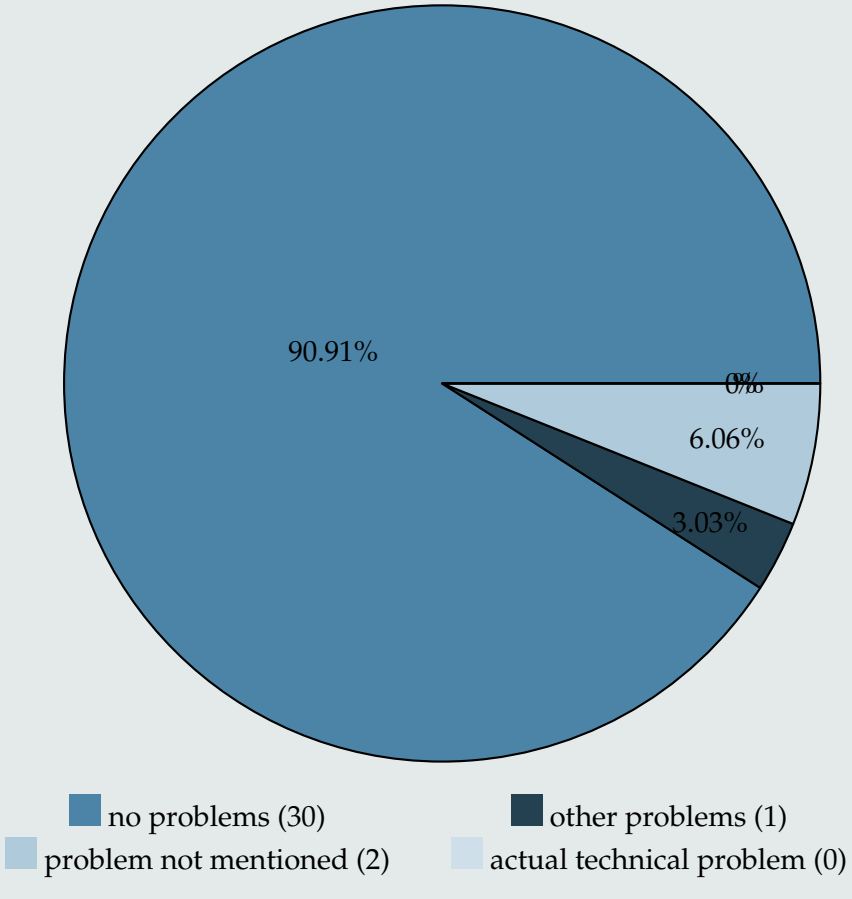
DeFr4x1.3

I felt I had sufficient time to complete the task without feeling rushed.

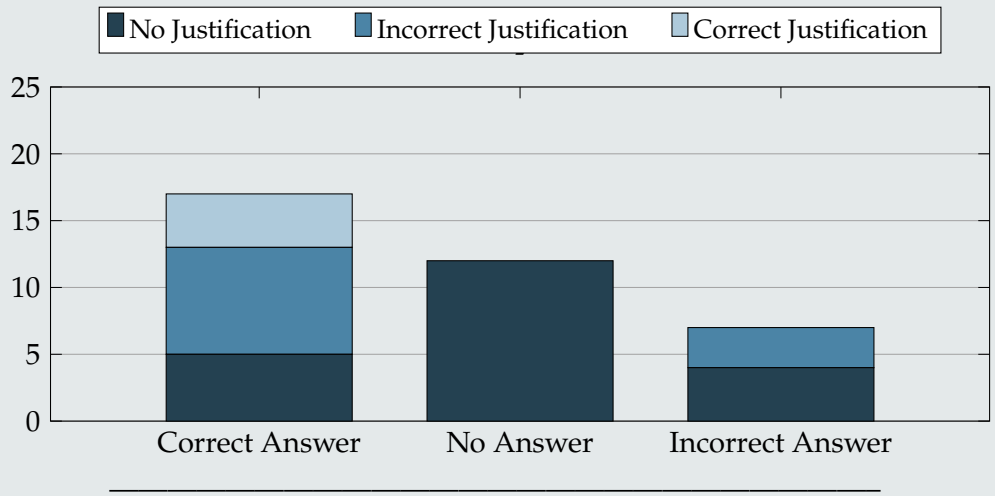


DeTe2x1

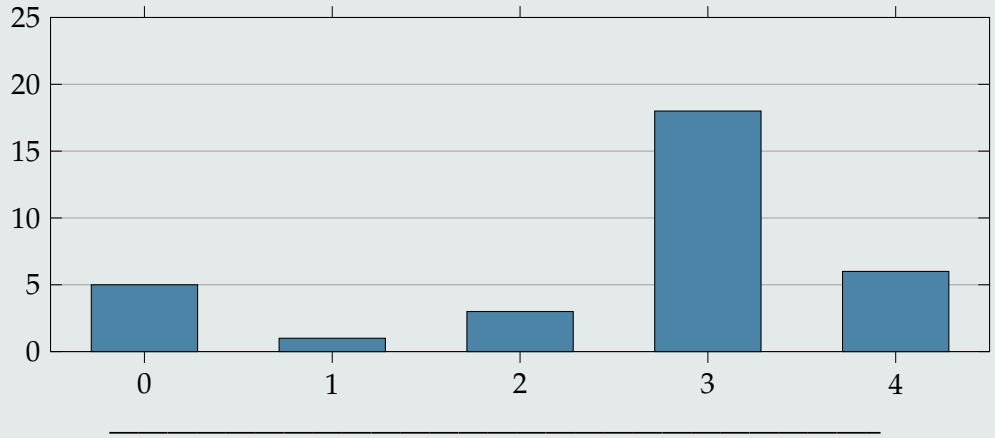
Did you experience any technical difficulties while completing the task?
What was the problem?



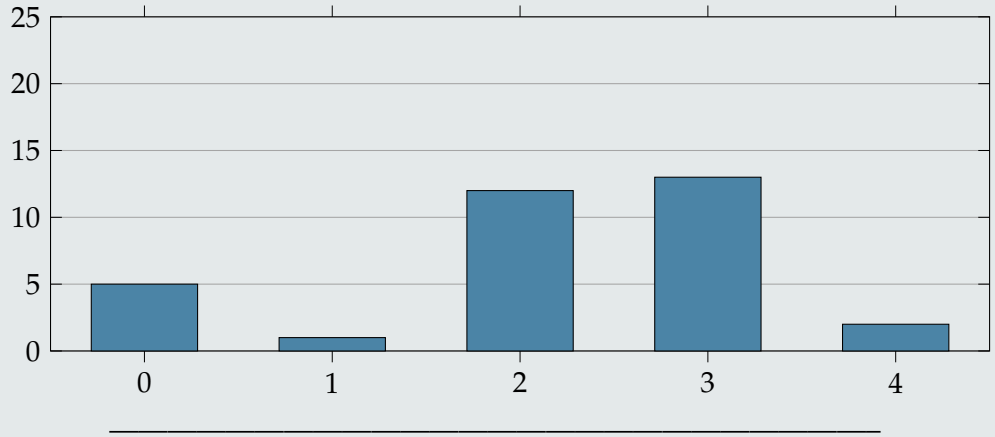
Fr5x1



DeFr5x1.1
The task was clear and unambiguous.

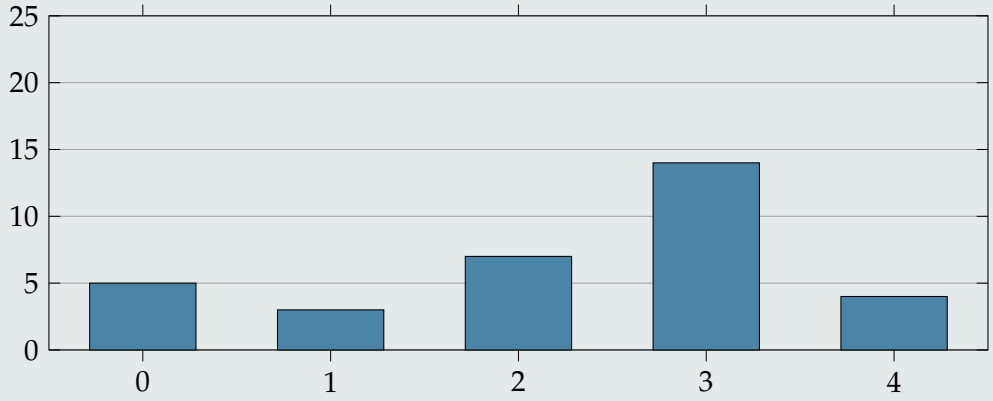


DeFr5x1.2
The difficulty level of the task matched my skill and experience.



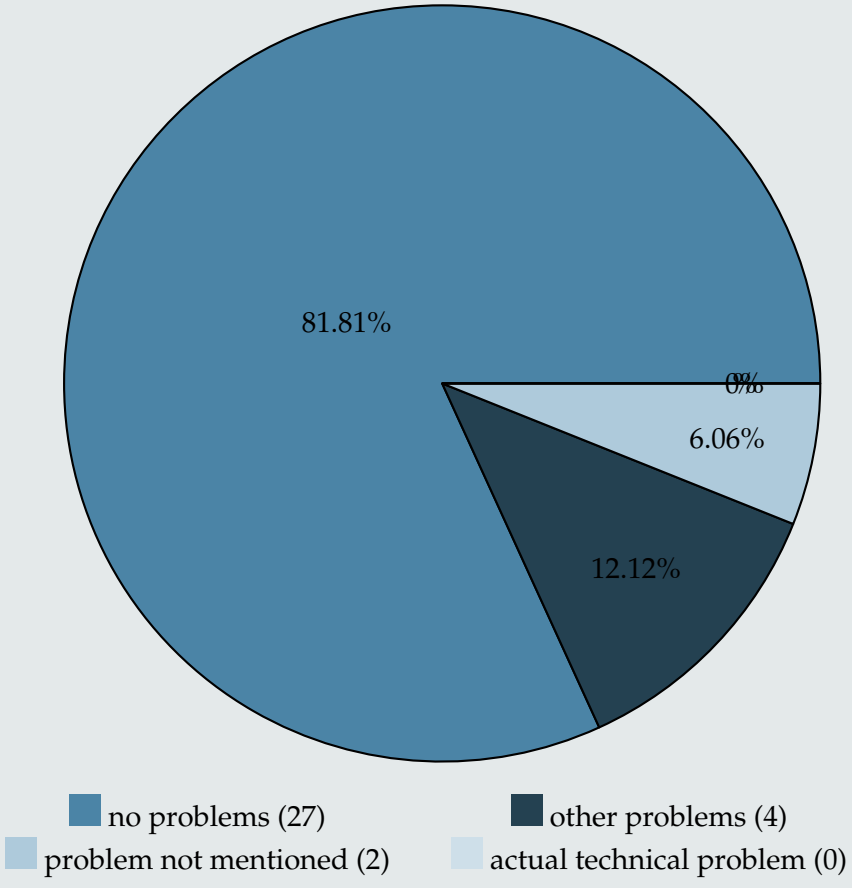
DeFr5x1.3

I felt I had sufficient time to complete the task without feeling rushed.

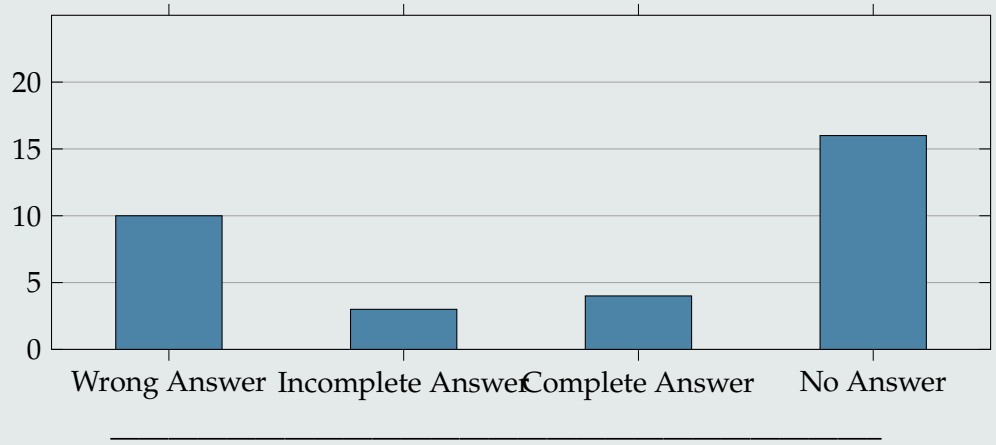


DeTe2x1

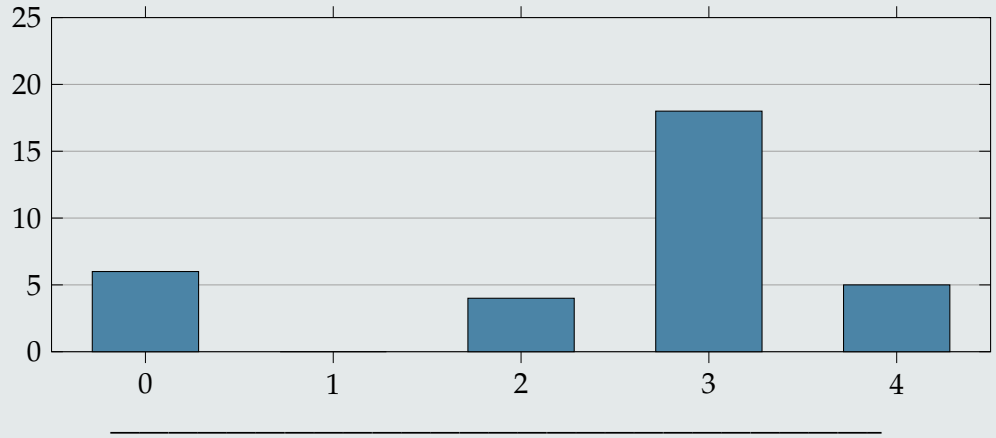
Did you experience any technical difficulties while completing the task?
What was the problem?



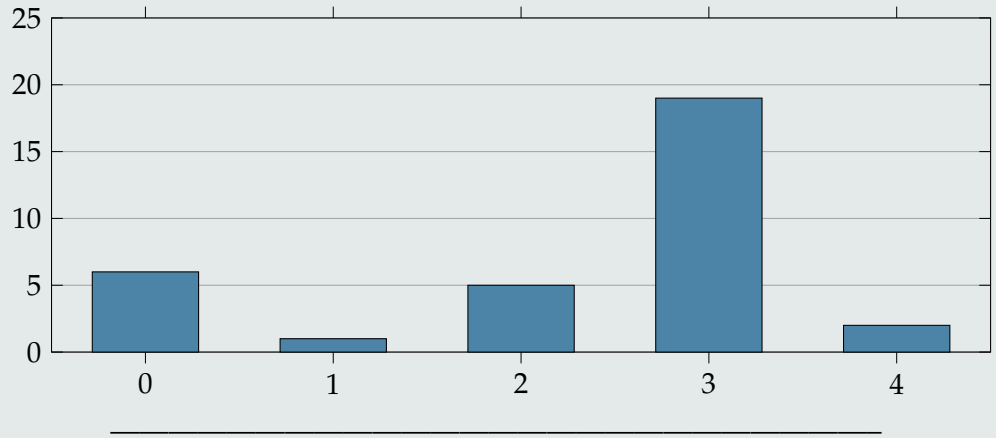
Fr5x2



DeFr5x2.1
The task was clear and unambiguous.

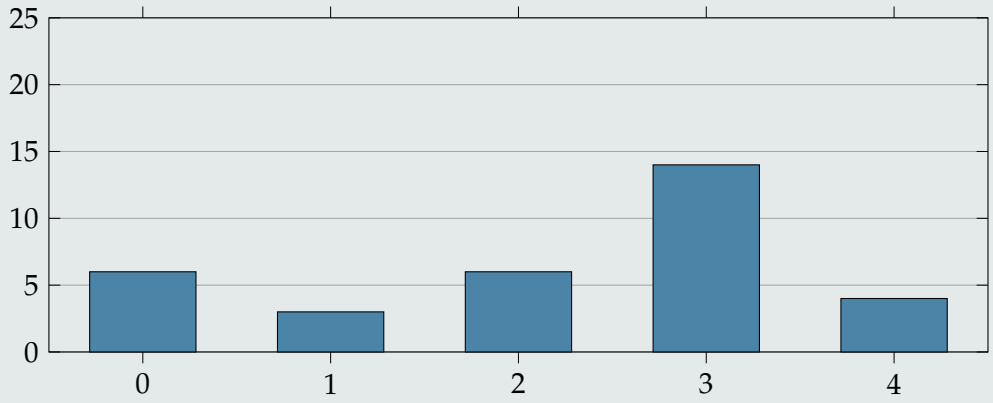


DeFr5x2.2
The difficulty level of the task matched my skill and experience.



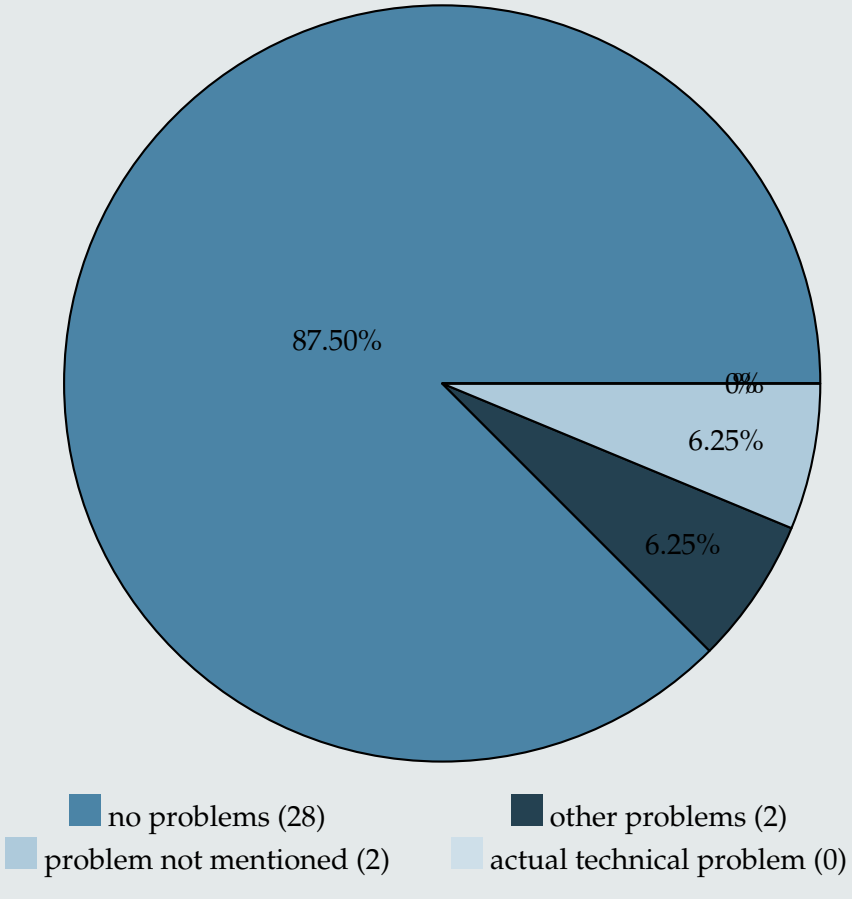
DeFr5x2.3

I felt I had sufficient time to complete the task without feeling rushed.

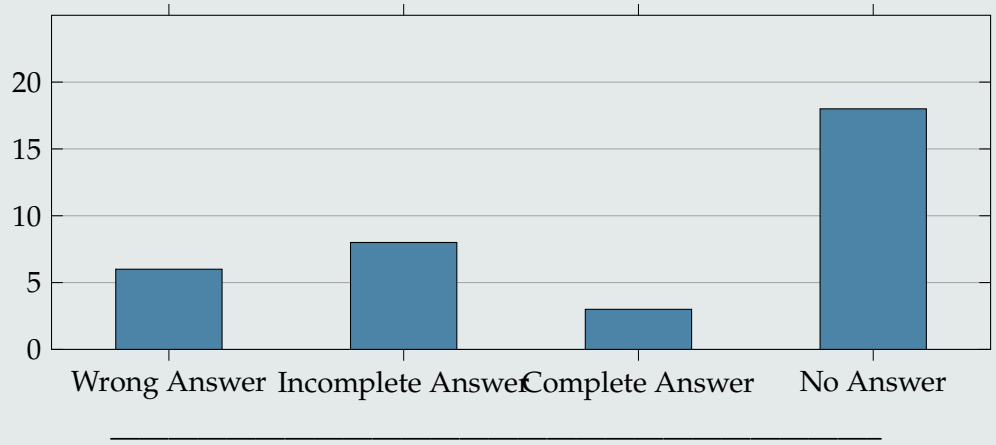


DeTe1x3

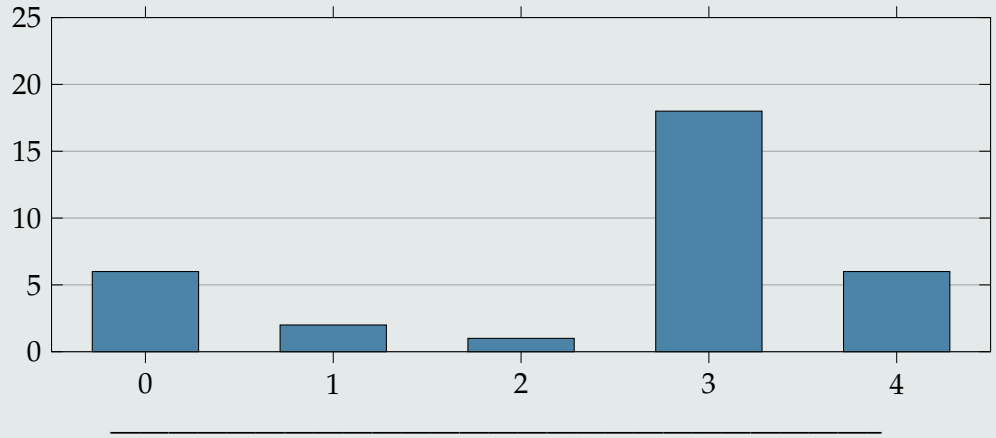
Did you experience any technical difficulties while completing the task?
What was the problem?



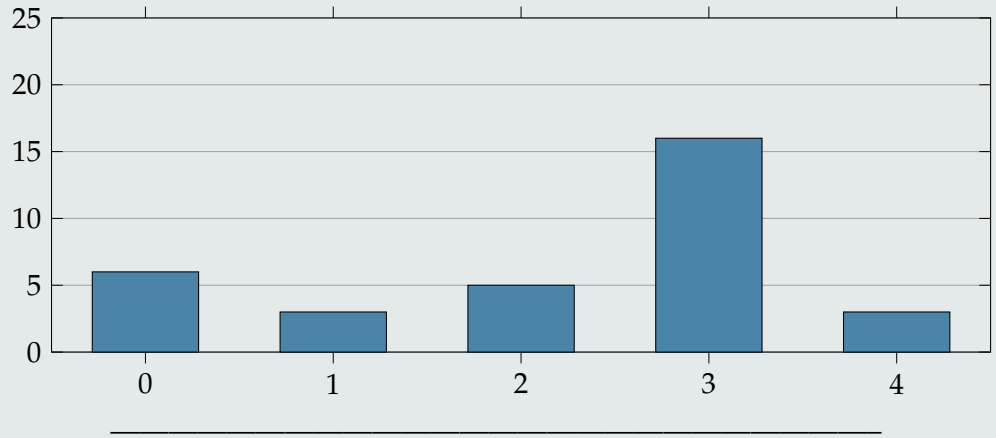
Fr5x3



DeFr5x3.1
The task was clear and unambiguous.

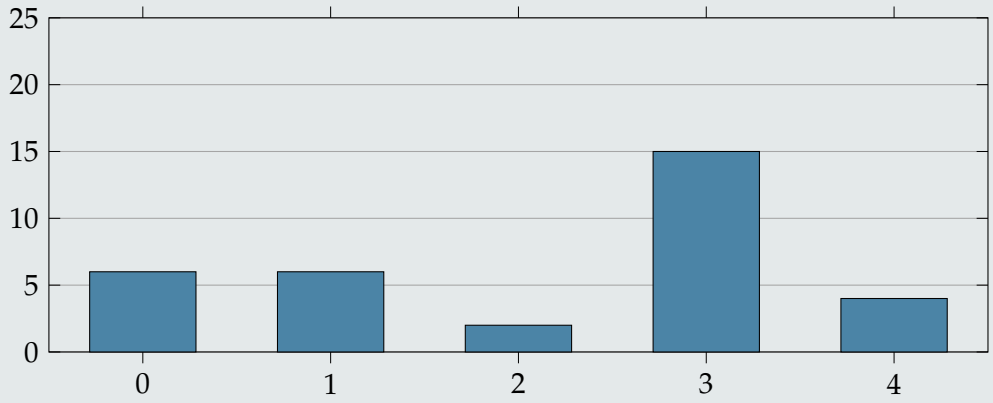


DeFr5x3.2
The difficulty level of the task matched my skill and experience.



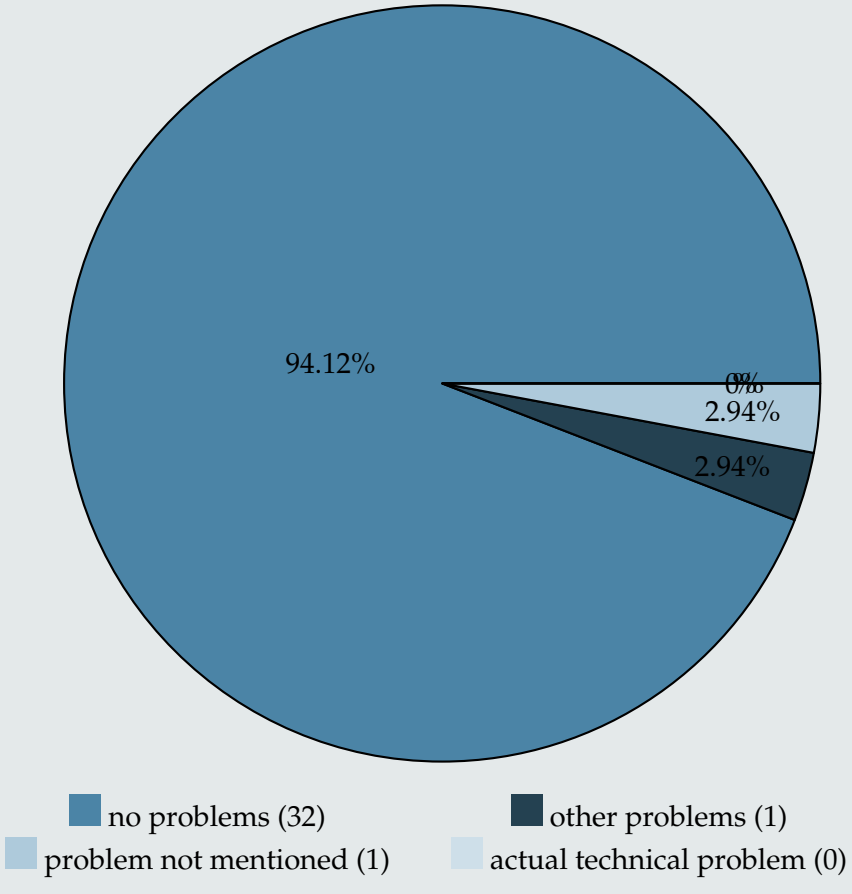
DeFr5x3.3

I felt I had sufficient time to complete the task without feeling rushed.



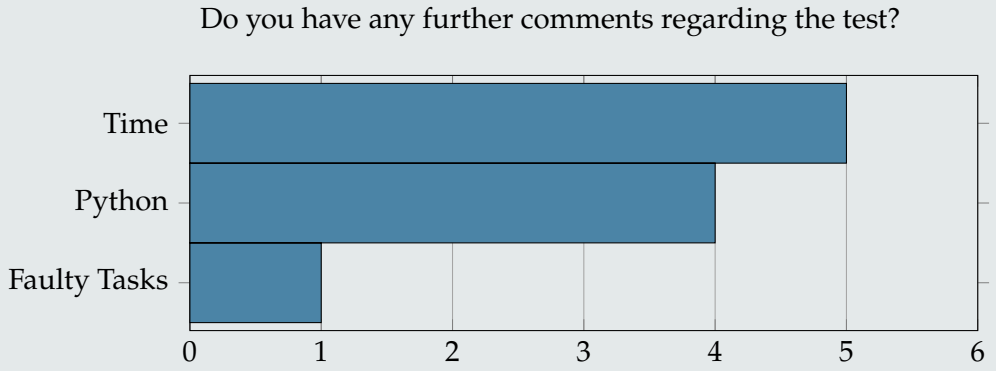
DeTe1x3

Did you experience any technical difficulties while completing the task?
What was the problem?



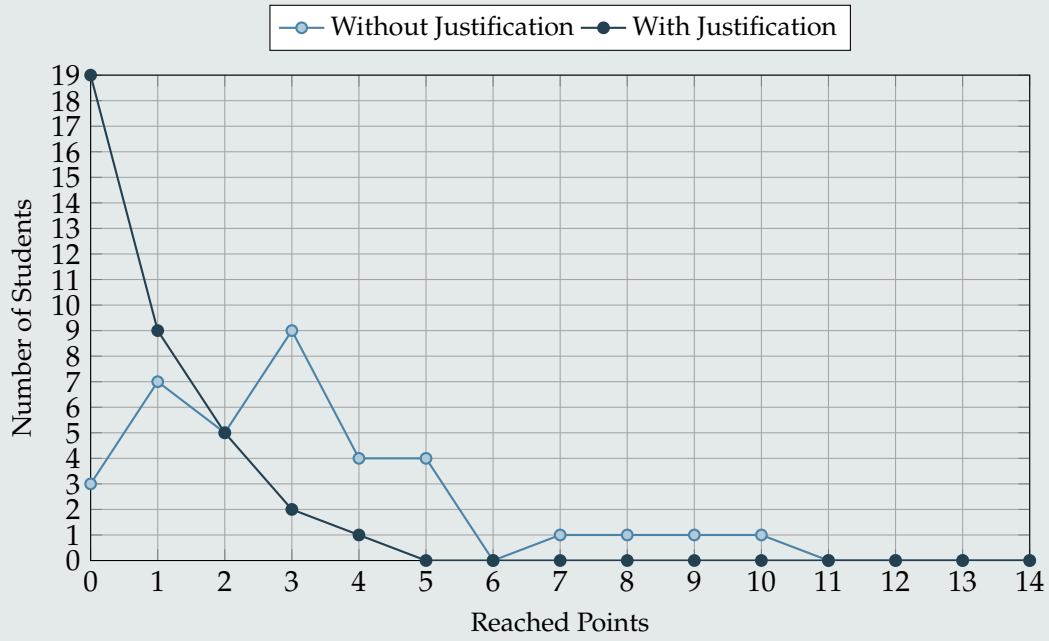
D.3 Overall Comments

Overall Comments



D.4 General Results

Results



Declaration of Authenticity

I declare to Chemnitz University of Technology that I have completed this Master's thesis independently and without the use of sources and aids other than those specified. This thesis is free of plagiarism. All statements that are taken verbatim or content from other writings have been identified as such. This thesis has not yet been submitted in the same or a similar form to any other examiner and has not yet been published.

.....
Author