Bachelor Degree Project

Code Correctness and Quality in the Era of AI Code Generation

- Examining ChatGPT and GitHub Copilot

Authors: Emilia Hansson & Oliwer Ellréus
Supervisor: Welf Löwe
Semester: VT 2023
Subject: Computer Science

Abstract

The use of AI tools for code generation is increasing in popularity, and two of these tools are ChatGPT and GitHub Copilot. These tools could potentially reduce development time and costs for developers and companies, however, ensuring the correctness and quality of AI-generated code is crucial for its adoption. This study conducted a quantitative controlled experiment to evaluate the code generation capabilities of Copilot and ChatGPT in terms of code correctness and quality. The experiment aimed to address research questions regarding the performance of these AI tools. The results indicate that both ChatGPT and Copilot can generate correct code from given instructions, though there is room for improvement. ChatGPT achieved a correctness rate of 87.33%, while Copilot performed slightly better at 89%. Statistical analysis revealed no significant difference in code correctness between the two tools. Regarding code quality, ChatGPT demonstrated impressive performance, with 98.52% of generated lines free from quality rule violations. Furthermore, 80.7% of ChatGPT-generated algorithms had no quality rule violations. Copilot generated correct lines for 94.07% of total lines but only achieved 64.7% of algorithms with no quality rule violations. The statistical analysis showed a statistically significant difference in code quality between ChatGPT and Copilot, indicating that ChatGPT generally produces higher quality code. This research contributes to understanding the capabilities of AI code generation tools and highlights their potential to produce correct and high-quality code.

Keywords: AI Code Generation, ChatGPT, Copilot, Code Correctness, Code Quality

Preface

We'd like to thank our supervisor, Welf Löwe, at Linnaeus University for guiding us through this project. Your help and knowledge have been invaluable, and we appreciate you answering our questions and concerns and reassuring us that we were on the right track. We also want to thank Daniel Toll, who helped us figure out how to make the unit testing of the generated algorithms automatic when we had generations by ChatGPT and Copilot that caused build errors.

Contents

1 Introduction

Recently, the interest in AI has exploded, partly due to the public release of ChatGPT in November 2022 [1]. AI tools have become popular helping hands when programming, and there have even been claims that human developers will become obsolete within a few years [2, 3]. At the same time, Stack Overflow, an online community for developers, has banned content generated by ChatGPT due to the average rate of correct code being too low [4].

This is a 15 HEC bachelor thesis in computer science, comparing the code-generating abilities of ChatGPT and GitHub Copilot. In this study, a controlled experiment was conducted to assess the correctness and quality of the code generated by ChatGPT and Copilot.

Although the code generated by Copilot has been researched in recent years, previous studies have focused more on correctness, efficiency, and validity rather than the quality of the code generated. As for ChatGPT, there have been few to no studies on its coding abilities.

1.1 Background

AI generated code has the potential to reduce time and costs associated with manual coding. If the code generated proves to be correct and of high quality, an AI tool could be a good complement to increase productivity and efficiency for both developers and companies. However, as previous research is limited, code generated by AI tools need additional studies in order to know if the code is correct and of an acceptable quality.

There are many different AI tools capable of generating code, and new and updated models are released frequently. ChatGPT and Copilot are two AI tools that can generate code.

ChatGPT is an AI chatbot developed by OpenAI [5]. While a chatbot's primary role is to imitate human conversation, ChatGPT is incredibly flexible. It has the capability to create web applications, generate song lyrics, write fairy tales, poetry, and student essays, as well as translate texts in multiple languages.

GitHub Copilot is a tool used by developers in integrated development environments [6]. The tool is able to assist developers by suggesting parts of code, such as methods or the completion of existing code. It can also suggest code based on code comments.

Code generation with tools like these have been researched before; however, those studies have been more focused on validity and correctness [7, 9]. The research on code quality for AI generated code has been limited, as well as research on code generation with ChatGPT. Since code correctness and code quality is important in order for the code to be usable and maintainable, both code correctness and code quality will be included in this study.

1.2 Related Work

In 2022, B. Yetistiren, I. Ozsoy, and E. Tuzun [7] analyzed GitHub Copilot's code generation in terms of correctness, validity, and efficiency, and found that it generated 28.7% fully correct code for problems in the HumanEval dataset. A significant portion (51.2%) of the problems were partially correct, and the remaining 20.1% were generated incorrectly. The study also evaluated the impact of input parameters on the code generation and the authors found that the tool is promising, but states that future work is needed to increase accuracy and assess code quality using maintainability and reliability metrics. We are going to assess both the abilities of Copilot and ChatGPT in generating correct code and if any differences can be observed between the two. We are also going to examine how the two tools perform in terms of code quality.

In a recent empirical study, S. Imai [8] found that while programming with Copilot generated more code than human pair-programming, the quality of the code generated was lower. The study evaluated code quality indirectly by observing how many lines of code were removed post-creation, indicating potential issues with the produced code. We intend to assess the code quality of the generated code based on rules chosen from the book *Clean Code* by Robert C. Martin.

In a study from 2021, M. Chen et al. [9] conducted an evaluation of the functional correctness of code generated by various large language models. The study specifically analyzed the performance of models, among them Codex, which powers GitHub Copilot. The authors noted that Codex solved 28.8% of the given problems on the evaluation set. We aim to build upon this evaluation by not only assessing the functional correctness but also considering the quality of the generated code by Copilot. We will also assess the code correctness and quality for code generated by ChatGPT. By examining both correctness and quality, our work aims to provide additional assessment of the code generated by these tools.

In a study from 2022 [10], S. Sengamedu and H. Zhao presented a framework for identifying code quality issues using large language models. They successfully developed a framework that leveraged neural language models to detect code quality problems like repetitive code and unnatural code. While the authors' focus was on code quality identification, they did not specifically use code generated by large language models. Our objective is to assess the code quality and correctness of ChatGPT and Copilot, specifically by manually generating code from these tools. The study provides insights into code quality identification using language models, and our work expands upon it by directly examining the quality of code generated by large language models.

There is active research in the area since a lot of studies mentioned have been conducted in the last few years. All of the related work articles mentioned have been published during 2021 and 2022.

1.3 Problem Formulation

The current state of research in code generation primarily focuses on the functionality of the generated code, neglecting its quality. However, code quality is a crucial aspect in software development as poorly written and difficult-to-maintain code can result in significant costs and wasted time for development teams [11, 12]. In light of this, it is important to evaluate the quality of code generated by AI tools, such as GitHub Copilot and ChatGPT. Previous studies have explored the use of LLMs in identifying code quality issues [10], but there is a lack of research to determine if these models consider code quality during the code generation process.

Code correctness has been evaluated in previous research for code generation with Copilot, but there is limited to no research on the code generation abilities of ChatGPT. By evaluating the two AI tools on equal terms, they can both be assessed individually but also compared to observe any differences between them.

These research gaps highlight the need for further investigation into the potential of AI tools to generate both correct code and high-quality code. To this end, we have identified six research questions:

RQ1: How well does ChatGPT provide correct code from specified instructions?
RQ2: How well does GitHub Copilot provide correct code from specified instructions?
RQ3: How good is ChatGPT at generating code of high quality?
RQ4: How good is GitHub Copilot at generating code of high quality?
RQ5: What differences can be observed between ChatGPT and GitHub Copilot in terms of code correctness?
RQ6: What differences can be observed between ChatGPT and GitHub Copilot in terms of code quality?

1.4 Motivation

The evaluation of AI code generation tools, such as ChatGPT and GitHub Copilot, has the potential to bring significant benefits to developers by reducing time and resources spent on manual coding. The quality and correctness of the generated code are crucial factors in determining the effectiveness of these tools. In cases where the code is found to be of high quality and correctness, companies that employ developers could potentially realize increased productivity and efficiency in their code writing processes. Consequently, these tools have the potential to offer substantial savings in terms of time and resources, provided that the code they generate meets acceptable standards.

The field of code generation has received substantial attention from researchers in recent years, with previous studies primarily focusing on the validity and correctness of the generated code in Copilot [7, 9]. However, there has been limited research on the quality of the code produced by Copilot, and as ChatGPT was released to the public in November 2022, there has been little to no research on its code generation capabilities. With the rapid advancements in AI technology, a growing number of developers are turning to automatic code generation tools [13]. The research area aims to investigate the quality and correctness of code generated by such tools to provide knowledge of how well large language models can generate code from text.

The intended audience for the study is developers and organizations that have a developer workforce and aim to optimize their time and resources while writing code without neglecting code quality and correctness.

1.5 Results

The results of this study will provide insights into the code generation capabilities of GitHub Copilot and ChatGPT in terms of code correctness and code quality. The results are divided into two main categories: code correctness and code quality. Code correctness refers to the ability of ChatGPT and Copilot to provide correct code from specified instructions, and code quality refers to the ability of ChatGPT and Copilot to provide high-quality code. The experiment aims to answer the research questions outlined in section 1.3, and the findings will contribute to a deeper understanding of the capabilities and limitations of AI tools in code generation. The results will also offer valuable insights for future research and development in this field.

1.6 Scope/Limitation

This study was limited to examining two AI tools that have the ability to generate code, and the tools chosen were ChatGPT (the GPT-3.5 model) and GitHub Copilot.

The AI tools were evaluated based on code correctness and code quality. To evaluate the quality of the generated code, we utilized chosen principles from the book *Clean Code* by Robert C. Martin. The principles chosen from the book were limited by the possibility of implementing the rules in the Checkstyle tool. When evaluating the code quality, we decided to assess the overall result of all the code quality rules, rather than examining how the AI tools performed for each individual rule. This decision was made due to time constraints.

We chose to generate code exclusively in Java, which was selected because it is the programming language used in the Clean Code book. We specifically chose to generate algorithms selected from the book *Beginning Java Data Structures and Algorithms* by James Cutajar. The algorithms we chose were based on an article listing 25 common algorithms every programmer should know [14]. The selected algorithms were Binary Search Algorithm, Binary To Decimal Algorithm, Breadth First Search Algorithm, Knapsack Algorithm, Merge Sort Algorithm, and Quicksort Algorithm.

1.7 Target Group

This study is aimed at researchers, developers, and organizations interested in the quality and correctness of code generated by AI. For instance, companies or developers seeking to integrate ChatGPT or Copilot into their development process to reduce the time and resources spent on manual coding may find this study useful.

1.8 Outline

This report is organized as follows: In Chapter 2, we discuss the research project, research methods, reliability, validity, and ethical considerations. Chapter 3 provides the theoretical background. In Chapter 4, the implementation of the controlled experiment is described. Results are provided in Chapter 5, and the analysis of the results is in Chapter 6. Chapter 7 provides a discussion about the results related to the research questions, and lastly, Chapter 8 concludes the study and provides recommendations for future work.

2 Method

A quantitative controlled experiment was conducted to evaluate the code generation capabilities of GitHub Copilot and ChatGPT in terms of code correctness and code quality, to be able to answer the research questions. This chapter describes the activities carried out in the experiment and discusses the reliability and validity, as well as the ethical considerations.

2.1 Controlled Experiment

In this section, the activities performed in the experiment will be described in detail including the process of selecting algorithms, collecting data, performing unit testing, and checking code quality. We will also explain the data extraction process and the statistical analysis that was performed. The implementation of the experiment is described in further detail in Chapter 4. The book *Experimentation in Software Engineering* [15], authored by Claes Wohlin, Martin Höst, Björn Regnell, Per Runeson, Magnus C. Ohlsson, and Anders Wesslén, served as a valuable resource for conducting the controlled experiment.

2.1.1 AI Tools Selection

GitHub Copilot and ChatGPT were selected as the AI tools to generate the algorithms from. ChatGPT was chosen based on its increased popularity since its release and the potential for code generation, which is an area that lacks extensive research. The GPT-3.5 model was used when generating algorithms with ChatGPT. GitHub Copilot was chosen as it has been previously researched. This approach combines both a popular but less researched tool and a tool that has been studied before.

2.1.2 Algorithm Selection

Six algorithms were selected from the book *Beginning Java Data Structures and Algorithms* by James Cutajar. These algorithms were:

- Binary Search Algorithm [16]
- Breadth First Search (BFS) Algorithm [17]
- Merge Sort Algorithm [18]
- Quicksort Algorithm [19]
- Knapsack Algorithm [20]
- Binary To Decimal Algorithm [21]

The selection of the algorithms was based on an article listing 25 common algorithms every programmer should know [14]. The first five algorithms were originally selected from the article. However, as we were unsuccessful in finding a deterministic test case for the Depth First Search (DFS) algorithm where the outcome was independent of the internal implementation, we chose to leave out the DFS algorithm and select another one. We randomly selected the 0-1 Knapsack algorithm from the repository of the book's code example on GitHub [22]. However, the algorithm diversity was taken into account when choosing the replacement algorithm for DFS. In order to further increase diversity, we opted to also include the Binary To Decimal Algorithm.

The book implementation of these six algorithms was used to formulate the instructions given to ChatGPT and Copilot. The unit tests used to evaluate the correctness of the algorithms generated by ChatGPT and Copilot were also developed from the implementation of these algorithms as described in the book.

2.1.3 Code Generation

The data was collected by generating the chosen algorithms with ChatGPT and Copilot. Each of the algorithms was generated 50 times for the same set of instructions. The repetition of 50 generations per algorithm was imposed due to time constraints. Each algorithm had one specific instruction that was used when generating the algorithms with ChatGPT and Copilot. The instructions were the same for every generation to ensure that the result of the generated algorithm was not affected by inconsistent formulations. The instructions used in the study to generate the algorithms can be found in Appendix 1.

The code generation with Copilot and ChatGPT differs from each other (the AI tools are described in Chapter 3). Copilot is integrated into an IDE of choice, while the code generated from ChatGPT is made through a chat web interface. We did multiple tries with different formulations for the instructions but ultimately decided to be quite specific to be able to test every generation of a specific algorithm with a unit test that was based on the book algorithms. As a result, we used instructions that specified the name of the class, method names, input arguments and return types. The reason that we chose to use code comments for both tools is that ChatGPT is more flexible, and therefore, we decided to create instructions based on what works well for Copilot.

To make sure that the instructions were well-defined and that failing algorithm generations were not due to a poor formulation, we made sure that both Copilot and ChatGPT could generate at least one correct generation of each algorithm for the given instructions before the data collection started.

2.1.4 Correctness Evaluation

Unit tests were created for each algorithm to evaluate the correctness of the generated code. The unit tests were written in JUnit on the basis of the chosen algorithms as implemented in the repository containing the book examples. Figure 2.1 shows the procedure of implementing the unit tests.

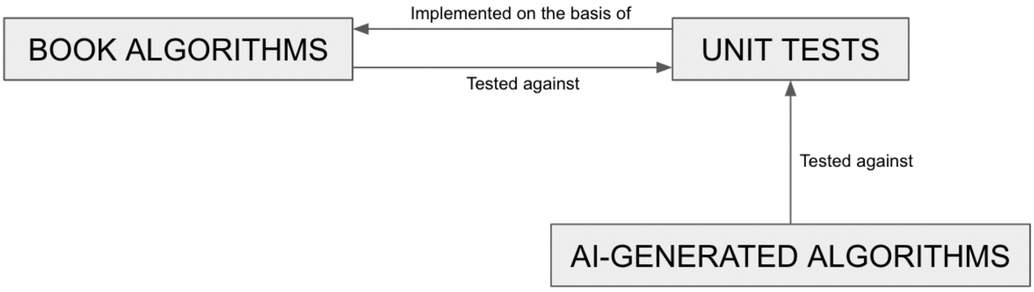


Figure 2.1: The method of implementing the unit tests

The unit tests were used to ensure that the code generated by ChatGPT and Copilot produced the expected output for a given input, and every algorithm had at least one test.

2.1.5 Quality Evaluation

Checkstyle was selected as the tool for evaluating the quality of the generated code based on predefined rules. This tool was used to assess the generated code based on selected principles from the book *Clean Code* by Robert C. Martin. The principles were chosen based on their feasibility for implementation using Checkstyle. A motivation and reference to the rules implemented can be found in Appendix 2. The principles implemented in Checkstyle were:

- The files should not be over 500 lines long.
- A line should not be more than 120 characters long.
- Magic numbers should be hidden behind constants.
- Functions should not be more than 20 lines long.
- Functions should not have more than three arguments.
- There should not be nested loops of a depth of more than one level.
- There should not be more than one statement per line.
- There should not be any inner assignments.

2.1.6 Statistical Analysis

To examine whether ChatGPT and Copilot can produce correct and high-quality code, descriptive statistics will be used to analyze and present the data. The independent variables are the AI tools, ChatGPT and Copilot, and the dependent variables are the code correctness and the code quality.

The ability of each AI tool to generate correct code will be evaluated by comparing the number of generated algorithms that passed all the unit tests to the total number of generated algorithms.

To assess code quality, frequency tables and corresponding graphs will be generated for each tool, displaying the number of generated algorithms that had a specific number of quality errors. Additionally, the total number of correct lines was compared to the total number of lines containing quality errors.

To investigate whether there are differences between ChatGPT and Copilot in terms of code correctness and code quality, hypothesis testing will be conducted to determine whether there is a statistically significant difference between the two tools. To ensure the validity of the results, normality tests will be performed and data will be examined graphically before conducting the hypothesis testing. A two-tailed non-parametric test, the Mann-Whitney U test, will be performed on independent groups. The Mann-Whitney U test compares the distribution of two independent groups of data. It tests the null hypothesis that the two groups have the same distribution, and the alternative hypothesis that one group has a different distribution than the other [23].

The null hypothesis for the research question on code correctness is that there is no difference between ChatGPT and Copilot (or that differences are random), while the alternative hypothesis is that there is a systematic and significant difference between the two tools. Similarly, the null hypothesis for the research question on code quality is that there is no (or random) difference between the two tools. The alternative hypothesis is that there is a systematic and significant difference between them.

2.2 Reliability and Validity

In the controlled experiment, we will evaluate the ability of ChatGPT and Copilot to generate correct and high-quality code. By using defined rules from the book *Clean Code* by Robert C. Martin, we limited the study to define high code quality as code that fulfills these requirements and therefore reduces any bias or construct validity issues regarding the assessment of the quality. As we also used algorithms from *Beginning Java Data Structures and Algorithms* by James Cutajar, we addressed construct validity concerns as what was deemed to be correct code was defined by the book algorithms.

To increase reliability, the testing and checking of the generated algorithms were done automatically using automated testing and checking tools such as JUnit and Checkstyle. To complement the automation tools for special cases, automated scripts were written and used in favor of manually checking these cases. Furthermore, in order to reduce the likelihood of errors in our test cases, we consulted an algorithm expert who teaches at Linnaeus University. Their expertise helped to improve our understanding of the output produced by specific algorithms. As we also had defined specific instructions to give to Copilot and ChatGPT at every generation of an algorithm, the reliability increased further by ensuring consistency of how the code was generated.

As such, the results can be replicated in future research as we used automation tools like JUnit and Checkstyle with defined quality rules and expected behavior to perform the testing and quality checking of the generated algorithms, as well as having predefined instructions. The entire code base, including scripts, tests, and configurations, is publicly available on GitHub. See Appendix 4 for links to the GitHub repositories as well as hands on instructions on how to repeat the research project. However, as updated models emerge, it is a possibility that the GPT-3.5 model will not be available in the ChatGPT web interface when repeating the project in the future.

An internal validity threat is that the instructions, to some extent, affect the results of the code quality. In the instructions, we specified what parameters the class and the public methods should be implemented with. One implemented quality rule is that a method should not have more than three parameters. As we specified how many parameters should be used in the instructions, we controlled the outcome for this specific rule and did not let the AI tool decide. However, this specification was only limited to the requested class instantiation and public methods in the instruction and did not affect any additional methods that the AI tool decided to implement. The reason for specifying the number of parameters was to be able to check code correctness using automated unit tests. As the same instruction was used for both ChatGPT and Copilot when an algorithm was generated, it did not affect the result when analyzing the differences between the tools as both of them were equally affected.

Tests can’t completely guarantee correct code as they may miss certain bugs or errors, which could affect the results of the study. This limitation could lead to incorrect or incomplete conclusions about the effectiveness of the tested code in achieving the research goals, and therefore represents a threat to internal validity.

To minimize the potential internal validity threat of errors in the scripts used in the project, the scripts were manually tested multiple times during and after implementation. A documented test was conducted, and a link to the test documentation can be found in Appendix 4. All the scripts used in the project can be inspected as they are publicly available on GitHub, and links to these repositories can also be found in Appendix 4.

The limitation of generating only six Java algorithms raises concerns about the external validity and generalizability of the findings. While our study focused on examining the code correctness and quality of these specific Java algorithms, it is important to acknowledge that generating different types of code, such as web components or code in other programming languages, may yield different results. Furthermore, our approach involved using predefined instructions and selecting the first solution provided by the AI tools, which may not align with the common way of consulting these tools for producing accurate results. Nevertheless, our study aimed to assess the ability of ChatGPT and Copilot to generate correct and high-quality code based on given instructions within the scope of Java algorithms.

2.3 Ethical Considerations

When considering the use of code generation tools, it is important to address certain ethical considerations. It should be noted that security issues in the generated code were not specifically checked in this experiment. Therefore, it is important for readers to understand that while the results indicate the potential of these tools for code generation, security concerns have not been taken into account. It is recommended to undergo thorough security testing and analysis to identify vulnerabilities. Consequently, the decision to use these tools for code projects should consider multiple factors beyond code correctness and quality, including robust security measures.

The contribution of automated code generation systems to the development process can be a controversial topic. While these tools offer efficiency and productivity benefits, they may raise concerns regarding accountability and the preservation of human creativity in software development. Ongoing discussions and guidelines are necessary to strike a balance between the advantages of automation and the preservation of human involvement in the development process.

The adoption of code generation tools on a wide scale can have implications for software developers and related job markets. The use of these tools can shift responsibility from developers to AI systems for manual coding tasks. This raises ethical concerns regarding potential job displacement for human developers. If machines replace developers in significant numbers, there could be a reduction in job opportunities within the software development field. It is important to consider strategies for managing potential job displacement and establish ethical frameworks and policies that protect the interests of human workers if transitioning to automated code generation systems.

3 Theoretical Background

This chapter presents the theoretical background of the study's concepts. It provides detailed descriptions of important concepts, including AI, natural language processing, ChatGPT, Copilot, unit testing, code quality, and clean code. Chapter 1.2 introduces related work, while Chapter 1.3 identifies the research gap.

3.1 Artificial Intelligence (AI)

Intelligent computer systems have been created in order to make computers able to behave like human minds [24]. These systems allow computers to interpret speech, play games, and see patterns.

In order for an AI system to be usable for a specific task, it needs to be trained [24]. There are two common ways to train AI. The first alternative is with human supervision, and the other is without human supervision. Both training methods use large amounts of data that the AI uses to see patterns that will later be used for decision-making. In the human supervision method, the AI is encouraged to choose good decisions by humans.

There are two types of AI, strong and weak AI [24]. Strong AI is AI that can do tasks that it has never been trained on. This type of AI has intelligence on the same level as humans. However, researchers have not been able to achieve this kind of intelligence. Weak AI, on the other hand, has been more successful. It is AI that has been trained to do a specific task. This could, for example, be self-driving cars, voice assistants, or spam filters. This type of AI is usually very good for the task it is built for. However, they are only a simulation of human intelligence and are therefore weak AI.

Machine learning is a specific type of AI [25]. It is used to create algorithms that are able to find patterns and create recommendations from them. They are both trained by the initial training data and new data they get over time. Since machine learning is able to train and improve algorithms over time, it becomes better at what it is supposed to do.

Deep learning is a more advanced type of machine learning [25]. Similar to machine learning it uses human supervision in order to produce accurate results, however deep learning does not need as much supervision as machine learning without deep learning. It uses neural networks to process data. Neural networks work in a similar way as the neurons in human brains. They are able to make decisions based on learned knowledge from previous data. For example, deep learning can learn patterns in inputs over time, such as similar objects in images.

Algorithms that are able to create new content, such as text, images, or videos, using artificial intelligence are called Generative AI [26]. Generative AI uses machine learning and neural networks to create new content based on training data and/or user input. Generative AI has many use cases. For example, it can be used to generate text for companies, correct code, or answer questions.

3.2 Transformer Architecture

Transformer architecture is an architecture used for natural language processing [27]. Both GitHub Copilot and ChatGPT are using this architecture [27, 28]. Section 3.3 describes GitHub Copilot, and section 3.4 describes ChatGPT. The architecture is divided into two parts: an encoder and a decoder [27]. The encoder processes the input by creating vectors of the input, and the decoder then processes the output .

Unlike other architectures, such as recurrent neural networks that are focusing on feedback loops, the transformer architecture focuses on attention and self-attention [28]. In order to find out the context of a text, self-attention is used. This is used to find out the meaning of words that can have different meanings in different contexts. The architecture is able to do this by paying attention to the other words in the text.

Query and key vectors are generated for each word [28]. This is used to find out if another word has a relevant context. For example, if the query of a word matches the key of another word, it means that the context is related. For related contexts, a value vector is created. This vector is used to relate the words to each other.

The ability to find related contexts is one of the advantages of this architecture [28]. Another advantage is its ability to use parallelization. Parallelization helps the architecture to process more data faster. For example, during the same amount of time, it can be trained on more data compared to other architectures.

OpenAI has developed transformer models using this architecture [28]. They are called Generative Pre-trained Transformers or GPT. These models are used for language modeling tasks. The first GPT model was originally used to guess the following word. Later models have been created such as GPT-2 and GPT-3 which can be used for different language-based tasks. For example, GPT-3 can be given input and then generate a response.

3.3 GitHub Copilot

GitHub Copilot is a tool focused on completing code [29]. It is used as a plugin in integrated development environments (IDEs) such as Visual Studio Code. Copilot has specifically been trained on data with computer code. This data has been collected from all over the internet and on GitHub. Since the tool is accessible directly in the developer's IDE, it is easy and fast to use in order to become more productive as a developer. The code completion suggestions are generated directly in the IDE, and the developer can accept them if they are suitable. Since this tool focuses on code completion, it is best suited for solving smaller parts of code during development.

GitHub Copilot has many strengths [29]. It is good at generating code snippets and suggestions based on the code that has already been written. It also makes developers more efficient since they do not have to write all the code by themselves when using Copilot. Lastly, the use of Copilot does not require any separate software since Copilot is built directly into the IDE that the developer is already using.

It also has some weaknesses, such as being less good at generating more advanced code with higher complexity [29]. There is a risk that Copilot generates code with faults. Sometimes there might be better solutions than the one that Copilot suggests.

Copilot uses the Codex language model [30]. Codex is based on GPT-3, which is a model created by OpenAI. The difference between Codex and GPT-3 is that Codex has been trained only to generate code.

3.4 ChatGPT

ChatGPT is a chat-based AI tool that is capable of generating code [29]. It is developed by OpenAI, and it was released to the public in November 2022. ChatGPT is not specifically built in order to become a code-generation tool; instead, it is designed for general language tasks. The language model used for ChatGPT is not only trained on computer code languages, but it is mostly trained on human language text.

Since ChatGPT is trained in human language, it is very good at explaining computer code and writing code based on human-written descriptions [29]. This makes ChatGPT better at larger code tasks compared to Copilot.

ChatGPT has many strengths [29]. It is easy to use for humans without a lot of coding experience due to its human language chat interface. It is also good at explaining solutions and addressing follow-up questions. Another advantage is that ChatGPT is not only limited to answering coding questions, which can be useful.

Some of ChatGPT's weaknesses are the ability to solve complex problems [29]. Then the responses may become inaccurate. This can become a problem if the user does not understand the suggested solution and believes that the solution is correct.

The underlying language model of ChatGPT used in this study is based on GPT-3.5 [31]. The model is trained on many things, such as websites, articles, and books. This has made the model able to generate text like humans.

3.5 Unit Testing

Unit testing is the lowest level of tests that are written in order to test code [32]. The application is divided into small parts called units that can be tested. Each unit is usually a single function that is tested separately from the rest of the code. Testing the code in small units helps developers and testers to confirm that each part of the application is functionally correct.

Unit tests do not only have to be a single function; they can also be a line of code or an entire class [33]. Depending on what is tested, the unit test might look a little bit different. If a single method is tested that divides two numbers, it is easily tested by calling the function with two numbers. The expected return value from the function should then equal the value of the two numbers divided by each other. Small unit tests, like this example, are considered to be the best since they test the code on a very low level with a lot of detail. Another advantage of small tests is that they can be executed fast compared to a few large tests that take a lot of time.

There are some advantages of using unit tests [32]. First, the unit tests give developers and testers a better understanding of their code base. This helps them to understand how to solve upcoming problems faster. It also helps them to find issues faster since the unit tests are the lowest level of testing that is done first. Lastly, since each unit test only tests a single part of the code, they are easy to reuse if the parts are reused in other projects. Then the developers and testers do not have to write new tests for the same code again.

Unit tests can be used for many things during code development [33]. They can be used for the code design technique test-driven development (TDD). TDD means that the developer must write a unit test for a unit before it is implemented into the code. After the unit test has been written by the developer, the unit can be implemented. This is not the usual way of writing code, but it makes it easier to refactor the code in the long term, since a test is already written that should pass. If the code is refactored and the test fails, the developer will immediately see the failed test and fix the issue.

Checking is another way of using unit tests [33]. Checking means that the developer writes a test after the code has been implemented in order to confirm that it works as expected. Both TDD and checking are equally useful since both test if the code is doing what it should do. The only difference is when tests are written. It does not matter if the test is written before or after the initial implementation if the code is refactored.

Since the unit tests are testing small parts of the code, they can also be used to describe what the code does [31]. Therefore, unit tests become a part of the code documentation. When the code is updated, the unit tests describe what the implementation does by themselves.

A framework used for automatic unit testing is JUnit [34]. The JUnit framework includes many different features. Tests are written by writing test files that can be executed by JUnit automatically. The results of the tests are written in reports in order to get an overview of the results. Tests (also called test cases) can be grouped together in test suites. This is useful when test cases are related to each other.

The test files are Java files [34]. In the test file, the JUnit framework is imported, as well as the package of the file to be tested. A test class is created for the test case. In the class, methods are added in order to verify that the source code works as expected. For example, a method called `test` can be created in which the JUnit method `assertEquals` can be called. This method has two parameters, one for the input data and one for the expected result.

3.6 Code Quality

Code quality is a way to measure how well-written code is [35]. When assessing code quality, there are a few important factors to consider. First, the code has to be functionally correct without bugs. It also needs to be understandable and consistent. The code must also be well-documented, secure, and testable. The tests must pass without finding any errors or bugs. Code written to be reusable is also a parameter that should be considered when measuring code quality.

There are many reasons that make code quality important [35]. Readability is hugely affected by the code quality level. Code of high quality is much easier to read compared to low code quality. This simplifies the process of code comprehension for developers, whether it's their own code or that of their colleagues.

Maintainability is another thing that is affected by the quality of code [35]. The quality can affect how long the code can be used. High-quality code is easier to improve over time and can therefore be used longer than low-quality code that has to be replaced every time something has to be changed. Even if changes do not have to be made, the higher code quality will require less change because of its already good quality.

The code quality also determines the transferability of the code [35]. This is the ability to transfer code to other projects. Low code quality requires a lot more work compared to high-quality code since it does not require as much work to get the code working after transferring it.

Costs are highly affected by code quality [35]. Code that is of high quality lasts much longer. Because of this, there is less need for rewriting old code since it is still working. If a lot of the code needs to be rewritten regularly, the costs go up, and the developers must spend more time on writing code doing the same thing again.

There are many ways to improve code quality [36]. Using a coding standard is one example. A coding standard is a ruleset for writing code [37]. It includes guidelines and best practices about how the code should be written. These rules can be used in order to make the code quality consistent in the entire code base since all the developers are following the same code standard.

Another way to improve code quality is by analyzing the code [36]. When the code has been written, it should be analyzed before it is handed over to be code reviewed. The results from the analyzer can then be used to improve the code before the review process. Automatic analyzers such as Helix QAC or Klocwork can be used in order to automate this process.

Code quality can also be improved by code review [36]. A code review is manual and therefore it must be done carefully in order to improve the code quality. By taking best practices into consideration when reviewing, the code quality can be improved by the feedback from the reviewer.

Refactoring is also an important process in order to improve code quality [36]. Refactoring is the process of improving code and code quality by lowering the complexity of existing code.

The entire code base should not be refactored at the same time; instead, it should be a continuous process over time.

Checkstyle is a tool used by developers to verify that code has been written according to a coding standard [38]. The tool can be configured according to the developer's needs by configuring it using configuration files. This gives Checkstyle the ability to be configured for any coding standard. It can find layout, formatting, and design issues according to the configuration file.

Standard checks are provided with Checkstyle [39]. These checks can be used to check the source code for different types of issues. Customized checks can also be written in order to create specific rules for Checkstyle [40]. This makes Checkstyle customizable for any coding standard.

Checkstyle can be added to a coding project in multiple ways [41]. It can be integrated using build tools such as Maven or using IDE plugins. When Checkstyle is executed, a report is generated. In this report, all found issues according to the check's configuration are listed.

Both already existing Checkstyle configurations and custom configurations can be used [41]. For example, there are already two configurations included in Checkstyle: Google and Sun configurations. When creating a custom configuration, an XML file is created. The configuration is divided into different module XML elements. The root module element has the name attribute "Checker." This module is responsible for defining properties that all the other modules in the configuration use. TreeWalker is a child module of the Checker module. It is responsible for checking each Java file by using the child modules of itself. The TreeWalker's child elements are different types of check modules. Each module has a name of a check, and property elements can be added as child elements. A property has two parameters, the name of the property and a value that can be changed.

3.7 Clean Code

According to Cetin Ogut, the author of the article *What does Clean Code mean,* clean code is code that has been written in a way that it is possible to read, change, expand, and maintain easily [42]. If the code base is not kept clean, costs and development time will increase over time as the project becomes more complex. The complexity makes the code hard to understand, and therefore, the developers must spend more time understanding how the code works. Some of the key points in clean code are readability, understandability, and maintainability [43].

Readability refers to the ease with which code can be read by other developers, as well as by oneself when reviewing it at a later stage [44]. One example of improving readability is to use names that are easily understood by humans. Instead of naming a variable that stores the name of a cat to `x`, the variable name `catName` would be preferred. Readability is important as it simplifies comprehension and maintenance of the code. Additionally, when code is easy to read, debugging becomes less challenging in case any problems arise.

Understandability, on the other hand, refers to the ease with which code can be *understood* by other developers [45]. The code should be clear and well-organized, and not be too complex. It is important to improve understandability as, when developers have an easier time understanding the code, it will be easier to maintain, modify and extend.

Maintainable code refers to code that could easily be changed, modified or extended over time, and allows developers to efficiently make changes to the codebase, fix bugs, and add new features without the risk of causing any errors elsewhere in the code base if the code, for example, has many dependent modules [46].

Clean Code is a book written by Robert C. Martin that emphasizes the importance of writing clean and maintainable code. It offers practical advice and principles for improving code quality by making the code more understandable and readable. According to R. C. Martin, writing clean code is similar to art [47]. Like art, most people can see when code is of good quality and when it is not. In order to write clean code the programmer needs to have something he calls code-sense. It is the ability to find poorly written code and find a solution to improve the code. The *Clean Code* book was chosen to be used as a base for assessing the code quality of the generated algorithms in this study. However, as the study was limited to using only a subset of rules from the book, only the rules used in the study will be highlighted in the next paragraphs.

To write functions that are readable, understandable and maintainable, rules from the *Clean Code* book can be applied. One of these rules is that functions should be small [48]. It is not ideal for functions to span 100 lines. In fact, it is generally recommended to keep functions significantly shorter, preferably no more than 20 lines in length. By keeping functions short, the code becomes easier to comprehend. Therefore, nested blocks of statements should be minimized, and the indent level should be kept low, typically no more than one or two levels.

Another rule to enhance code readability and understandability is to limit the number of parameters a function accepts [49]. Ideally, a function should have no more than three parameters. Having a shorter parameter list simplifies testing, as it reduces the number of test cases required to cover all possible combinations of inputs.

To maintain clean code, it is crucial to consider the formatting of the code. One aspect to consider is the vertical size and the ideal length of a source file [50]. While there is no absolute rule, it is highly advantageous to aim for smaller files, with a maximum of around 500 lines. The reason is that smaller files tend to be more understandable than larger files. Another aspect is the horizontal size which refers to the width of a line [51]. The width of a line should generally not exceed 120 characters to avoid having to scroll to the right when viewing the code on a screen.

Something common for all of these principles is that they do not focus on running the code [42]. Instead, they focus on the readability, understandability, and maintainability of the code base. These things are important for developers to become efficient during development . However, it is not a good idea to only focus on clean code since it is not uncommon for clean code to be inefficient and slow [52]. Therefore the developers must keep a balance between performance and keeping the code clean.

4 Controlled Experiment – Implementation

This chapter describes the implementation of the controlled experiment and the process of data collection by code generation, unit testing, and code quality checking. The process of extracting the data to prepare for statistical analysis is also described, as well as all the scripts implemented in the project. All the code used in this project is publicly available on GitHub, and the links to the repositories can be found in Appendix 4. You can also find a link with instructions on how to conduct the experiment. The results of the experiment are presented in Chapter 5 and analyzed in Chapter 6.

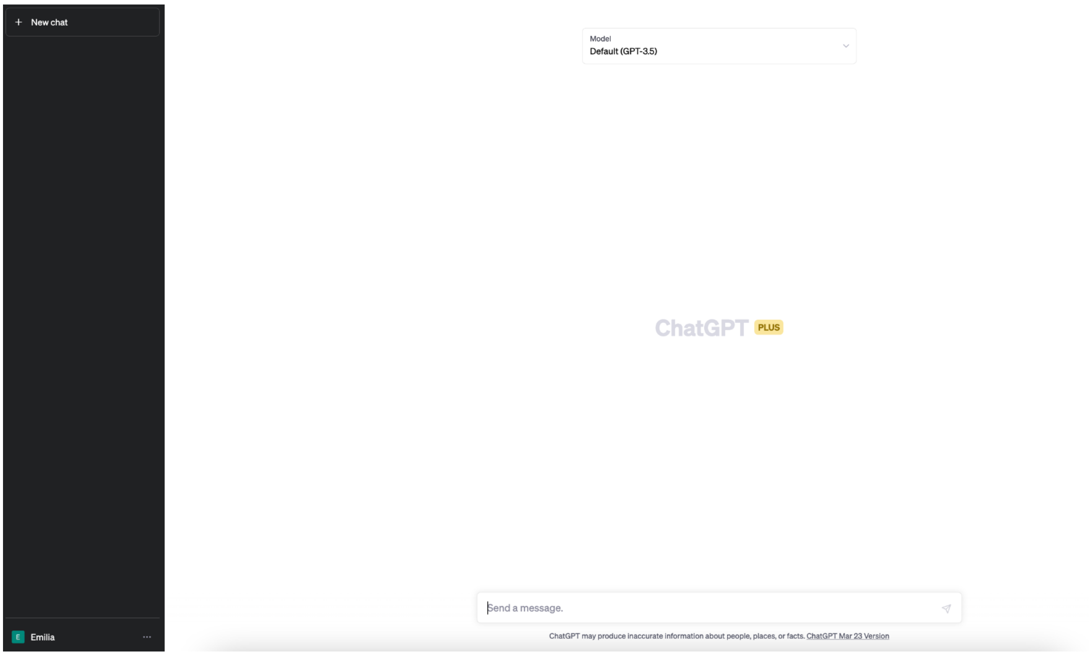
4.1 Code Generation Process

The data was collected by manually generating code from ChatGPT and Copilot and each algorithm was generated 50 times. The following sections describe the code generation process.

4.1.1 ChatGPT

Generating code with ChatGPT was done through its web interface, and the model that was used to generate algorithms was GPT-3.5. ChatGPT uses an input field in order to generate a response. In the input field, code instructions were added in order to generate the algorithms. The construction of the instructions is described in detail in Chapter 2.

For each new generation, a fresh chat window was created within ChatGPT's web interface. Neglecting this action would have resulted in ChatGPT relying on earlier inputs or responses when generating solutions for the current chat. Creating a new chat for each generation ensured that ChatGPT generated an algorithm that was not influenced by previous generations. Figure 4.1 shows a newly opened chat window in the ChatGPT interface.

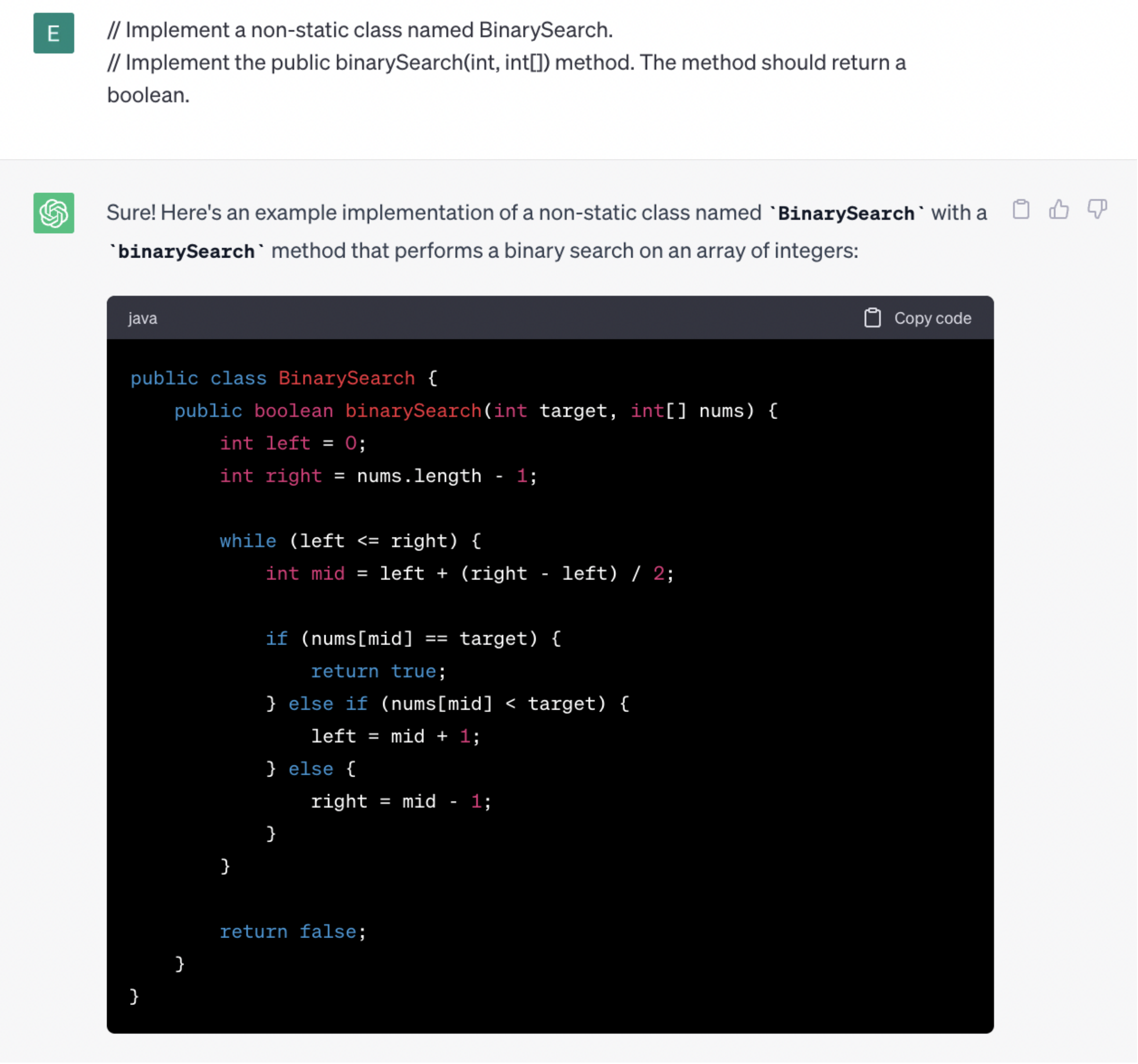


Model
Default (GPT-3.5)

ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT Mar 23 Version

Figure 4.1: New chat in ChatGPT with default model (GPT-3.5).

Once a new chat had been created, the instruction was copied and pasted into the "Send a message" input field. Upon pressing the send button, ChatGPT started the algorithm generation. In addition to the algorithm, ChatGPT often provided an explanation of the code. Figure 4.2 shows the code generating process for the binary search algorithm using ChatGPT.



// Implement a non-static class named BinarySearch.
// Implement the public binarySearch(int, int[]) method. The method should return a boolean.

Sure! Here's an example implementation of a non-static class named `BinarySearch` with a `binarySearch` method that performs a binary search on an array of integers:

```java
public class BinarySearch {
    public boolean binarySearch(int target, int[] nums) {
        int left = 0;
        int right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return true;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return false;
    }
}
```

Figure 4.2: ChatGPT code generating process for the binary search algorithm.

These suggestions were then stored in the Java project in order to be tested for code quality and correctness.

4.1.2 Copilot

As Copilot is not able to generate code from an input field in the same manners as ChatGPT, there were some prerequisites. First, the Copilot extension was installed in Visual Studio Code. Then, an empty Java project was created. Figure 4.3 shows the created Java project.

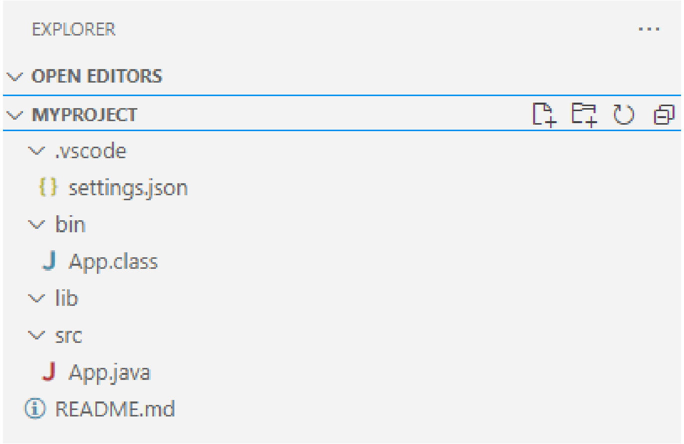
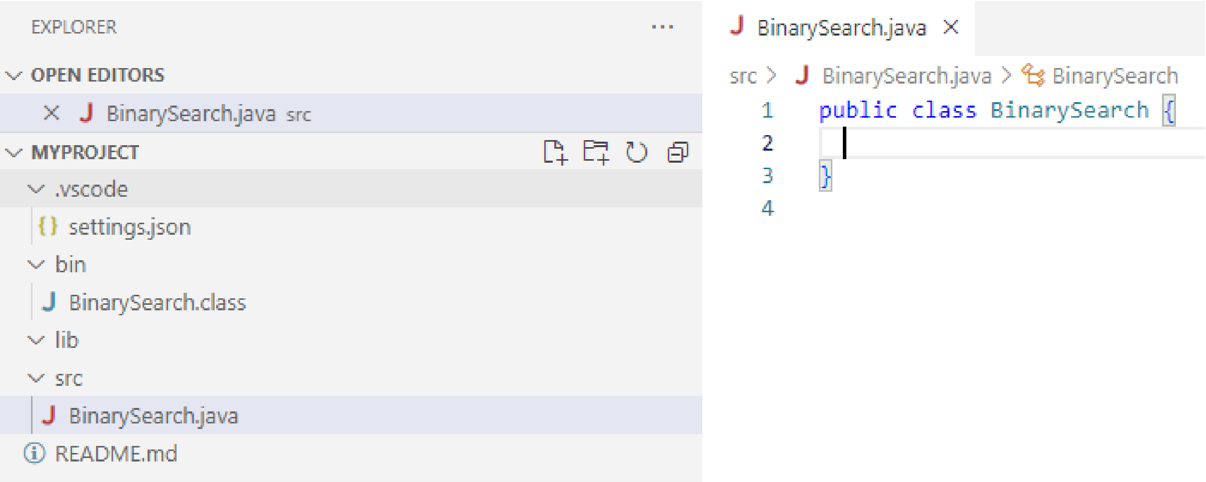


Figure 4.3: Empty Java project.

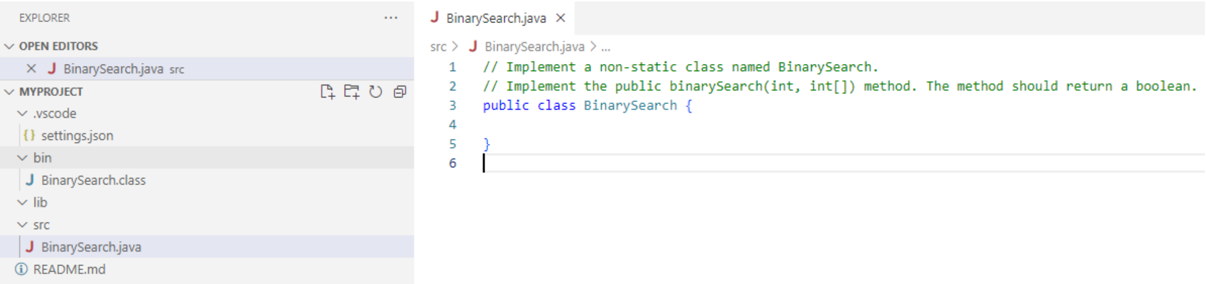
The `App.java` file was removed from the project, and a new Java file was created with the name of the algorithm that was going to be generated. When the Java file was created, an empty class file was automatically added. Figure 4.4 shows the preparation needed for generating a binary search algorithm using Copilot.



```java
public class BinarySearch {
    
}
```

Figure 4.4: Example Java file for binary search generation without user input.

The next step was to add the instruction as a code comment above the class definition. Copilot uses the existing code in the project and the comment to generate suggestions. Figure 4.5 shows how the file looks when the instruction has been added.

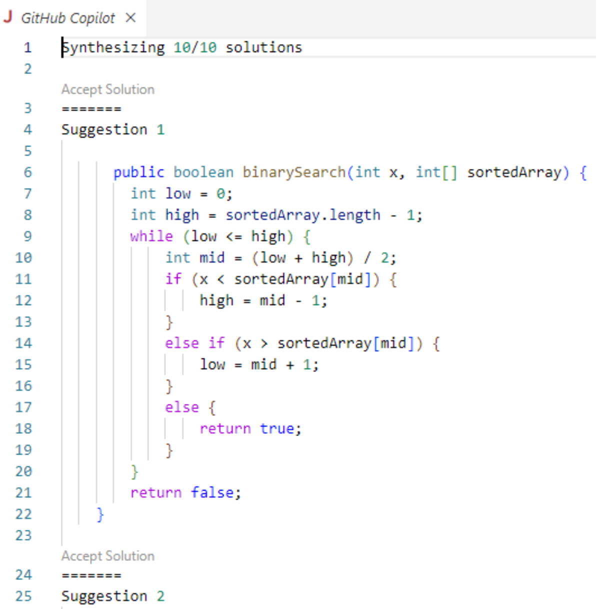


```java
// Implement a non-static class named BinarySearch.
// Implement the public binarySearch(int, int[]) method. The method should return a boolean.
public class BinarySearch {

}

```

Figure 4.5 Example Java file for binary search generation with user input.

There are several ways to get code suggestions from Copilot. The first alternative that we considered was to start adding code inside the class by, for example, writing “public”. A grayed-out code suggestion would then appear which can be accepted by pressing the tab key.

The other alternative we considered, which can be seen in Figure 4.6, was to press the control and enter keys, which opens a new window showing multiple suggestions. This alternative is faster since it suggests up to ten different alternatives simultaneously. As this option also avoided the problem of having to create new projects for every generation to get independent solutions, this was the approach used for code generation with Copilot.



```
Synthesizing 10/10 solutions

=======
Suggestion 1

    public boolean binarySearch(int x, int[] sortedArray) {
        int low = 0;
        int high = sortedArray.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (x < sortedArray[mid]) {
                high = mid - 1;
            }
            else if (x > sortedArray[mid]) {
                low = mid + 1;
            }
            else {
                return true;
            }
        }
        return false;
    }

=======
Suggestion 2
```

Figure 4.6: Copilot suggestions for the Binary Search algorithm.

These suggestions were then stored in the Java project to be tested for code quality and correctness.

4.2 Storing the generated code in the project

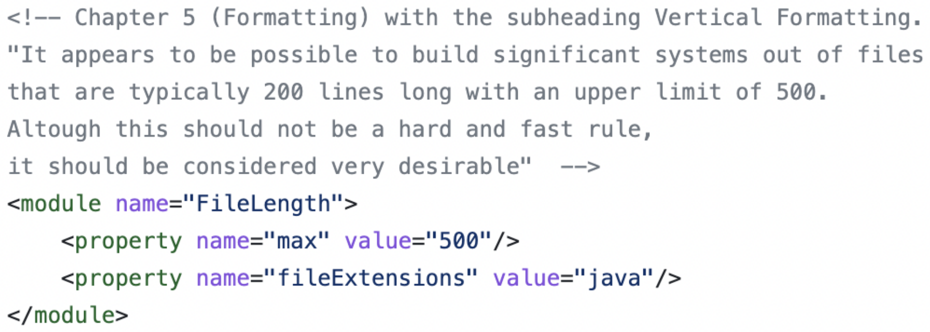
The generated code was stored in the main Java project used when the controlled experiment was performed. The code generated by ChatGPT and Copilot was divided into separate folders. In these folders, the algorithms were divided into separated subfolders. A folder was then created for each generation where the generated algorithm was stored. These folders were named `Tx`, where `x` is the generation number. Scripts to automate this process were implemented and are described in section 4.5.4. In Figure 4.7, the folder structure can be seen for ChatGPT and the binary search algorithm.

```
∨ src
  ∨ main / java
    > BookExamples
    ∨ ChatGPT
      ∨ BinarySearch
        ∨ T1
          J BinarySearch.java
```

Figure 4.7: Folder structure for ChatGPT

4.3 Testing code quality

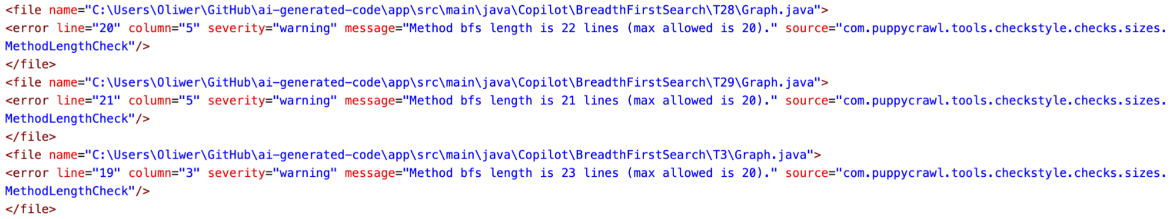
The code quality was tested by implementing principles from the book *Clean Code* by Robert C. Martin in Checkstyle. A Checkstyle configuration file was written in order to implement the selected code quality principles from the book. Figure 4.8 shows a rule implemented in Checkstyle.



```
<!-- Chapter 5 (Formatting) with the subheading Vertical Formatting.
"It appears to be possible to build significant systems out of files
that are typically 200 lines long with an upper limit of 500.
Altough this should not be a hard and fast rule,
it should be considered very desirable"  -->
<module name="FileLength">
    <property name="max" value="500"/>
    <property name="fileExtensions" value="java"/>
</module>
```

Figure 4.8: Module from Checkstyle configuration file used to check vertical formatting.

Checkstyle was executed directly from a Checkstyle JAR file, enabling the tool to be used without automation tools such as Gradle or Maven. A script, described in further detail in section 4.5.1, was created to automate the process of collecting Checkstyle data from each Java file automatically. The Checkstyle data was then written to an XML file, which displays all errors and includes information about the file and the line where each error occurred. Figure 4.9 shows a part of the XML file. Finally, the report was manually imported into Matlab for statistical analysis.

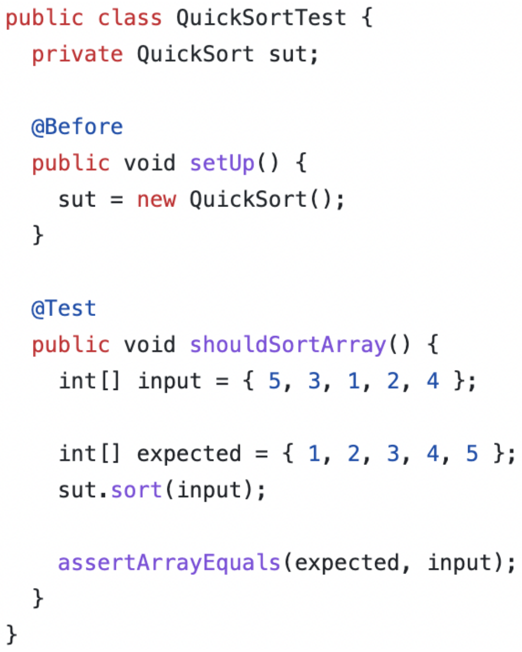


```
<file name="C:\Users\Oliwer\GitHub\ai-generated-code\app\src\main\java\Copilot\BreadthFirstSearch\T28\Graph.java">
<error line="20" column="5" severity="warning" message="Method bfs length is 22 lines (max allowed is 20)." source="com.puppycrawl.tools.checkstyle.checks.sizes.
MethodLengthCheck"/>
</file>
<file name="C:\Users\Oliwer\GitHub\ai-generated-code\app\src\main\java\Copilot\BreadthFirstSearch\T29\Graph.java">
<error line="21" column="5" severity="warning" message="Method bfs length is 21 lines (max allowed is 20)." source="com.puppycrawl.tools.checkstyle.checks.sizes.
MethodLengthCheck"/>
</file>
<file name="C:\Users\Oliwer\GitHub\ai-generated-code\app\src\main\java\Copilot\BreadthFirstSearch\T3\Graph.java">
<error line="19" column="3" severity="warning" message="Method bfs length is 23 lines (max allowed is 20)." source="com.puppycrawl.tools.checkstyle.checks.sizes.
MethodLengthCheck"/>
</file>
```

Figure 4.9: Part of a Checkstyle report

4.4 Testing code correctness

JUnit 4 was used to assess the code correctness. The JUnit tests were based on the algorithm implementations in the GitHub repository from the author of the book *Beginning Java Data Structures and Algorithms* by James Cutajar. Figure 4.10 shows a test class for the Quicksort algorithm.



```java
public class QuickSortTest {
  private QuickSort sut;

  @Before
  public void setUp() {
    sut = new QuickSort();
  }

  @Test
  public void shouldSortArray() {
    int[] input = { 5, 3, 1, 2, 4 };

    int[] expected = { 1, 2, 3, 4, 5 };
    sut.sort(input);

    assertArrayEquals(expected, input);
  }
}
```

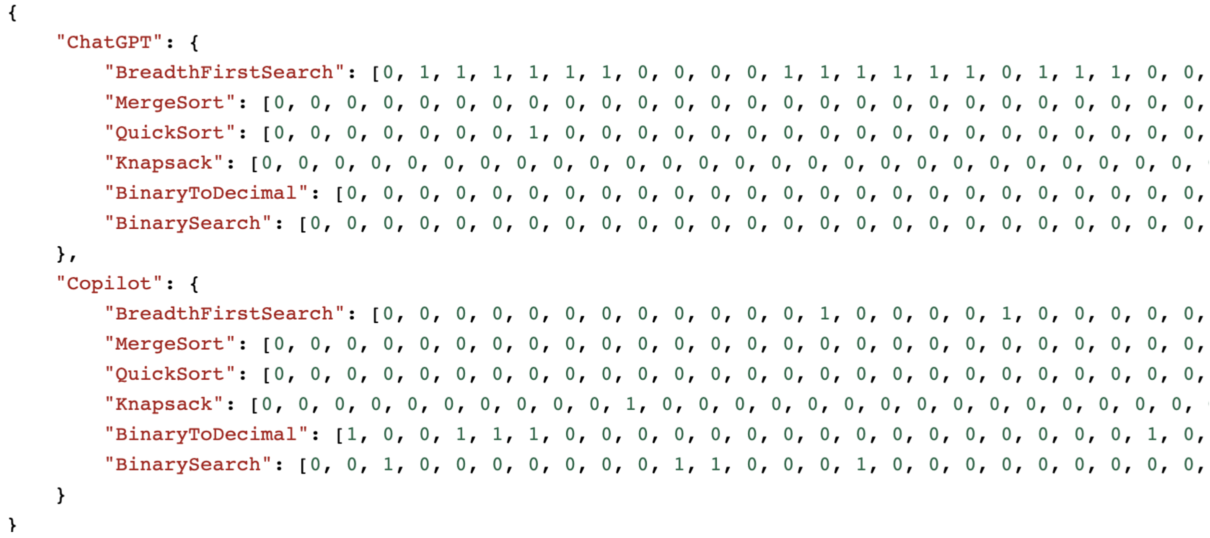
Figure 4.10: JUnit test class for the Quicksort algorithm

Each algorithm had one or multiple test cases. If all the tests passed, the generated implementation was considered to be correct.

Before the testing was conducted, a shell script was used for compiling the code generated by ChatGPT and Copilot and the corresponding unit tests. The purpose of the script was to solve the problem with uncompilable test files when the AI tools generated algorithms with, for example, a different number of method or class arguments than what was requested in the instructions. The script is described in more detail in section 4.5.5.

After the code compilation script had finished, another shell script, described further in section 4.5.2, was used to run the compiled source code with the compiled tests. The script produced four different test result files. A Javascript script was then used to merge the four scripts into one JSON file with the final test results.

The test results were saved in a JSON file where the AI tools were separated, and the test results were stored in arrays for each algorithm. A passed test has the value 1, and a failed test has the value 0. Figure 4.11 shows a part of the JUnit test results.



```
{
    "ChatGPT": {
        "BreadthFirstSearch": [0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0,
        "MergeSort": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        "QuickSort": [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        "Knapsack": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        "BinaryToDecimal": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        "BinarySearch": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    },
    "Copilot": {
        "BreadthFirstSearch": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
        "MergeSort": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        "QuickSort": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        "Knapsack": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        "BinaryToDecimal": [1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
        "BinarySearch": [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    }
}
```

Figure 4.11: Part of the JUnit test results

4.5 Scripts

Different scripts were written in order to automate the process of collecting code quality and correctness data from the generated algorithms. These scripts made it easier to perform tests and checks and to extract the results.

4.5.1 Checkstyle Standalone

The checkstyle standalone script was responsible for creating the Checkstyle report. The Checkstyle report contained the code quality errors that were found in the generated algorithms.

The script was a shell script combined with a NodeJS script. In order to run the script, the `runChecks.sh` script was executed in the root of the project.

When executed, a script named `run.sh` tried to build all Java files in the `Tx` folders that existed in `app/src/main`. If the build succeeded, a `.class` file was created in the `Tx` folder. Figure 4.12 shows the `.class` file.

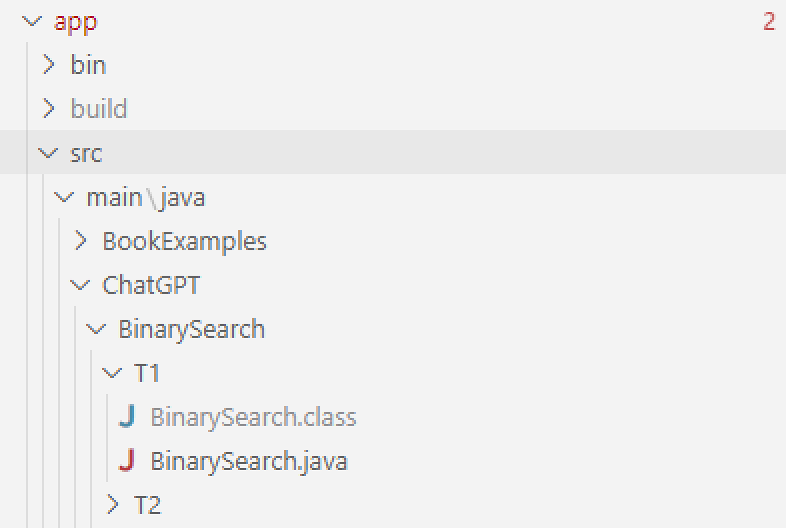
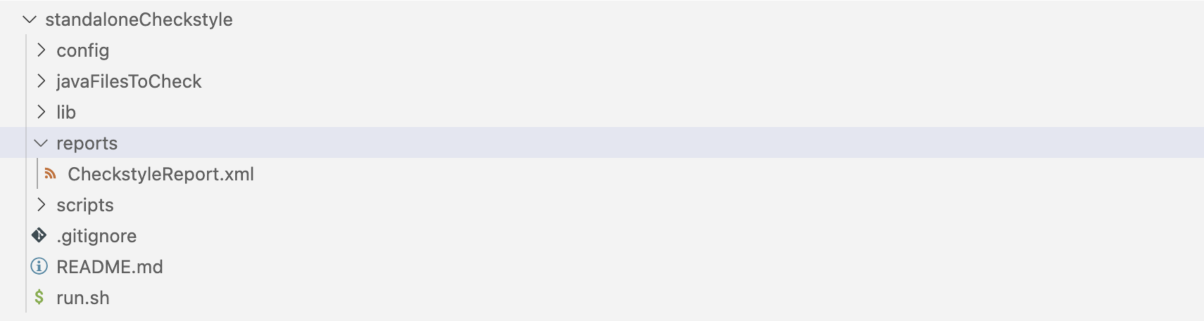


Figure 4.12: Example of successful build in a `Tx` folder where a `.class` file exits

Checkstyle was then executed in the folder where a `.class` file was present. The generated report was added to an already existing XML file. This file contained all the reports that had been generated.

Since the content of multiple Checkstyle reports is merged into a single XML file, it becomes an invalid file. Therefore, the report had to be formatted in order to become a valid XML file. This was solved by a NodeJS script.

`FixCheckstyleReport` was a NodeJS script that took the `CheckstyleReport.xml` file from the reports folder as input and then formatted the XML file in order for it to become valid. The output was then written to `FixedCheckstyleReport.xml` in the reports folder. After the output had been written successfully, the file was renamed to `CheckstyleReport.xml` by the shell script. The resulting file was used in Matlab for statistical analysis. Figure 4.13 shows the folder structure of the `FixCheckstyleReport` script.

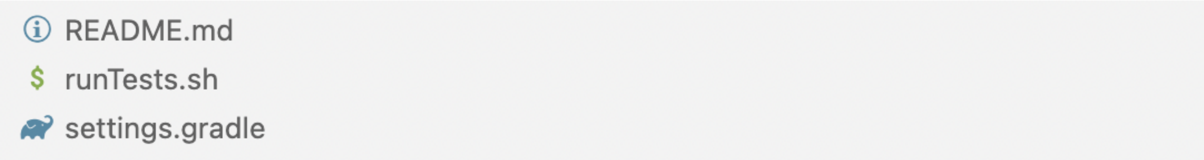


```
standaloneCheckstyle
  config
  javaFilesToCheck
  lib
  reports
    CheckstyleReport.xml
  scripts
  .gitignore
  README.md
  run.sh
```

Figure 4.13: Standalone checkstyle report

4.5.2 Test Script

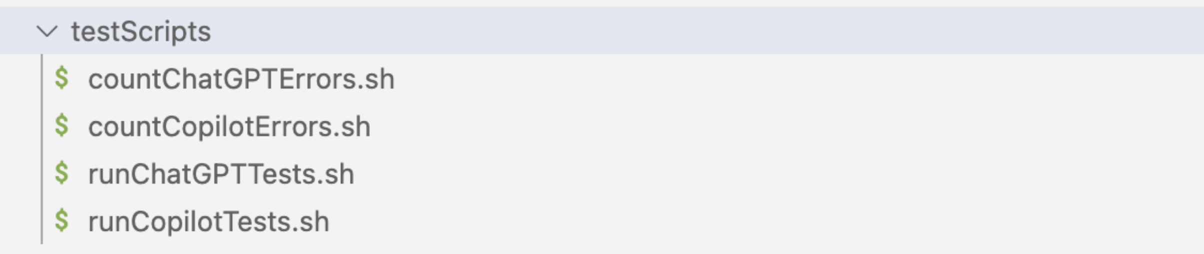
The test script was responsible for testing and creating a `testResults.json` file with all the test results. The `runTests.sh` shell script was used to run the JUnit tests for all the algorithms and tests that had been compiled using another script that is described in section 4.5.5. Figure 4.14 shows the location of the file in the project root.



```
README.md
runTests.sh
settings.gradle
```

Figure 4.14: `runTests.sh` script in the root of the project

The runTests.sh script triggered four other test scripts: `runChatGPTTests.sh` and `runCopilotTests.sh`, which were responsible for running the JUnit tests for ChatGPT and Copilot, and `countChatGPTErrors.sh` and `countCopilotErrors.sh`, which counted all the errors caused by a compilation error. Figure 4.15 shows the test scripts executed by `runTests.sh`.



```
testScripts
  countChatGPTErrors.sh
  countCopilotErrors.sh
  runChatGPTTests.sh
  runCopilotTests.sh
```

Figure 4.15: Test and error scripts

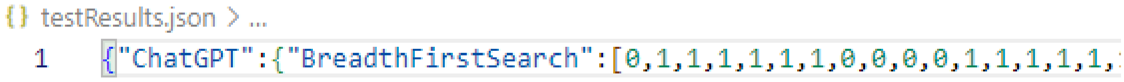
These scripts were needed because JUnit tests cannot be executed if the source code or tests cause a build error. Instead, these errors were counted by subtracting the number of compiled tests from the total amount of tests.

Lastly, the results from the tests and build errors were combined using a Javascript script to merge the four test result files created by the test scripts into one. Figure 4.16 shows the folder structure of this script.

```
results
  {} chatGPTErrors.json
  {} chatGPTResults.json
  {} copilotErrors.json
  {} copilotResults.json
.gitignore
JS index.js
{} package.json
```

Figure 4.16: Merge test results

The tests were combined into one JSON file, and this file was written to the root of the project. Figure 4.17 shows a part of the final test results.

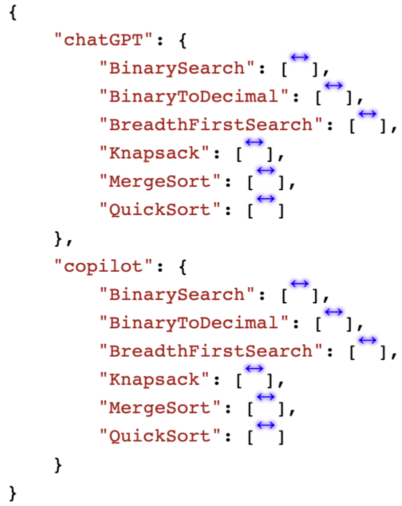


```
{} testResults.json > ...
1    {"ChatGPT":{"BreadthFirstSearch":[0,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,
```

Figure 4.17: Part of the JUnit test results

4.5.3 Line Counter

Line counter is a NodeJS script that counts lines in the generated algorithm files. It automatically creates a JSON file with the number of lines in each generated algorithm. The counted lines are sorted into which algorithm it is and which ai tool that was used to generate the content of the file. The number of lines is stored in an array without any reference to the file that was counted since it is not needed for the analysis.



```
{
    "chatGPT": {
        "BinarySearch": [↔],
        "BinaryToDecimal": [↔],
        "BreadthFirstSearch": [↔],
        "Knapsack": [↔],
        "MergeSort": [↔],
        "QuickSort": [↔]
    },
    "copilot": {
        "BinarySearch": [↔],
        "BinaryToDecimal": [↔],
        "BreadthFirstSearch": [↔],
        "Knapsack": [↔],
        "MergeSort": [↔],
        "QuickSort": [↔]
    }
}
```

Figure 4.18: A shortened version of the line counter output

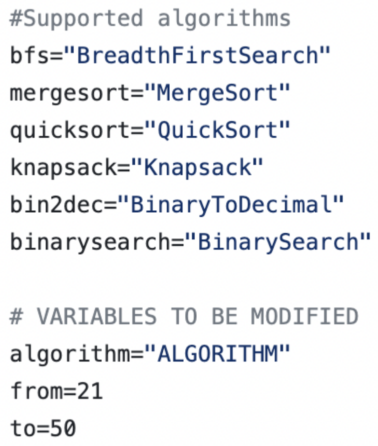
4.5.4 Copilot And ChatGPT Folders Script

`copilotFolders.sh` and `chatGPTFolders.sh` are two scripts used to create the `Tx` folder structure in the source code folder and the test folder.

Depending on which tool was used for the code generation, either the `copilotFolders.sh` or the `chatGPTFolders.sh` script was executed to create the `Tx` folder structure in both the source code folder and the test folder. Figure 4.19 shows the folder structure created by the `chatGPTFolder.sh` script. The functionality of the scripts is the same except for the algorithm defined in each file. The reason two almost identical scripts were created was in order to not accidentally create folders for the wrong AI tool.

Figure 4.19: `Tx` folder structure in the source code folder

The script created `Tx` folders within the chosen range. In Figure 4.20, the given range would create folders from `T21` up to `T50`. The supported algorithms were specified to avoid any mistakes when adjusting the `algorithm` variable.



```
#Supported algorithms
bfs="BreadthFirstSearch"
mergesort="MergeSort"
quicksort="QuickSort"
knapsack="Knapsack"
bin2dec="BinaryToDecimal"
binarysearch="BinarySearch"

# VARIABLES TO BE MODIFIED
algorithm="ALGORITHM"
from=21
to=50
```

Figure 4.20: Algorithms and intervals to create folders from T21 to T50

When the script was executed, a Java file with an empty algorithm class was created in each `Tx` folder in the project source code folder. A test file was also copied from folder `T1` into each `Tx` folder in the project test code folder. The test file in `T1` was added manually when implementing the test cases for the specific algorithm as described in section 4.2.

4.5.5 Compile Code Script

The `compileCode.sh` script was used to compile all the source code files and test files. Each Java class file in main and test was compiled. The source code class files and test class files are then moved to `app/bin/main` or `app/bin/test`, respectively.

In order to use the script, the variables `junit_path` and `hamcrest_path` had to be updated with the correct path for the local JAR files. These had to be downloaded.

```
#!/bin/bash

# Set the path to the JUnit JAR file
# Download the jar files from here: https://github.com/junit-team/junit4/wiki/Download-and-Install
junit_path="/Users/emilia/java/junit-4.13.2.jar"
hamcrest_path="/Users/emilia/java/hamcrest-core-1.3.jar"

# Find all Java files in the project directory and its subdirectories
java_files=$(find app/src/main/java/ChatGPT app/src/main/java/Copilot -name "*.java")
test_files=$(find app/src/test/java/ChatGPT app/src/test/java/Copilot -name "*Test.java")

# Create the bin directory if it doesn't exist
mkdir -p app/bin/main
mkdir -p app/bin/test

for file in $java_files; do
  # Get the name of the file without the extension
  filename=$(basename "$file" .java)

  # Compile the file and move the class file to the bin directory if successful
  if javac -cp .:"$junit_path":"$hamcrest_path" -d app/bin/main "$file"; then
    echo "Compiled $file"
  else
    echo "Failed to compile $file"
  fi
done

for file in $test_files; do
  # Get the name of the file without the extension
  filename=$(basename "$file" .java)

  # Compile the file and move the class file to the bin directory if successful
  if javac -cp .:"$junit_path":"$hamcrest_path":app/bin/main -d app/bin/test "$file"; then
    echo "Compiled $file"
  else
    echo "Failed to compile $file"
  fi
done
```

Figure 4.21: Compile code script

4.6 Code analysis

Two tools were used in order to analyze the data. The following section will briefly describe the use of Matlab and AI-Therapy Statistics for the code analysis.

4.8.1 Matlab

Three scripts were created in Matlab in order to extract the data and visualize it in diagrams: `convertCheckstyleXMLToJson`, `correctnessResults`, and `qualityResults`.

The script `convertCheckstyleXMLToJson` was used to extract necessary data from the XML file and convert the code quality report to JSON format. The JSON file was divided into each AI tool and algorithm. The number of errors in each algorithm is summarized and added into the algorithm array as an integer.

The `correctnessResults` and `qualityResults` scripts were used to merge the data from the provided JSON files, calculate frequencies in data sets, and create all the plots used in Chapter 5 and Chapter 6.

4.8.2 AI-Therapy Statistics

AI-Therapy Statistics is a web-based tool that has been used for data analysis. AI-Therapy Statistics consists of many statistical features for analyzing data.

Descriptive statistics, including mean, mode, median and dispersion, were performed using AI-Therapy Statistics. Normality tests were also conducted, followed by hypothesis testing. The different tests were done for both code quality and code correctness for Copilot and ChatGPT. Appendix 3 provides links for all the tests performed with AI-Therapy Statistics.

5 Results

The following chapter presents the raw results of the controlled experiment. In order to provide both statistical and visual clarity, the data is presented in the form of tables and graphs.

5.1 Code Correctness

This section presents the raw results for the code correctness. For both ChatGPT and Copilot, there were a total of 300 algorithm generations each, with 50 generations per algorithm.

	Number of generations	**Correct generations**	**Incorrect generations**
ChatGPT	300	262	38
Copilot	300	267	33

Table 5.1: Frequency table for the code correctness results for ChatGPT and Copilot

As table 5.1 shows, ChatGPT generated 262 correct algorithms and 38 incorrect algorithms, while Copilot generated 267 correct algorithms and 33 incorrect algorithms. The results are presented graphically in Figure 5.1.

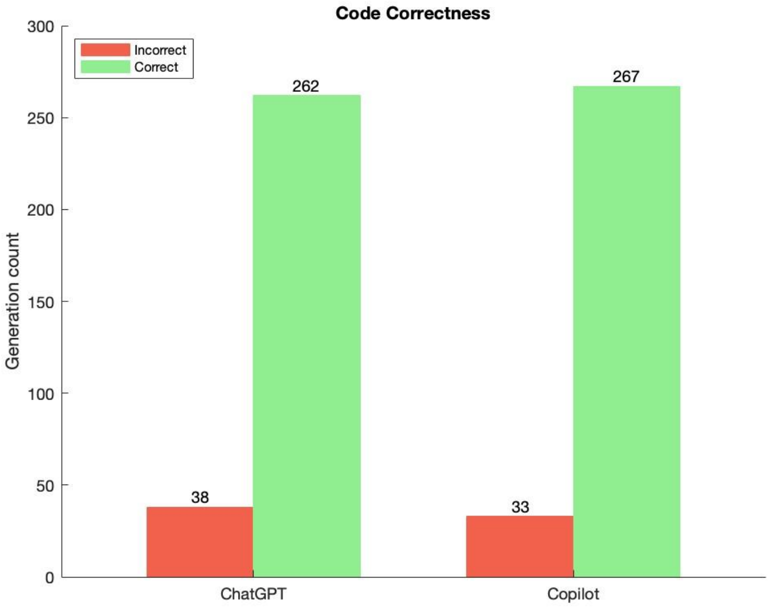


Figure 5.1: Code correctness results

5.2 Code Quality

This section presents the raw results for the code quality. Table 5.2 shows the code generation results of ChatGPT and Copilot for the total number of lines generated.

	Total number of lines	**Lines with errors**	**Lines without errors**
ChatGPT	7895	117	7778
Copilot	8006	475	7531

Table 5.2: Results of quality test on generated algorithms presented as lines

ChatGPT produced 7895 lines of code over 300 algorithm generations, whereas Copilot generated 8006 lines of code for the same number of algorithms.

Of the total lines generated, ChatGPT produced 7778 lines of code without errors and 117 lines with errors. Copilot, on the other hand, generated 7531 lines without errors and 475 lines with errors. The graphical representation of these results is shown in Figure 5.2.

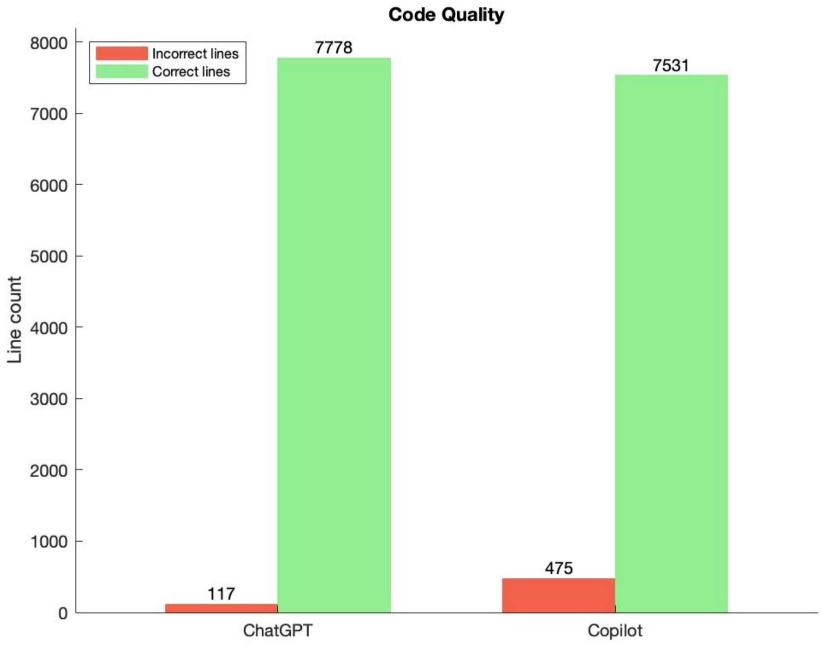


Figure 5.2: Representation of code quality results

6 Analysis

The following chapter presents the analysis of the results from the controlled experiment. In order to provide both statistical and visual clarity, the analysis is presented in the form of tables and graphs. Appendix 3 contains all the statistical analysis results mentioned in this chapter.

6.1 Code Correctness

6.1.1 ChatGPT

Figure 6.1 shows the relative correctness results from code produced by ChatGPT, where both the incorrect and correct generations are shown.

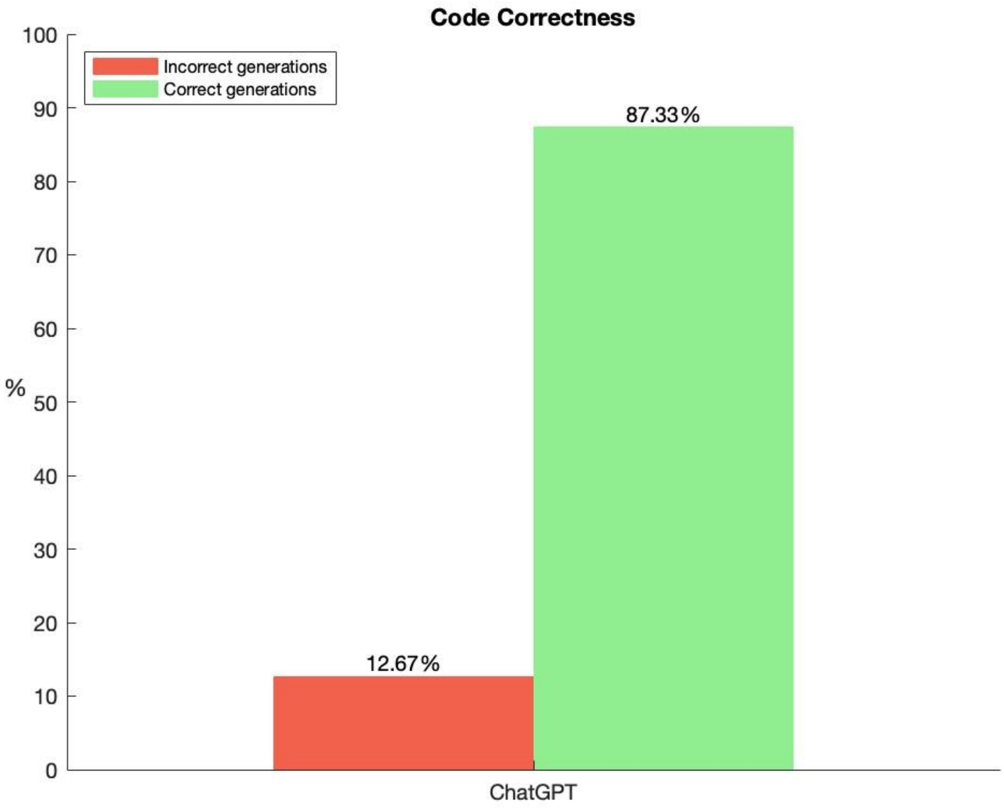


Figure 6.1: Code correctness for ChatGPT

As can be seen in the figure, ChatGPT's code generation was accurate in 87.33% of the cases, with an error rate of 12.67%. The 95% confidence interval suggests that the true mean of incorrect code generation in the population lies between 8.9% and 16.5%.

As such, ChatGPT demonstrated reasonable effectiveness in producing correct code based on the provided instructions, achieving an accuracy of 87.3%. Nonetheless, there is still potential for improvement since incorrect code was generated in 12.7% of the cases.

6.1.2 Copilot

Figure 6.2 shows the relative correctness results from code produced by Copilot.

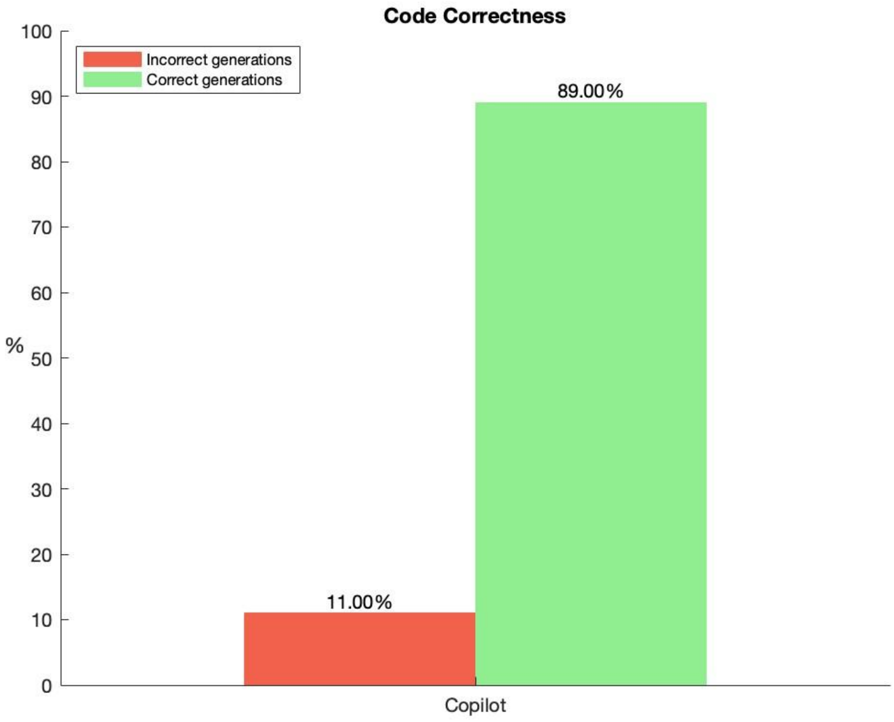


Figure 6.2: Code correctness for Copilot

The analysis revealed that Copilot was able to generate accurate code in 89% of the cases, while producing incorrect code in 11% of the cases. The 95% confidence interval suggests that the true mean of incorrect code generation in the population lies between 7.4% and 14.6%.

The results indicate that Copilot is quite capable of providing correct code from specified instructions. However, like ChatGPT, there is still some potential for improvement as Copilot generated 11% incorrect algorithms.

6.1.3 Hypothesis testing

As the data is binary for both Copilot and ChatGPT, the Mann-Whitney U test was performed in order to determine if there is a statistically significant difference between ChatGPT and Copilot in terms of code correctness. As mentioned in Chapter 2, the Mann-Whitney U test is a non-parametric test and does not assume normal distribution of the data [23]. The results of this test at a significance level of 0.05 are presented in table 6.1.

Mann-Whitney U Test	
U	44250.000
z-score	-0.631
p	0.528
Effect size *r*	-0.026

Table 6.1: Mann-Whitney U test results

Based on a significance level of 0.05, the Mann-Whitney U tests showed no significant difference between ChatGPT and Copilot and the null hypothesis can, therefore, not be rejected.

6.2 Code Quality

6.2.1 ChatGPT

Figure 6.3 shows the relative quality results from code produced by ChatGPT where the percentages of both incorrect and correct lines are displayed.

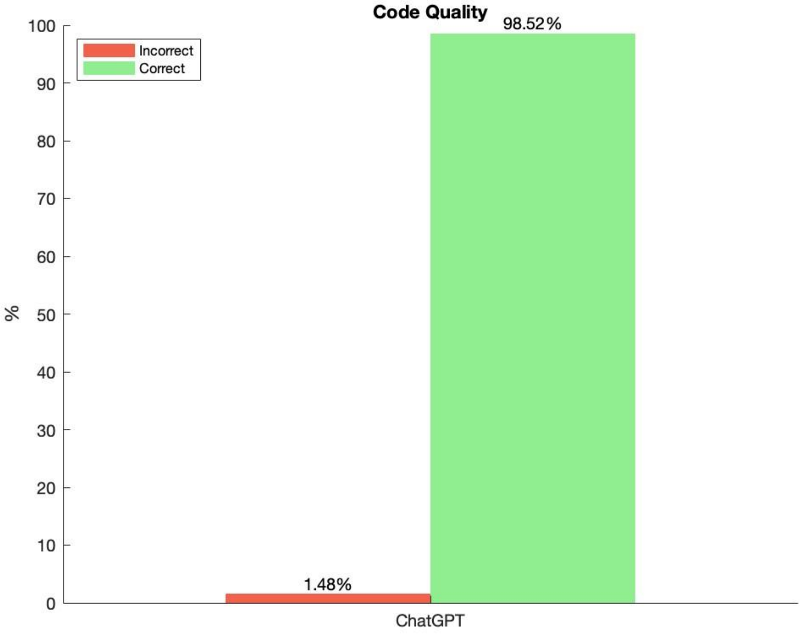


Figure 6.3: Code quality for ChatGPT

As can be seen in Figure 6.3, the results state that ChatGPT was able to generate lines of code of high quality in 98.52% of the cases while breaking the code quality rules in 1.48% of the cases.

Table 6.2 shows the frequency table for the ChatGPT data. The number of quality rule violations represents how many violations occurred in one generation. The frequency represents how many generations contained a specific number of quality rule violations.

No. of quality rule violations	0	1	2	3	4
Frequency	242	19	20	18	1

Table 6.2: Frequency table for ChatGPT

As can be seen in table 6.2, the frequency of code generations with no quality rule violations is 242 out of 300, representing 80.7% of the samples. This means that a majority of the code produced by ChatGPT has no quality rule violations, suggesting high-quality code generation.

Furthermore, the frequency of code generations with 1, 2, or 3 quality rule violations represents a combined total of 19.3% of the samples. While these samples have a small number of quality rule violations, they still indicate relatively good code quality.

The frequency of code generations with 4 quality rule violations represents only 0.33% of the samples. This low frequency indicates that cases with a high number of quality rule violations are rare, further supporting the idea that ChatGPT generally generates high-quality code. Table 6.2 is visualized in Figure 6.4.

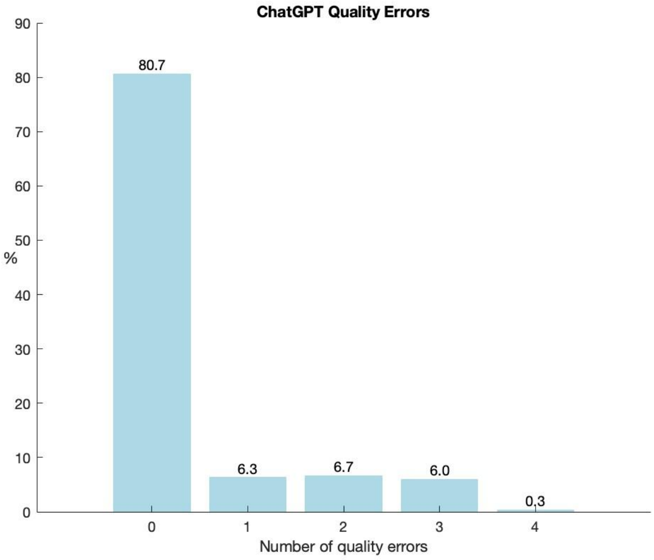


Figure 6.4: Frequency plot for ChatGPT

Table 6.3 shows the descriptive statistics of the ChatGPT code quality results.

Number of samples	300
Mean	0.390
Standard error of the mean	0.051
95% confidence interval for the mean	[0.290, 0.490]
Median	0.00

Mode	0 (count = 242)
Range	4 - 0 = 4
Interquartile range (IQR)	0.000
Sample standard deviation	0.880

Table 6.3: Descriptive statistics for ChatGPT code quality results

The mean number of lines with errors is 0.390, indicating that on average, there are less than half a quality rule violation per generation. The median and mode both suggest that in the majority of cases, no quality rule violations are present in the generated code. The low mean error rate implies that ChatGPT generally generates high-quality code.

The data in Table 6.3 exhibits low variability and suggests that the majority of the generated code has consistent quality with minimal quality rule violations. The 95% confidence interval estimates that the actual mean number of quality rule violations in the generated code for the entire population lies between 0.290 and 0.490 errors per generation. This further supports the idea that ChatGPT generally provides code with a low number of quality rule violations.

Moreover, the sample standard deviation of 0.880 is relatively small compared to the range of the dataset. This indicates that most of the data points are close to the mean, suggesting that the quality of the code generated by ChatGPT is consistent.

In summary, the provided statistical analysis indicates that ChatGPT can indeed provide high-quality code, as demonstrated by the low mean error rate, low variability, and the majority of the code having no quality rule violations. However, there are occasional instances where quality rule violations are present.

6.2.2 Copilot

Figure 6.5 shows the relative quality results from code produced by Copilot, where the percentages of both incorrect and correct lines are displayed.

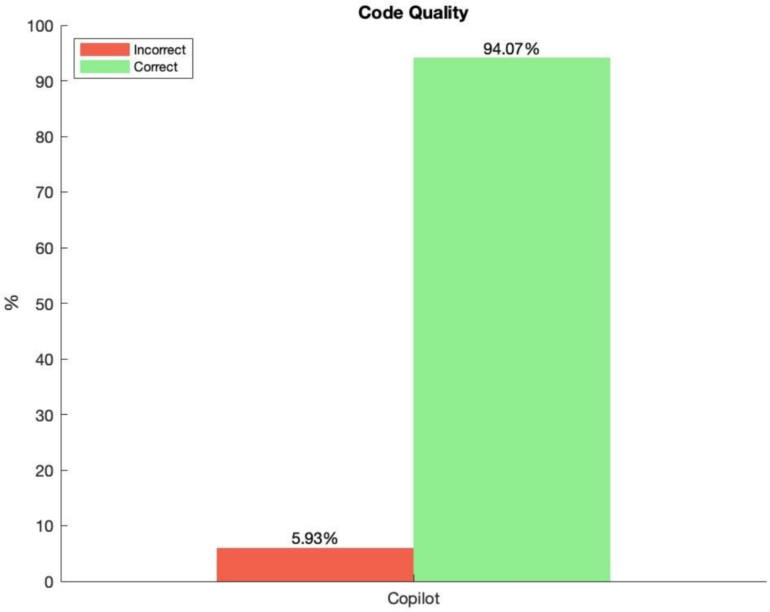


Figure 6.5: Code quality for Copilot

For Copilot, the study results revealed that the tool produced correct lines 94.07% of the time, with 5.93% of the cases resulting in lines of code that break the defined code quality rules.

Table 6.4 shows the frequency table for the Copilot data. Number of quality rule violations represents how many quality rule violations occurred in one generation. The frequency represents how many generations contained a specific number of quality rule violations.

Quality rule violations	0	1	2	3	4	5	6	7	8	9	10	11	12	13	20
Frequency	194	42	8	11	1	5	4	13	3	4	6	5	1	1	2

Table 6.4: Frequency table for Copilot

Out of 300 code generations, 64.67% were produced with 0 quality rule violations by Copilot, indicating its ability to generate high-quality code.

A total of 61 code generations had 1-3 quality rule violations, representing 20.33% of the samples. While the number of quality rule violations in these samples is relatively small, it's worth noting that one-fifth of the generated code may require some level of correction or adjustment to achieve high code quality.

Code generations with 4-9 quality rule violations were relatively infrequent at 8%. Although these samples have a moderate number of quality rule violations, they may require more significant adjustments to correct.

Instances with 10 or more quality rule violations were the least common, at only 7%. This low frequency indicates that cases with a high number of quality rule violations are not very common but should be carefully reviewed and corrected to achieve code of high quality. Figure 6.6 shows the percentage of generations having a specific number of quality rule violations.

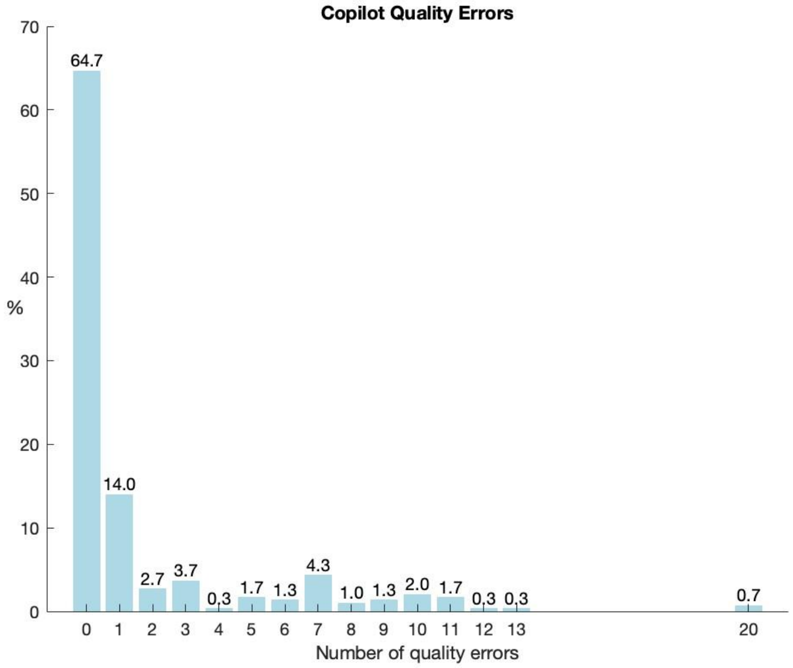


Figure 6.6: Frequency plot for Copilot

Table 6.5 shows the descriptive statistics of the Copilot code quality results.

Number of samples	300
Mean	1.583
Standard error of the mean	0.188
95% confidence interval for the mean	[1.213, 1.954]

Median	0.00
Mode	0 (count = 194)
Range	20 - 0 = 20
Interquartile range (IQR)	1.000
Sample standard deviation	3.260

Table 6.5: Descriptive statistics for Copilot code quality results

Based on the metrics in table 6.5, we can infer that Copilot tends to generate code with a low number of quality rule violations. The median and mode both being 0 suggest that Copilot often produces code without any quality rule violations. However, the sample standard deviation of 3.260 indicates that there is some variation in the number of quality rule violations generated, suggesting that the quality of the code generated by Copilot is not entirely consistent.

On average, there are slightly over one and a half quality rule violations per generation, with a mean number of quality rule violations of 1.583. The 95% confidence interval for the mean is 1.213 to 1.954, which implies a 95% level of confidence that the actual population mean of quality rule violations is contained within this range. This result further supports the notion that Copilot typically produces code with a relatively low to moderate number of quality rule violations.

While the results indicate that Copilot generally generates code with few quality rule violations, there are some instances where the quality rule violation count is relatively high. The presence of these outliers suggests that Copilot may occasionally produce code with a significant number of code quality rule violations.

In conclusion, Copilot can provide high-quality code, as shown by the low mean, median, and mode of the quality rule violation count. However, there is some variability in how many code quality rules were broken, and a small number of samples exhibit a high number of quality rule violations.

6.2.3 Hypothesis testing

Before the hypothesis testing was performed, normality tests were conducted for the resulting code quality data of both ChatGPT and Copilot (the results are presented in section 5.2). of both ChatGPT and Copilot. The results of the normality tests can be found in Appendix 3. Neither the data from ChatGPT nor the data from Copilot was deemed to have a normal distribution.

In order to determine if there is a statistically significant difference between the code quality produced by ChatGPT and Copilot, a non-parametric test, the Mann-Whitney U test, was performed. This test was chosen because it does not assume normal distribution of the data [23]. The results of this test at a significance level of 0.01 are presented in Table 6.6.

Mann-Whitney U Test	
U	37105.500
z-score	-4.741
p	< 0.001
Effect size *r*	-0.194

Table 6.6: Mann-Whitney U Test results

Based on a significance level of 0.01, the Mann-Whitney U test showed a statistically significant difference in code quality between ChatGPT and Copilot. While the effect sizes are relatively small, the results provide strong evidence that the code quality generated by the two tools is significantly different.

7 Discussion

In this study, the major challenge was to find a good way to generate the code from ChatGPT and Copilot to compare them on equal terms and to be able to draw conclusions from the results. As described in further detail in Chapter 2, we ultimately chose to generate the code using code comments for both tools and to only give the AI tools one chance to make a correct generation. Normally, a developer using these tools might ask the AI to fix the code if it's incorrectly generated. However, it's still of interest to examine the ability of the AI to generate correct code on the first try. For future work, it would be interesting to see if the updated models, like ChatGPT-4, are able to generate code of higher accuracy and quality. When the code is generated correctly the first time, it could save time for the developer.

Another challenge was to generate code from Copilot. Copilot sometimes writes just a small part of the code, and the developer has to help the tool out by writing some code or even just typing a space. Another problem with Copilot is that it's affected by previous code generations and existing code in a project, even if the previous generations have been deleted. To generate algorithms independently of the previous generation, we decided to go for the alternative of generating multiple suggestions at the same time. The process is described more accurately in Chapter 4.

When generating code with Copilot, it often gave code suggestions containing "to do" comments, unfinished implementations, or method definitions. This was especially apparent for simpler algorithms. We first decided to include all of the suggestions we got when generating multiple solutions. However, when we had collected the data, we observed that Copilot generated invalid code more often for simpler algorithms. As an example, in the first results we got, there were 20 generations with compilation errors out of 50 for the Binary Search algorithm. As a comparison, only 3 out of 50 generations had compilation errors for the BFS algorithm.

These results could be attributed to the fact that more complex algorithms offer a broader range of potential solutions, thereby affording Copilot greater flexibility to suggest alternative approaches. Conversely, shorter algorithms offer fewer possibilities, resulting in more limited suggestions and possibly requiring solutions with only, for example, a method definition. As such, we started to doubt the validity of the Copilot results and how we collected our data. In their research, B. Yetistiren, I. Ozsoy, and E. Tuzun [7] distinguished between validation and correctness. As our research questions aimed to examine the code quality and correctness and not if the code validates, we ultimately decided not to include the invalid code suggestions generated by Copilot and instead focus on analyzing the correctness of valid code only. The discarded suggestions include method definitions, code with to-do comments, unfinished implementations, and code with syntax errors. With ChatGPT, we didn't get any invalid solutions during code generation.

The analysis of the correctness results for ChatGPT showed that it generated correct code in 87.33% of the cases, and therefore, the answer to *RQ1: How well does ChatGPT provide correct code from specified instructions?* is that ChatGPT is generally good at providing correct code from specified instructions. However, there's still a fair bit of incorrect generations at the complexity level of the chosen algorithms, but in general, ChatGPT seems to be a good tool for generating simpler algorithms.

The analysis of the correctness results for Copilot was similar to that of ChatGPT. Copilot performed marginally better and generated correct code in 89% of the cases. As such, the research question *RQ2: How well does GitHub Copilot provide correct code from specified instructions?* gives the same answer as for ChatGPT, that it, in general, is good at producing correct code from given instructions. This is consistent with the results of previous studies such as B. Yetistiren, I. Ozsoy, and E. Tuzun [7] who found that Copilot generated 28.7% fully correct code for problems in the HumanEval dataset. However, our result is significantly higher, which might be attributed to the improvements made in the model since the previous studies were conducted. It can also be due to differences in the complexity level between the problems in the HumanEval dataset and the algorithms we chose to include in this study. It could also be attributed to the evaluation process being used. If the partially correct code of 51.2% is taken into account, the findings by B. Yetistiren, I. Ozsoy, and E. Tuzun would be 79.9% correct generations compared to the findings in our study of 89%.

The results of the Mann-Whitney U test showed no significant difference between ChatGPT and Copilot in terms of code correctness. Therefore, it can be concluded that both tools are similarly capable of generating correct code from given instructions, and that no differences could be observed, which answers *RQ5: What differences can be observed between ChatGPT and GitHub Copilot in terms of code correctness?*. The result suggests that the choice between the two tools may depend on other factors, such as the specific needs of the user, rather than their ability to generate correct code.

The analysis of the code quality results showed that, as much as 98.52% of the total lines generated by ChatGPT, didn't have any quality errors. The results also showed that a total of 80.7% of the generated algorithms contained 0 code quality errors. This is an impressive result, indicating that not only can ChatGPT provide correct code, but it can also maintain a high standard of quality in the process. The findings show the potential of ChatGPT to be used as an automated code generation tool, not only for generating functionally correct code but also for generating code that aligns with quality best practices. The answer to the research question *RQ3: How good is ChatGPT at generating code of high quality?* is therefore that ChatGPT seems to perform well when it comes to producing code of high quality.

For Copilot, the analysis showed that it generated correct lines for 94.07% of the total generated lines. The result seems to indicate that Copilot can provide code of high quality equally as well as ChatGPT. However, the findings also revealed that Copilot generated code with no code quality errors 64.7% of the time, which means that more than one-third of the generations contained code quality errors. This suggests that while Copilot is generally capable of generating high-quality code, there may be room for improvement in terms of ensuring consistent adherence to code quality standards. This finding aligns with the results of S. Imai [8], who found that while Copilot generated more code than human pair-programming, the quality of the code generated was lower. The answer to the research question *RQ4: How good is GitHub Copilot at generating code of high quality?* is that Copilot more often than not is good at generating code of high quality. However, in 36.3% of the cases, Copilot generated algorithms with code quality errors. To ensure the highest quality code, users should thoroughly review the output provided by Copilot.

The Mann-Whitney U test performed to answer *RQ6. What differences can be observed between ChatGPT and GitHub Copilot in terms of code quality?* showed a statistically significant difference in code quality between ChatGPT and Copilot. The analysis showed that ChatGPT generally produces higher quality code than Copilot. This suggests that while both tools are capable of generating correct code, ChatGPT may be more reliable for generating high-quality code.

8 Conclusions and Future Work

The purpose of this study was to evaluate how ChatGPT and Copilot perform in terms of generating correct code and code of high quality. The tools were both examined individually and compared to each other to determine any differences between them. A controlled experiment was conducted to address the research questions. Data was collected by generating algorithms with ChatGPT and Copilot, and the generated algorithms were then tested for code correctness and code quality. The research questions are as follows:

- **RQ1:** How well does ChatGPT provide correct code from specified instructions?
- **RQ2:** How well does GitHub Copilot provide correct code from specified instructions?
- **RQ3:** How good is ChatGPT at generating code of high quality?
- **RQ4:** How good is GitHub Copilot at generating code of high quality?
- **RQ5:** What differences can be observed between ChatGPT and GitHub Copilot in terms of code correctness?
- **RQ6:** What differences can be observed between ChatGPT and GitHub Copilot in terms of code quality?

The results of this study suggest that both ChatGPT and Copilot are capable of generating functionally correct code from specified instructions, with ChatGPT generating 87.33% of correct algorithms and Copilot generating 89% of correct algorithms (RQ1, RQ2). Copilot appears to have a slight advantage to ChatGPT, but the difference is minor and not statistically significant (RQ5).

Both ChatGPT and Copilot more often than not generate code with no code quality errors with ChatGPT generating 80.7% of algorithms without any quality rule violations, while Copilot generating violation-free algorithms 64.7% of the time. Therefore, both ChatGPT and Copilot seem to perform well in generating code of high quality (RQ3, RQ4). However, as can be seen from the results, ChatGPT seems to perform better than Copilot and the analysis showed a statistically significant difference between the two tools (RQ6). These findings provide useful insights for developers and teams considering the use of AI-powered tools for automated code generation. Further research may be needed to explore these tools.

The code quality results, however, are limited to the rules from the book *Clean Code* by Robert C. Martin that was implementable in Checkstyle. To get a more extensive code quality check, another tool than Checkstyle might have been preferred. Future work could use other tools or approaches when examining code quality to be able to check for more complex quality rules than what could be checked with Checkstyle, or to get results for a broader spectrum of quality rules in the *Clean Code* book to increase generalizability. As we only evaluated the overall result from all the quality rules, it could also be interesting to build upon this research and evaluate the performance of generating high quality code in more detail by looking at how the AI tools performed for each code quality rule.

For both code correctness and quality, the findings are limited by the number of algorithms chosen to be generated. Future work could add to this study by selecting additional algorithms to be generated to get a more general result. Other types of code could also be generated to generalize it even further. One example could be to generate web components or APIs.

Finally, other programming languages could also be studied to assess if the results differ between languages.

As these models and tools are regularly updated, repeating this study for the updated models could be of interest for future researchers to see if there have been any improvements. Only during the course of this thesis project, ChatGPT-4 was released to the public, and a preview of GitHub Copilot X built on GPT-4 was introduced.

References

[1] A. Boulton, “There's an explosion in AI interest but there's lots of life left in real human intelligence | Adam Boulton” Sky News, 10 Feb.2023. Accessed: May 2, 2023. [Online]. Available: https://news.sky.com/story/theres-an-explosion-in-ai-interest-but-theres-lots-of-life-left-in-real-human-intelligence-adam-boulton-12807211.

[2] A. Moka and J. Zinkula, “ChatGPT may be coming for our jobs. Here are the 10 roles that AI is most likely to replace.” Business Insider, April 9, 2023. Accessed: May 2, 2023. [Online]. Available: https://www.businessinsider.com/chatgpt-jobs-at-risk-replacement-artificial-intelligence-ai-labor-trends-2023-02?r=US&IR=T#tech-jobs-coders-computer-programmers-software-engineers-data-analysts-1.

[3] A. Hughes, “ChatGPT Will Replace Programmers Within 10 Years”, GitConnected, Mars 1, 2023. Accessed: May 2, 2023. [Online]. Available: https://levelup.gitconnected.com/chatgpt-will-replace-programmers-within-10-years-91e5b3bd3676.

[4] Stack Overflow Meta, “Temporary Policy: ChatGPT is banned”, *Stack Overflow*, December 5, 2022. Accessed: May 2, 2023. [Online]. Available: https://meta.stackoverflow.com/questions/421831/temporary-policy-chatgpt-is-banned.

[5] Wikipedia contributors, “ChatGPT”, *Wikipedia, The Free Encyclopedia*, May 2, 2023. Accessed: May 2, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=ChatGPT&oldid=1156516050.

[6] Wikipedia contributors, “GitHub Copilot”, *Wikipedia, The Free Encyclopedia*, May 2, 2023. Accessed: May 2, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=GitHub_Copilot&oldid=1154849983.

[7] B. Yetistiren, I. Oszoy and E. Tuzun, “Assessing the Quality of GitHub Copilot’s Code Generation”, *PROMISE 2022: Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 62-71, November 2022, doi: 10.1145/3558489.

[8] S. Imai, “Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study”, *ICSE '22: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 319-321, May 2022, doi: 10.1145/3510454.3522684.

[9] M.Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde de Oliveira Pinto, J. Kaplan, … W. Zaremba, “Evaluating Large Language Models Trained on Code”, July 2021, doi: 10.48550/arXiv.2107.03374.

[10] S. Sengamedu and H. Zhao, “Neural Language Models for Code Quality Identification”, *MaLTeSQuE 2022: Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation*, pp. 5.10, November 2022, doi: 10.1145/3549034.3561175.

[11] H. Krasner, “The Cost of Poor Software Quality in the US: A 2022 Report”, *Consortium for Information and Software Quality*, December 15, 2022. Accessed: 3 June, 2023. [Online]. Available: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf

[12] T. Armerding, “Poor software quality can cost time and money, straightforward solutions are available”, *IDG Communications, Inc*, Mars 12, 2023. Accessed: 3 June, 2023. [Online]. Available: https://www.csoonline.com/article/3690320/poor-software-quality-can-cost-time-and-money-straightforward-solutions-are-available.html

[13] S. Glen, “ChatGPT writes code, but won’t replace developers”, *TechTarget*, December 14, 2022. Accessed: January 30, 2023. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/news/252528379/ChatGPT-writes-code-but-wont-replace-developers.

[14] Coding Freak, “Top 25 Algorithms Every Programmer Should Know”, *Medium*, June 11, 2020. Accessed: Mars 15, 2023. [Online]. Available: https://medium.com/techie-delight/top-25-algorithms-every-programmer-should-know-373246b4881b.

[15] C. Wohlin, M. Höst, B. Regnell, P. Runeson, M. C. Ohlsson, and A. Wesslén, *Experimentation in Software Engineering*. Berlin: Springer Berlin Heidelberg, 2012, pp. 73-152, doi: 10.1007/978-3-642-29044-2.

[16] TrainingByPackt, “BinarySearch.java”, *GitHub*, Mars 7, 2018. Accessed: Mars 23, 2023. [Online]. Available: https://github.com/TrainingByPackt/Data-Structures-and-Algorithms-in-Java/blob/master/src/main/java/com/packt/datastructuresandalg/lesson1/binarysearch/BinarySearch.java.

[17] TrainingByPackt, “Graph.java”, *GitHub*, April 15, 2018. Accessed: Mars 23, 2023. [Online]. Available: https://github.com/TrainingByPackt/Data-Structures-and-Algorithms-in-Java/blob/master/src/main/java/com/packt/datastructuresandalg/lesson6/bfs/Graph.java.

[18] TrainingByPackt, “MergeSort.java”, *GitHub*, Mars 18, 2018. Accessed: Mars 23, 2023. [Online]. Available: https://github.com/TrainingByPackt/Data-Structures-and-Algorithms-in-Java/blob/master/src/main/java/com/packt/datastructuresandalg/lesson2/activity/mergesort/solution/MergeSort.java.

[19] TrainingByPackt, “QuickSort.java”, *GitHub*, Mars 18, 2018. Accessed: Mars 23, 2023. [Online]. Available: https://github.com/TrainingByPackt/Data-Structures-and-Algorithms-in-Java/blob/master/src/main/java/com/packt/datastructuresandalg/lesson2/sorting/QuickSort.java.

[20] TrainingByPackt, “Knapsack.java”, *GitHub*, Mars 10, 2018. Accessed: Mars 23, 2023. [Online]. Available: https://github.com/TrainingByPackt/Data-Structures-and-Algorithms-in-Java/blob/master/src/main/java/com/packt/datastructuresandalg/lesson4/knapsack/Knapsack.java.

[21] TrainingByPackt, “BinaryToDecimal.java”, *GitHub*, Mars 7, 2018. Accessed: Mars 23, 2023. [Online]. Available: https://github.com/TrainingByPackt/Data-Structures-and-Algorithms-in-Java/blob/master/src/main/java/com/packt/datastructuresandalg/lesson1/binarytodecimal/BinaryToDecimal.java.

[22] TrainingByPackt, “Data-Structures-and-Algorithms-in-Java”, *GitHub*, September 18, 2018. Accessed: Mars 23, 2023. Available: https://github.com/TrainingByPackt/Data-Structures-and-Algorithms-in-Java.

[23] Wikipedia contributors, “Mann-Whitney U test”, *Wikipedia, The Free Encyclopedia*, May 2, 2023. Accessed: May 2, 2023. [Online]. Available:

https://en.wikipedia.org/w/index.php?title=Mann%E2%80%93Whitney_U_test&oldid=1156358886.

[24] A. Schroer, “What Is Artificial Intelligence?”, *Built In*. Accessed: April 29, 2023. [Online]. Available: https://builtin.com/artificial-intelligence.

[25] “What is AI?”, *McKinsey & Company*, April 24, 2023. Accessed: April 29, 2023. [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-ai.

[26] “What is generative AI?”, *McKinsey & Company*, January 19, 2023. Accessed: April 29, 2023. [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-generative-ai.

[27] G. D. Rocha, “AI - Design, architecture, and use of Github Copilot”, *Medium*, May 6, 2023. Accessed: June 5, 2023. [Online]. Available: https://medium.com/@genedarocha/ai-design-architecture-and-use-of-github-copilot-21ca85481e0e.

[28] A. Prakash, “What is transformer architecture and how does it power ChatGPT”, *ThoughtSpot*, February 23, 2023. Accessed: June 5, 2023. [Online]. Available: https://www.thoughtspot.com/data-trends/ai/what-is-transformer-architecture-chatgpt.

[29] G. Lawton, “GitHub Copilot vs. ChatGPT: How do they compare?”, *TechTarget*, April 10, 2023. Accessed: April 29, 2023. [Online]. Available: https://www.techtarget.com/searchenterpriseai/tip/GitHub-Copilot-vs-ChatGPT-How-do-they-compare.

[30] B. Dickson, “GitHub Copilot is the first real product based on large language models”, *The Next Web*, July 10, 2022. Accessed: April 29, 2023. Available: https://thenextweb.com/news/github-copilot-large-language-models

[31] R. Khandelwal, “A Basic Understanding of the ChatGPT Model”, *Medium*, December 28, 2022. Accessed: April 29, 2023. [Online]. Available: https://arshren.medium.com/a-basic-understanding-of-the-chatgpt-model-92aba741eea1.

[32] “Unit Testing”, *JavaTpoint*. Accessed: April 30, 2023. [Online]. Available: https://www.javatpoint.com/unit-testing.

[33] “What Is Unit Testing?”, *SmartBear*. Accessed: April 30, 2023. [Online]. Available: https://smartbear.com/learn/automated-testing/what-is-unit-testing/.

[34] “JUnit Tutorial For Beginners – What Is JUnit Testing”, *Software Testing Help*. Accessed: April 30, 2023. [Online]. Available: https://www.softwaretestinghelp.com/junit-tutorial.

[35] Indeed Editorial Team, “Code Quality: What It Is and How To Measure It (With Tips)”, *Indeed*, February 3, 2023. Accessed: April 30, 2023. [Online]. Available: https://www.indeed.com/career-advice/career-development/what-is-code-quality.

[36] R. Bellairs, “What is Code Quality? Overview + How to Improve Code Quality”, *Perforce Software*, July 11, 2019. Accessed: April 30, 2023. [Online]. Available: https://www.perforce.com/blog/sca/what-code-quality-overview-how-improve-code-quality.

[37] “Coding Standards For Quality and Compliance”, *Perforce Software*. Accessed: April 30, 2023. [Online]. Available: https://www.perforce.com/resources/qac/coding-standards.

[38] ”Overview”, *Checkstyle*. Accessed: April 30, 2023. [Online]. Available: https://checkstyle.sourceforge.io/.

[39] “Standard Checks”, *Checkstyle*. Accessed: April 30, 2023. [Online]. Available: https://checkstyle.sourceforge.io/checks.html.

[40] “Writing Checks”, *Checkstyle*. Accessed: April 30, 2023. [Online]. Available: https://checkstyle.sourceforge.io/writingchecks.html.

[41] Baeldung, “Introduction to CheckStyle”, *Baeldung*, June 4, 2022. Accessed: April 30, 2023. [Online]. Available: https://www.baeldung.com/checkstyle-java.

[42] CetinOgut, “What does Clean Code mean?”, *Medium*, July 5, 2021. Accessed: April 30, 2023. [Online]. Available: https://cogut.medium.com/what-does-clean-code-mean-2190e4aed818.

[43] LLM Editor, “Top 5 Principles of Clean Code: Including Readability, Maintainability, and Simplicity”, May 16, 2023. Accessed: May 22, 2023. [Online]. Available:

https://londonlifestylemag.co.uk/top-principles-of-clean-code-including-readability-maintainability-and-simplicity/.

[44] B. Cook, “The Complete Guide to Readable Code: 11 Principles”, December 9, 2022. Accessed: May 22, 2023. [Online]. Available: https://fellow.app/blog/engineering/the-complete-guide-to-readable-code/.

[45] L. Haimovitch, “Understandability: The Most Important Metric You’re Not Tracking”, June 6, 2020. Accessed: May 22, 2023. [Online]. Available: https://www.infoq.com/articles/understandability-metric-not-tracking/.

[46] T. Davis, “What is "maintainable code"?”, April 14, 2009. Accessed: May 22, 2023. [Online]. Available: https://www.red-gate.com/simple-talk/blogs/what-is-maintainable-code/.

[47] R. C. Martin, “Clean Code” in Clean Code: A Handbook of Agile Software Craftsmanship. USA: Pearson Education, Inc., 2008, pp. 6-7.

[48] R. C. Martin, “Functions” in Clean Code: A Handbook of Agile Software Craftsmanship. USA: Pearson Education, Inc., 2008, pp. 34-35.

[49] R. C. Martin, “Functions” in Clean Code: A Handbook of Agile Software Craftsmanship. USA: Pearson Education, Inc., 2008, pp. 40.

[50] R. C. Martin, “Formatting” in Clean Code: A Handbook of Agile Software Craftsmanship. USA: Pearson Education, Inc., 2008, pp. 76-77.

[51] R. C. Martin, “Formatting” in Clean Code: A Handbook of Agile Software Craftsmanship. USA: Pearson Education, Inc., 2008, pp. 85.

[52] B. R. P. Putra, “Clean Code is slow, but you still need it anyway…”, *Medium*, March 12, 2023. Accessed: June 6, 2023. [Online]. Available: https://medium.com/@bawenang/clean-code-is-slow-but-you-still-need-it-anyway-ffcac6973c93.

A Appendix 1

This appendix presents the code generation instructions used in the study to generate algorithms with Copilot and ChatGPT. The same instructions were used at every generation of an algorithm, and the same set of instructions was used for both of the AI tools.

Binary Search Algorithm Instructions

```
// Implement a non-static class named BinarySearch.
// Implement the public binarySearch(int, int[]) method. The method should return a
boolean.
```

Breadth First Search (BFS) Algorithm Instructions

```
// Implement a non-static breadth search first class named Graph(int vertices).
// Implement the public bfs(int sourceVertex) method. The method should return an
integer array of parent nodes for each vertex in the graph. The array of parent
node values are all initialized to -1 in the bfs(int sourceVertex) method.
// Implement the public addEdge(int, int). The method should add an edge between
two vertices.
```

Merge Sort Algorithm Instructions

```
// Implement a non-static merge sort algorithm class named MergeSort with a public
void mergeSort(int[]) method.
```

Quicksort Algorithm Instructions

```
// Implement a non-static quick sort algorithm class named QuickSort with a public
void sort(int[]) method.
```

Knapsack Algorithm Instructions

```
// Implement a non-static 0-1 knapsack class named Knapsack.
// Implement the public int bottomUp(int capacity, int[] weights, int[] values)
method. The method should return an int with the maximum value.
```

Binary To Decimal Instructions

```
// Implement a non-static BinaryToDecimal class.
// Implement the public convertToDecimal(String binary) method. The method should
return an int with the decimal.
```

B Appendix 2

This appendix presents the code quality rules from the book *Clean Code* by Robert C. Martin, which were used in the study to evaluate the code quality of the generated algorithms. The table included in this appendix showcases the code quality rules along with corresponding quotes from the book, providing a direct reference to the source material (including page numbers).

Code Quality Rule	**Quote From the Book and Checkstyle docs**	**Reference To the Quote**
The files should not be over 500 lines long.	“It appears to be possible to build significant systems out of files that are typically 200 lines long with an upper limit of 500. Although this should not be a hard and fast rule, it should be considered very desirable.”	R. C. Martin, “Formatting” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 77.
A line should not be more than 120 characters long.	“This suggests that we should strive to keep our lines short. The old Hollerith limit of 80 is a bit arbitrary, and I'm not opposed to lines edging out to 100 or even 120. But beyond that is probably just careless.”	R. C. Martin, “Formatting” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 85.
Magic numbers should be hidden behind constants.	“In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.”	R. C. Martin, “Smells and Heuristics” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 300.
Functions should not be more than 20 lines long.	“Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.”	R. C. Martin, “Functions” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 34.
Functions should not have more than three arguments.	“More than three (polyadic) requires very special justification - and then shouldn't be used anyway.”	R. C. Martin, “Functions” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 40.

There should not be nested loops of a depth of more than one level.	“This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two.”	R. C. Martin, “Functions” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 35.
There should not be more than one statement per line.	“Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.”	R. C. Martin, “Good Comments” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 55.
	“It's very difficult to read multiple statements on one line.”	Checkstyle, “Standard Checks”, v10.11.0, May 2023. Accessed: May 30, 2023. [Online]. Available: https://checkstyle.org/config_coding.html#OneStatementPerLine
There should not be any inner assignments.	“Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments. Rather than spend your time writing the comments that explain the mess you’ve made, spend it cleaning that mess.”	R. C. Martin, “Good Comments” in *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Pearson Education, Inc., 2008, pp. 55.
	“With inner assignments, it is difficult to see all places where a variable is set.”	Checkstyle, “Standard Checks”, v10.11.0, May 2023. Accessed: May 30, 2023. [Online]. Available: https://checkstyle.org/config_coding.html#InnerAssignment

C Appendix 3

This appendix presents all the links to the results from the statistical analysis performed using AI-Therapy Statistics.

CODE CORRECTNESS

	ChatGPT data	**Copilot data**
Mean, mode & median	https://www.ai-therapy.com/psychology-statistics/results/20230416225520058	https://www.ai-therapy.com/psychology-statistics/results/20230416225106252
Dispersion	https://www.ai-therapy.com/psychology-statistics/results/20230416225610040	https://www.ai-therapy.com/psychology-statistics/results/20230416225215784
Normality test	https://www.ai-therapy.com/psychology-statistics/results/20230416225700927	https://www.ai-therapy.com/psychology-statistics/results/20230416225342508
Hypothesis testing	https://www.ai-therapy.com/psychology-statistics/results/20230416225902924	

CODE QUALITY

	ChatGPT data	**Copilot data**
Mean, mode & median	https://www.ai-therapy.com/psychology-statistics/results/20230416224317426	https://www.ai-therapy.com/psychology-statistics/results/20230416215022910
Dispersion	https://www.ai-therapy.com/psychology-statistics/results/20230416224413875	https://www.ai-therapy.com/psychology-statistics/results/20230416215316314
Normality test	https://www.ai-therapy.com/psychology-statistics/results/20230416224645602	https://www.ai-therapy.com/psychology-statistics/results/20230416215521826
Hypothesis testing	https://www.ai-therapy.com/psychology-statistics/results/20230604163954461	

D Appendix 4

This appendix presents the links to the code base of this project and instructions on how to repeat the experiment. The main project repository was used for testing the generated algorithms in terms of code correctness and code quality, while the statistical analysis repository was used for converting the code quality results from XML to JSON and for creating graphs to present the results. A link to the hands-on instructions for repeating the project is also provided.

Main project repository	`https://github.com/emiliaajax/ai-generated-code`
Statistical analysis repository	`https://github.com/emiliaajax/ai-tools-analysis`
Hands-on instructions for repeating the controlled experiment	`https://github.com/emiliaajax/ai-generated-code/blob/main/docs/instructions.md`
Test documentation	`https://github.com/emiliaajax/ai-generated-code/blob/main/docs/test-specification.md`