



^b
**UNIVERSITÄT
BERN**

Chat Bots in Software Engineering

A Case Study using ChatGPT

Bachelor's Thesis

Filip Nikolic
December 2023

Supervisor:
Prof. Dr. Timo Kehrer

Software Engineering Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Chatbots have been rising in popularity in the recent months. ChatGPT has been mentioned a lot when talking about these AI being able to "write" the code for you. This sparked the interest in whether and how one could use them to develop various software projects. This thesis focuses on a specific case study where chatGPT is used to try to independently generate code from input prompts so that, in the end, it results in a working application. This application represents the game of sudoku. In the end, there would be two different results, each resulting from a different approach to writing the prompts. The first version led to a working and running code, but the original requirements for a working sudoku game still needed to be achieved. The second version, in contrast, had a fully working application with all desired requirements completed. However, during the process of the second version, various problems occurred, which hindered the development. Ultimately, the conclusion highlights that chatGPT has potential and limitations when developing software. The reason is that even though the end product could be realized without knowing much of the logic behind the code generated, as soon as bugs or missing implementations occurred, chatGPT got lost. The user had to solve these problems by himself.

Contents

Abstract	1
1 Introduction	3
2 Methodology	4
2.1 Why ChatGPT?	4
2.2 Sudoku as the Case Study	4
2.3 Two Different Versions of the Case Study	5
2.4 Version of ChatGPT and Programming Language used	5
3 Execution	6
3.1 Sudoku Version 1	6
3.1.1 Prompts	6
3.2 Sudoku Version 2	8
3.2.1 Prompts	9
4 Results	12
4.1 Insight to the General Scale of the Applications	12
4.2 Results from Version 1	14
4.3 Results from Version 2	15
5 Evaluation	16
5.1 Quantitative Measurements of the different Prompts	16
5.1.1 Version 1	17
5.1.2 Version 2	17
5.2 Code Quality	19
5.2.1 Chidamber and Kemerer	19
5.2.2 Evaluation of the CK-Metrics	21
5.2.3 Additional Metrics and Evaluation on the Code Quality	23
5.3 Problems that frequently occurred	23
5.3.1 End of the Response in the Middle of Code Generation	24
5.3.2 Missing Code Implementations	25
5.3.3 Bug Fixing	27
5.4 Testing	27
6 Discussion	29
6.1 Summary of Key Findings	29
6.2 Implications	30
6.3 Threats to Validity	30
6.3.1 Internal Validity	30
6.3.2 External Validity	30
7 Conclusion and Future Work	31

1 Introduction

In recent months, AI chatbots have gained more and more traction and become a hot topic, especially in social media. ChatGPT has had the biggest impression and impact among the AI chatbots circulating in the last few months. Since the launch of chatGPT in November 2022, which introduced a language model that acts conversationally by making dialogue with the bot possible and answering follow-up questions [2], people have become more aware of using AI to help them in various ways. Be it debugging code, generating poems or routine plans for a workout and in some cases even letting the tool do the homework for the user, people were amazed at the possibilities that the tools provide and how it could affect their daily lives [5]. As briefly mentioned, these chatbots are sometimes used to generate and debug code snippets. So, for example, one would copy and paste a faulty code snippet as input and ask the tool to fix it. The fascinating thing would be that the chatbot allows for follow-up questions, so it feels like the user converses with another person about fixing their code.

The question is whether this tool can efficiently develop working software. One could ask the chatbot to generate code for specific requirements, and by putting together these code snippets, they should get fully functional software [4]. This paper will focus on a particular case study to evaluate how well one of these chatbots can help in developing working software by letting it generate code for a complete application all by itself, with the user having to put in minimal effort to have the application running in the end, for example by only telling what should be generated and not writing any code themselves and in the worst case just having to put two and two together so that in the end the code can adequately compile.

In the following sections, the methodology and execution of the case study will be explained in more detail. Then, the case study results will be evaluated and discussed so that, in the end, there is a conclusion as to whether these AI chatbots can be used to develop software efficiently.

2 Methodology

As mentioned before in the introduction, we are going to focus on a specific example, in other words, a case study, in which we are going to evaluate the use of the AI, in this case, chatGPT, to see how the AI will handle the specific task that we are about to distribute to it. The job we want to accomplish here is to let chatGPT generate the source code for a particular application without us having to do anything more than prompt the AI what it should do and then copy and paste the generated code and try to run it.

The essential aspects that we want to evaluate and observe with this task are how well the source code that chatGPT generated will perform and how much effort has to be put in to run the application entirely. Since there is limited time, this paper will focus on only one specific use case as an application.

The use case will be the popular puzzle game sudoku, elaborated in more detail in a later subsection. There will also be two different versions of the case study, thus resulting in having two different ways of prompting, end products and evaluations.

2.1 Why ChatGPT?

ChatGPT was chosen for this case study mainly because of its popularity and advanced popularity. There are many cases of people using chatGPT to fix bugs in their code or ask for code generation. The owner of chatGPT themselves, OpenAI, provides an example of debugging a piece of code as to how to use the tool. [2].

Thus, the author of this paper decided to use chatGPT as the tool to fill the task of the case study since it is one of the most used tools regarding this task.

2.2 Sudoku as the Case Study

With the same reasoning of only one case study for this paper, the time constraints suggest the application be simple enough so the project is feasible promptly. The goal is to evaluate how the "average" developer can efficiently use the tool to generate a simple application with minimal effort, so the application should not be too complex. Even then, it also should be more demanding of a task like, for example, making a tic-tac-toe application, which can be done in one prompt with a few lines. So, the author decides on a good middle ground by using sudoku as the application that will be implemented.

Sudoku is a puzzle game that is relatively simple to understand and has few rules to follow, but it is still not easy to implement. The tool would need to implement algorithms that create a complete puzzle with rows and columns in which each field has numbers ranging from one to nine, and the numbers in

the respective columns and rows are unique. Furthermore, it needs to have a solving algorithm that checks if the player's input is correct or not and if the conditions of the rows and columns are still full-filled.

Considering the above, sudoku would be a perfect first use case to test how well the tool will generate the source code to make a fully working and running sudoku game.

2.3 Two Different Versions of the Case Study

As already mentioned above, the case study will include two different versions. Both versions have the same task and application they need to implement. These versions differ because of the prompts used as inputs to generate the required code. For that, there are two versions with different prompts, which leads to different results based on these prompts that can be evaluated in correlation to the corresponding version. How the versions and their prompts differ, and the thoughts behind them will be explained in more detail in the third section about the execution of the case study.

2.4 Version of ChatGPT and Programming Language used

ChatGPT has, at the current time in which the paper was written, two different versions, GPT-3.5 and GPT-4, the latter being only accessible behind a pay-wall. With this in mind, the author chose to use the GPT-3.5 version since it is available for everyone and the version that would be used the most, thus being a more accurate display of the current status of the tool.

Java is the programming language the tool will use to generate the code and complete the task. The reasons are that Java is a well-established language and that the author is the most familiar with this language, hence leading to better judgement of the generated code and easier debugging to make the code compile and run in the end.

3 Execution

This section focuses in detail on how the whole case study was executed. The following subsections will explain the thought process behind the prompts of the two versions and what modifications had to be made so the generated code would be compiled and run.

3.1 Sudoku Version 1

This is the first attempt to generate the code for a working sudoku game. The focus was to make the prompts as simple and short as possible to see what chat-GPT would generate and how useful it would be. The latter will be discussed in the evaluation section down the line. It is also important to note that for the first version, the author wanted to give little to no help to the AI to compare it better to the second version. So if there were unfinished code snippets or functionality that was not working and even code that was not used, as long as the code was compiled, these problems were ignored. One can look through the source code on Git Hub and spot these instances.

3.1.1 Prompts

The first-ever prompt right at the start of the conversation in version one can be seen in Figure 1 below.

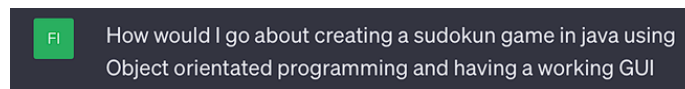


Figure 1: First ever prompt that was entered in chatGPT.

The initial prompt includes a simple phrase asking the tool how the user would create a sudoku game using object-orientated programming in Java.

This prompt results in the tool generating a step-by-step list of how the initial task can be accomplished, as seen in Figure 2 below.

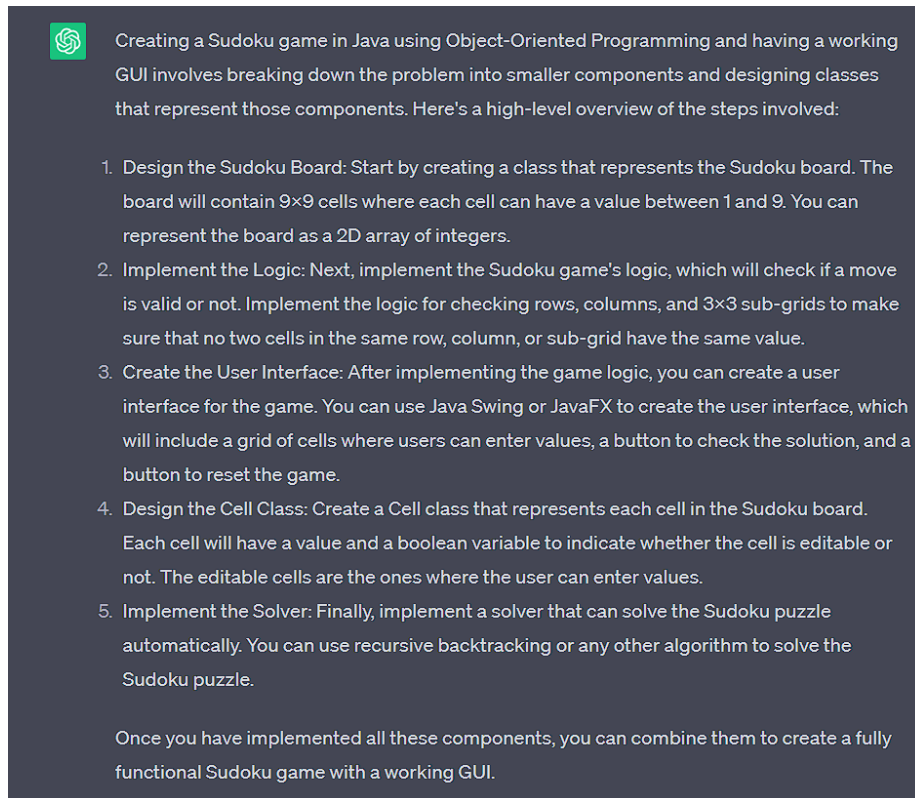


Figure 2: The result of the initial prompt that chatGPT generated, being a to-do list on how to implement a sudoku game in java using object orientated programming.

The next step is to get the tool to generate the code so the user does not have to do it themselves. This is accomplished by explicitly following up on the previous result and asking the AI to generate the code for each step one at a time, as shown in Figures 3 and 4.



Figure 3: The figure shows the prompt used to generate the code snippet for the specific step in the to-do list from the result of the initial prompt.



Figure 4: The generated code for step 1 in the to-do list and an additional explanation of what the code does.

This process would be repeated for the rest of the steps in the to-do list and then finished by asking the AI to combine these snippets into a working application. The complete log of the prompts and the results can be viewed on [Git Hub](#).

Now that all the generated code is present, it is left to take all the generated code and string it together so that the code compiles and runs in the end.

3.2 Sudoku Version 2

The main difference between the second and the first versions lies in the initial prompt. This time, the prompt would be more specific and list more detailed requirements for what should be done but still relatively simple.

3.2.1 Prompts

Figure 5 below displays the second version's initial prompt to start the conversation.

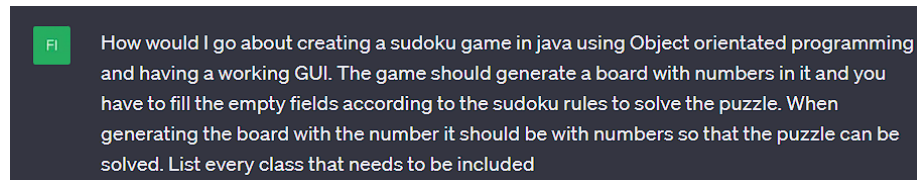


Figure 5: The initial prompt of the second version. This time, a more detailed start with clearer requirements for what the application should do.

One notices that the first sentence is essentially the same as in the prompt of version one. However now, there are additional requirements after the first sentence to state what the tool should generate. The last sentence also says that the AI should list every class needed to implement the application. This was done with the thought of achieving more cohesion. What is meant by that will be more apparent shortly.

Figure 6 shows the results of the specific prompt. As asked, it generated all the classes needed to be able to build the sudoku game and like in version one, it made a step-by-step to-do list of what needed to be done in each class.

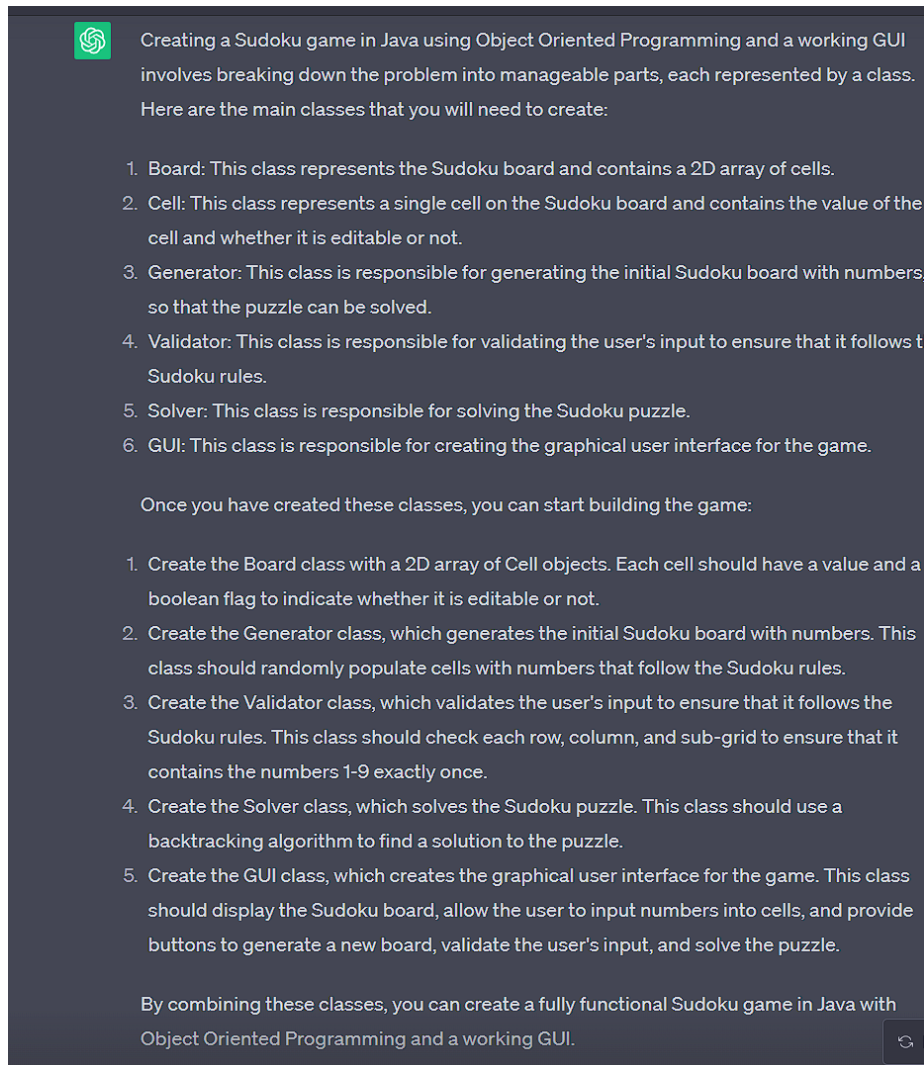


Figure 6: The result of the initial prompt in version two, displaying a step by step to-do list and the classes needed to implement the sudoku game.

The rest of the prompts are similar to version one with the difference that this time, there is an additional statement in the input that points out to the chatbot that it should generate the code with respect to the previous code, as shown in Figures 7 and 8.

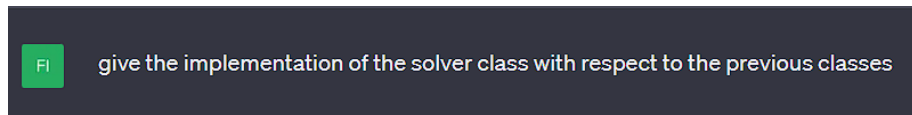


Figure 7: Similar prompt to the ones from the first version but adds the additional statement to respect the previous classes.

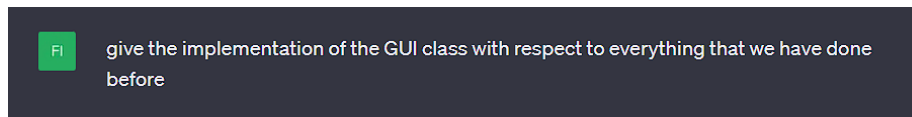


Figure 8: Another example of the extra statement.

Like in version one, this is continued until all the code for the classes has been generated. Then, in the last statement, the tool is asked to create all the code at once and combine the classes so the code can run and compile with minimal modifications from the user. Equal to version one, the complete log of the conversation can be found on the [Git Hub](#).

4 Results

This section briefly presents the results of each version by establishing some software metrics from the generated source code and then showing the functional outcome of both applications. For interested readers who want to try the applications on their local machine, the complete source code will be provided on the Git Hub by following the reference to the specific link below. Both versions will be provided for completion.

4.1 Insight to the General Scale of the Applications

To first get a better understanding of the size and scope of both versions, the source code was analyzed with multiple tools to count various software metrics, which were then summarized in Excel and shown in the following figures.

Version1					
class	NOA	NOM	LOC	CS	
SudokuGame.Cellhandler	0	1	6	1	
SudokuGame	15	3	102	18	
SudokuBoard	2	9	99	11	
SudokuSolver	0	2	54	2	
SudokuGUI	4	1	57	5	
SudokuValidator	0	1	28	1	
Total	21	17	346	38	
Average	3.5	2.833333	57.66667	6.333333	

Figure 9: Various software metrics after analyzing the first version of the application. The columns of the table represent from left to right in order: The class in question, the number of attributes in the class, the number of methods in the class, the number of lines of code and the total class size by combining the number of methods and attributes of the class. Each column also displays the total and average count of the given metric.

Version2					
class	NOA	NOM	LOC	CS	
Cell	4	10	49	14	
Generator	4	6	84	10	
Board	2	13	132	15	
Solver	1	4	92	5	
GUI	6	4	197	10	
Validator	1	5	67	6	
Total	18	42	621	60	
Average	3	7	103.5	10	

Figure 10: Various software metrics after analyzing the second version of the application. The table reads the same as in figure 9

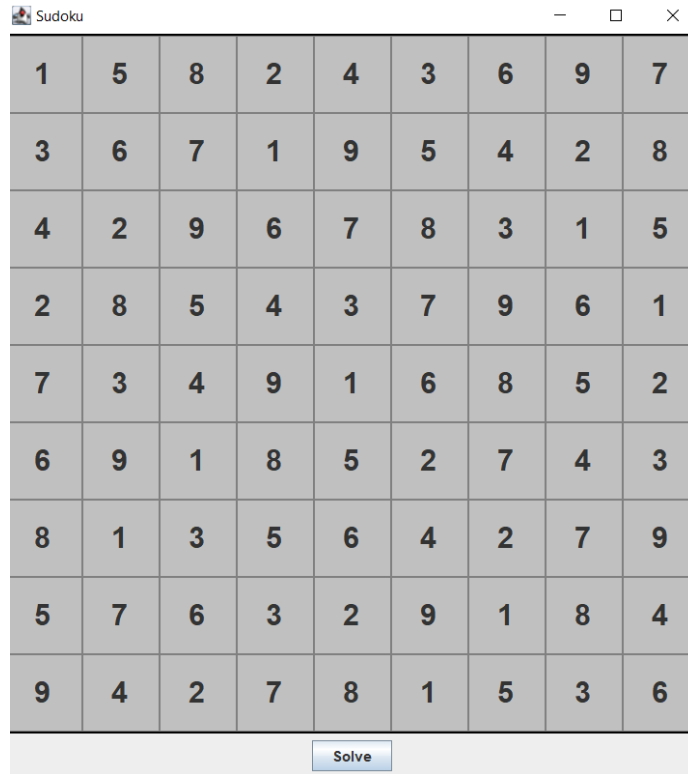
Figure 9 shows the result after analyzing the source code of the first version. The first application contains six classes, including one private class, with an average of 2.83 methods per class, 3.5 attributes, and 346 lines of code.

In comparison, as seen in Figure 10, the second application also contains six classes, with an average of seven methods per class, three attributes and a total of 621 lines of code. One can also see that the lines of code differ by a factor of around 1.8 and that the total class size is about 1.5 times bigger in the second application, suggesting the second application is almost twice as big as the first application. In addition, comparing the values between the number of methods and the lines of code in the class in both versions suggests a frequent occurrence of rather big methods in both applications. This correlation and further insights into the actual code quality are discussed more thoroughly in section 5.2, as this section only illustrates the scale of both applications.

These metrics should now give a better understanding and insight into the scale of both applications and help the reason for the differences in the functionality of the two versions shown in the following subsections.

4.2 Results from Version 1

Figure 11 shows the results generated by version one.



The image shows a window titled "Sudoku" with a standard 9x9 grid. The grid is filled with numbers 1 through 9, representing a solved puzzle. Below the grid is a button labeled "Solve". The window has a title bar with a minus sign, a maximize button, and a close button.

1	5	8	2	4	3	6	9	7
3	6	7	1	9	5	4	2	8
4	2	9	6	7	8	3	1	5
2	8	5	4	3	7	9	6	1
7	3	4	9	1	6	8	5	2
6	9	1	8	5	2	7	4	3
8	1	3	5	6	4	2	7	9
5	7	6	3	2	9	1	8	4
9	4	2	7	8	1	5	3	6

Figure 11: Result of the first version of the sudoku game

As can be seen, the tool managed to generate code that, in the end, displays a board with nine rows and nine columns, and each column and row has unique numbers ranging from one to nine, as intended for a typical sudoku game. However, one can almost instantly notice the problem with this game version. The puzzle is already solved and cannot be interacted with, which defeats the game's purpose. The only interaction is the "solve" button on the bottom of the board, which only generates a new, fully solved puzzle. This version can create the puzzle, but the user cannot play the game since the code that the tool provided made it so it always generates a fully solved board.

4.3 Results from Version 2

Figure 12 shows the results generated by version two.

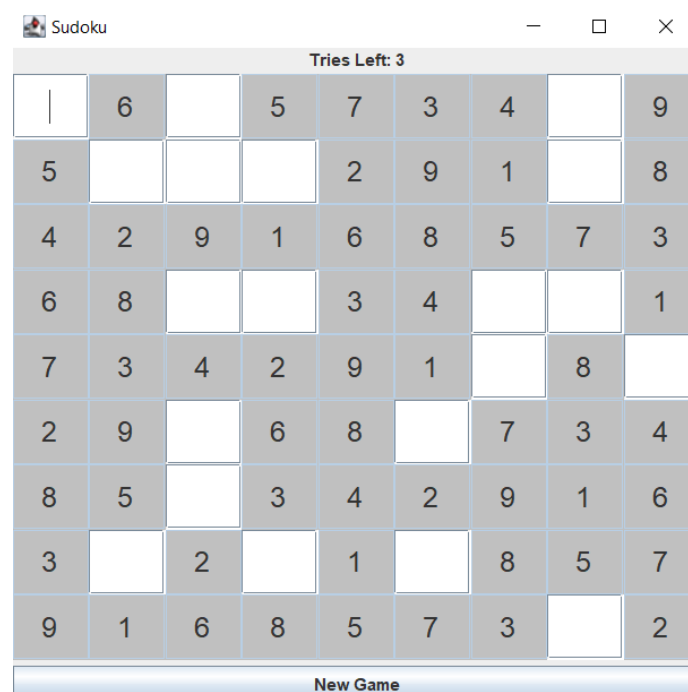


Figure 12: Result of the second version of the sudoku game

Compared to version one, the second version is now fully working and playable by the user. It still generates the complete puzzle and displays the board, but this time, it removes the numbers from random cells so the user can try filling them out as intended. This version also includes a "New Game" button to create a new game and the number of tries left for the user to guess the missing number. These features were prompted additionally after the game had already started working.

5 Evaluation

This section will detail the differences between the two versions while evaluating each specific version based on different criteria and metrics and pointing out the problems that occurred by combining the generated code snippets from the chatbot and trying to make the code compile, as well as how the functionality of the applications were tested.

5.1 Quantitative Measurements of the different Prompts

For this specific subsection, the focus will be on acquiring various metrics on the differences between the two versions and their respective prompts. This includes the number of prompts and other aspects like the longest and shortest prompt between the two versions and their average length.

The results were achieved by separating the conversation data, exported by chatGPT in an HTML file, and filtering out all responses from the chatbot so only the input prompts were left, which allowed for easier measuring. The differences will be quantified by the number of characters and words they include to determine the respective longest and shortest prompts, etc.

To count the number of prompts, the conversation of each version was put into a text editor and then by using "ctrl + f" to search for the keyword "User", the command would display the number of occurrences. The search command was set to case sensitive since all keywords were in upper case to ensure that the word "user" was not used in a response or anywhere else than to indicate that a prompt was starting. An example of how the cleaned version with the "User" keyword looks can be seen in Figure 13 below.

```
java Sudoku Game.
User
How would I go about creating a sudokun game in java using Object orientated programming and having a working GUI

User
Can you give me the code for step 1

User
Can you make code for step 2

User
Can you give the code for step 3

User
can you give me the code for step 4

User
Can you give me the code for step 5

User
Can you combine all these classes into one and make the game fully working

User
Continue completing the code from where you left off

User
Write the generateBoard method from sudokuboard since its currently missing
```

Figure 13: The whole conversation of version one. It is manually filtered to show the input prompts and no responses. One can notice the "User" keyword above each text to indicate the start of a prompt.

All the filtered conversations, the unfiltered separate versions and the single HTML file containing both chats when exporting the data can again be found on the Git Hub under the "ConversationData" folder.

To count the amount of character and words the raw text was simply copied and pasted to a online software that would then do the job. Note that for the amount of characters, the white space in the text was also counted, which is why the amount of words will also be presented.

5.1.1 Version 1

The first version contains a total amount of nine input prompts. This amount can be classified in three different ways. As we already know, the initial prompt starts the conversation with the user's need. Naturally, there is only one initial prompt. Then, the follow-up prompts follow up on the responses from the chatbot and ask to go further into detail. They consist of seven in total. The last prompts class, containing only one element, would ask the tool to continue generating code.

The longest was the initial prompt, with 20 words and 113 characters. The shortest prompt consists of seven words and 28 characters: "Can you make code for step 2?" which belongs to the "follow-up" class. Furthermore, the average prompt length is $498/9 = 55.3$ characters and $98/9 = 10.8$ words. Note that in version one, unlike version two, since this version is more straightforward, it did not need much modification for the code to be able to compile, so the prompts don't have copied code snippets in them asking for bugs to be fixed but that will be elaborated on more in the following sections.

5.1.2 Version 2

As explained in section 3.2 for the second version, the author tried to put more detail into the prompts in the hope of a better end product. The effort can be seen by looking at the following numbers.

The number of prompts the user made for this conversation was 45, 4.5 times more than for the first version. These can, like for version one, be put in different classes. The difference is that for this version, there are some additional classes. We again have one initial prompt. This is followed by three prompts asking to continue generating the response like before. Now, for the rest of the classes adding to the "follow-up" class, there are two additional classes for this version. These would include prompts related to bug fixing and prompts related to missing implementation. Taking all these three classes into consideration, there would be seven prompts belonging to the "bug-fixing" class, 14 belonging to the "missing implementation" class and 20 to the already known "follow-up" class. The difference between the "follow-up" and "missing implementation" is mainly that for the first, the purpose of the follow-up prompt would be to advance the

conversation and get to the following code implementation, while for the latter, the prompt was still dealing with the current response and trying to get the complete code since sometimes specific code would be missing but more on this in section 5.2. The criteria for the "bug-fixing" class would be all prompts that would ask the bot to fix bugs in their implementation. These prompts would usually also contain the specific code snippet regarding the bug, making them the longest prompts of all the classes for this version, as can be seen with the following numbers.

The longest prompt for this version consists of 3849 characters and 299 words, which belongs to the "bug-fixing" class. The longest prompt without any code snippets and pure text is the initial prompt with 400 characters and 74 words, about 3.5-3.7 times more words and characters than the initial prompt from the first version. The shortest text consists of one word and eight characters, containing the text: "continue" and belongs to this respective class. The shortest prompt ignoring the "continue" message is four words and 28 characters long: "What is the solutionsCount?", which belongs to the "follow-up" class. The average prompt length for this version is $13315/45 = 295.9$ characters and $1507/45 = 33.5$ words, having around six times more characters than version one. Figures 14 and 15 further visually showcase the differences between the prompts of version one and version two.

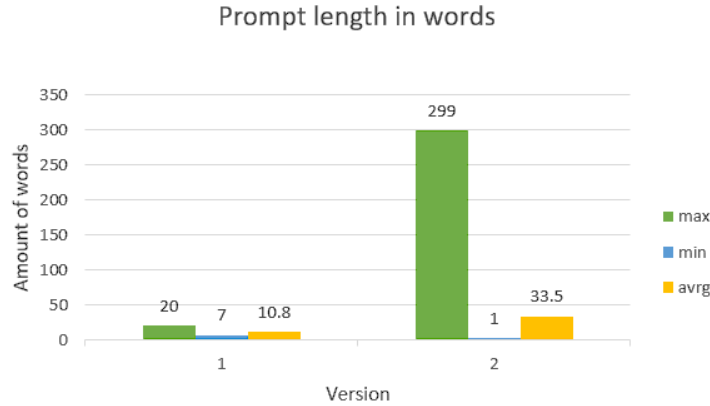


Figure 14: This diagram illustrates the differences in prompt length using the number of words as a metric. The green bar represents the maximum number of words in a prompt, the blue bar the minimum and the yellow bar the average number of words through all prompts of their respective version. It can be seen That version two has, on average, longer prompts than version one.

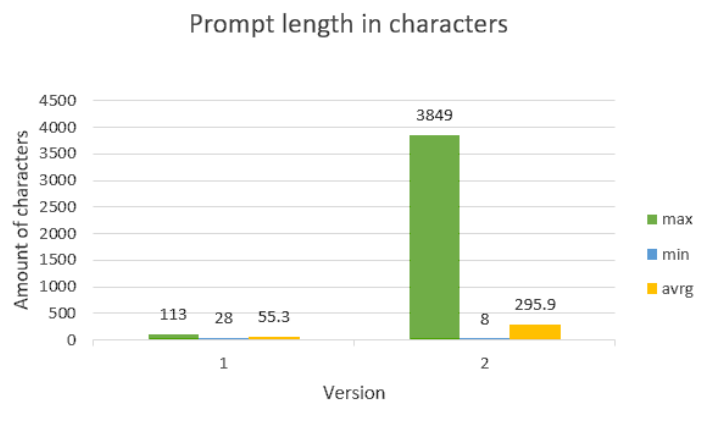


Figure 15: This diagram illustrates the differences in prompt length using the number of characters as a metric. The bars represent the same meaning as in Figure 12, the difference being that this time, the number of characters is used instead of the number of words. The result remains the same as in Figure 12.

This version has, on average, more and longer prompts than version one, which makes sense considering that more effort was put into formulating them to achieve a better product. The added complexity in the formulations also leads to an increase in code complexity generated by the chatbot by trying to complete the task given by the user. This created several bugs that, in correlation, also led to additional prompts asking the tool to fix them, increasing the total number of prompts and their average length by including code snippets as input.

But while more extended and more prompts can lead to a better result in the end, as mentioned with, for example, bugs, they can also yield more problems that could be annoying while developing the application. The following section will highlight these problems in more detail.

5.2 Code Quality

This subsection will focus on metrics used in object-oriented software development to evaluate the code quality of both versions by comparing them. The specific metrics used will be the Chidamber and Kemerer metrics in the following subsection and the previously established metrics in section four.

5.2.1 Chidamber and Kemerer

The Chidamber and Kemerer metrics (CK-metrics) were first introduced by Shyam R.Chidamber and Chris F.Kemerer in their paper titled "A Metrics Suite for Object-Orientated Design", published in 1994. [1] The CK-metrics suite contains six metrics. Figure 16 shows all of these six metrics for version

one.

Next follows a quick explanation of these metrics for readers who are unfamiliar with them; others can skip this section. From left to right in Figure 16, the first is "Coupling Between Objects" (CBO). CBO defines the number of other classes to which the specific class is coupled, for example, in terms of associations and dependencies [1]. This metric can be used to evaluate a software's maintainability and flexibility. Usually, a high CBO is not desired because excessive coupling between classes is detrimental to modular design and prevents reuse. Next is the "Depth of Inheritance Tree" (DIT), which defines the length of the longest path from a class to the root class of the inheritance hierarchy [1]. The DIT measures how many super-classes can affect a class. Usually, the deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex. The third metric is "Lack of Cohesion of Methods" (LCOM). LCOM, like in the name, describes the lack of cohesion among the methods of a class [1]. It measures how unrelated, specific methods within a class are and tries to help assess the design of the class. It represents the difference between the number of methods with standard instance variables and those without instance variables. Next is "Number of Children" (NOC), which counts the immediate sub-classes of a class [1]. It measures the number of classes directly inheriting from the specific class. The NOC can roughly indicate the level of re-usability in an application, with the NOC growing, meaning that the re-usability increases. An increasing NOC also means that the amount of testing should increase too because more children in a class indicate more responsibility. The fifth metric is "Response For a Class" (RFC). RFC describes the sum of the number of methods that can be invoked in response to a message received by an object [1]. Its purpose is to help provide insights into the potential complexity of the class and its interaction with other classes. Usually, the maintainability declines with an increasing RFC. The last metric is "Weighted Method Count" (WMC). This metric represents a weighted sum of methods implemented within a class. It is parameterized in a way to compute the weight in each method, with the weight usually being the lines of code or the "McCabe Cyclomatic Complexity" (CC). The CC value is a measure of the control structure complexity. It is the number of linearly independent paths, so the minimum number of independent paths can be taken inside a code structure. It can, therefore, also indicate the complexity of a method and measure its maintainability, where complexity increases with higher CC and maintainability decreases since there is a higher probability for errors when fixing or refactoring more complex code structures [1]. Therefore, WMC is usually used to give an understanding of the overall complexity of a class based on the number and complexity of its methods. Note that WMC can also be unweighted; in that case, the "weight" consists of the number one as a constant.

5.2.2 Evaluation of the CK-Metrics

Figure 16 shows the CK metrics for the first version, while Figure 17 shows the CK metrics for the second version. Note that for the WMC, the tool used to acquire the metrics utilized McCabe Cyclomatic Complexity (CC) as a weight to obtain the value.







class	▲	CBO	DIT	LCOM	NOC	RFC	WMC
 SudokuGUI		1	1	0	0	15	3
 SudokuGam		1	2	1	0	3	1
 SudokuValic		1	1	1	0	2	8
 SudokuSolve		2	1	1	0	4	15
 SudokuGam		3	1	1	0	38	15
 SudokuBoard		5	1	1	0	11	29
Total							71
Average		2.17	1.17	0.83	0.00	12.17	11.83

Figure 16: Obtained CK metrics after analyzing the source code from the first application.







class	▲	CBO	DIT	LCOM	NOC	RFC	WMC
 Validator		3	1	1	0	12	20
 Generator		4	1	1	0	20	16
 Solver		4	1	1	0	11	19
 Cell		5	1	3	0	13	12
 GUI		5	6	2	0	36	26
 Board		6	1	1	0	16	40
Total							133
Average		4.50	1.83	1.50	0.00	18.00	22.17

Figure 17: Obtained CK metrics after analyzing the source code from the second application.

By comparing both Figures, what can immediately be noticed is that the second application has a higher total WMC than the first and that the average WMC differs by around a factor of two, so roughly double the first version. This was to be expected since the prompts for the second version were more detailed. Thus, more responsibilities were added to the application, which resulted in a more complex code. Looking more into each class instead of the whole project, the "Board" class of the second application has a WMC of 40, which is high. Adding onto that, the class has 13 methods. This indicates that the class has too many methods, and some methods are complex, affecting the software's maintainability and re-usability since the class is more application-specific. Another case of a high WMC is in the "SudokuBoard" class in version one. Note that the highest WMC for both versions was in their respective class to handle

the board's responsibility. However, in the case of version one, the class has only two methods with a WMC of 29. This suggests that the methods are complex and have many control structures, decreasing the overall maintainability and further the readability of the specific methods. Some of these structures should be refactored into separate methods to give them a specific purpose and increase their readability. The rest of the WMC values of the other classes are still relatively high but not too extreme.

Moving on to RFC, the second version has a higher average amount. What can be noticed is that the second version has all RFC values in double digits, unlike the first, but the first one has the highest value in a class considering both applications, which is the "SudokuGame" class with an amount of 38. The second highest was the "GUI" class in the second version, with 36. A high RFC suggests low maintainability again since changes to the existing code can easily lead to bugs since each modification must ensure that all possible methods that can be invoked still have to work correctly and have to be able to be executed [1]. It also increases the amount of testing needed since more test cases need to be written for all the possible ways the methods could be invoked. Debugging is also more complicated with a high RFC.

Both applications have zero number of children (NOC), suggesting that there is no inheritance in both of them [1]. The reason for this may be in the way of prompting since each class was asked for separately, so the AI-generated the code without giving too much thought to the other classes and focused on the actual functionality of the code rather than the application as a whole. For the second version, the prompts especially included a phrase similar to: "in respect to previous classes." This mainly resulted in some method and variable names used in different classes being the same as they should have been but did not make the AI consider using inheritance for the project. Explicitly ordering the AI to use inheritance could have given different results. This can be considered for future projects.

Next is up the "Coupling Between Objects" (COB). The CBO is higher for the second version, implying more class coupling. A high CBO is generally not desired since it impacts re-usability because the class is less independent [1]. So, the higher the number of couples, the higher the sensitivity to changes. In this case, their respective "Board" classes, like RFC and WMC, have the highest CBO values for both applications. The second application is expected to have a higher average of CBO since it is more functional than the first. One can also see that the lowest CBO in the second version is three, which means that always at least half the classes are coupled together, so in case of an essential change in the code, at minimum half of the other classes have to undergo some change to, for example, a change in the attribute name. This would be undesirable in large projects, but since this project only has six classes and is relatively tiny, it is still okay.

For DIT and LCOM, not much more will be added since it is mostly the same for both versions, with most classes having a value of one, and the other values are more interesting to look at for this project. However, one significant fact is the "GUI" class of the second application, which has a DIT of six. This is because the GUI class extends the JFrame class, which is commonly used to make GUI in Java and is a part of the javax.swing package.[3].

5.2.3 Additional Metrics and Evaluation on the Code Quality

This subsection will briefly revisit the metrics from the results in section 4.1, go further into detail, and comment on the specific values and what they mean for the overall code quality of the applications. As seen in Figures 9 and 10 of section 4.1, version one has 346 lines of code and version two has 621 lines with averages of 58 and 104. The number of attributes for the second version is acceptable. As for the first version, the "SudokuGame" class has 15 attributes, which is too much and could indicate a god class since it also has the second most lines of code in its project. On the other hand, the second version has two classes that have a large number of methods. Version 2 also has way more methods, indicating a higher complexity than version one, which can be expected since the second application actually works compared to the first one and also has additional features that the first application does not have (Try new game button, amount of tries left, etc.). Those things aside, as mentioned in section 4.1, comparing the number of methods and the lines of code of the respective classes suggests rather extensive methods in some of them, especially in version one, where the "SudokuGame" class has only three methods, but 87 lines of code, excluding the lines for the attributes. In the second application, the "GUI" class has five methods and 191 lines of code, excluding attributes again. This suggests long methods, and the previous CK metrics also hint at a high application complexity. The code quality could be better overall, as the methods could have been split up or refactored into separate classes depending on their responsibilities. The latter may be because the list of classes that should occur was already predefined during the first prompts, so the AI tried to have the exact number of classes as in the list. This would mean that the AI wrongly evaluated the responsibilities for each object for a sudoku game by restricting itself to six classes in both versions, thus highly likely violating the single responsibility principle.

5.3 Problems that frequently occurred

This subsection will focus on general problems that occurred while putting together the generated code snippets from the chatbot. Since the first version was relatively simple, there were no real difficulties in assembling the application and making it run. There were only minimal problems that hindered the code from compiling, but they could be solved in a short time. The more significant issues regarding functionality or missing code were, as explained in section 3.1, ignored to keep the prompt complexity as simple as possible so that the complex part could be focused on version two and thus better compare the differences

between these two.

As already hinted with the "missing-implementation" and "bug-fixing" classes, the second version had a higher amount and also more complex problems than only changing variable names once.

5.3.1 End of the Response in the Middle of Code Generation

One of these problems would be that responses with much code usually stop right in the middle of the code generation. This requires an additional prompt telling the tool to continue generating the code from where it left off. An example can be seen in Figure 18 below.

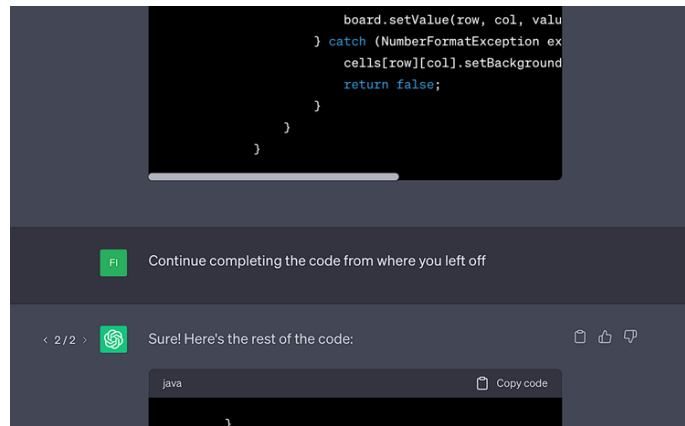


Figure 18: This figure shows a snippet of the response stopping during code generation, followed by the prompt asking for continuation. The bot would then respond by generating the code from where it left off.

This made it rather tedious to copy and paste the code snippets and apply them together since the code was cut off through the middle of the generation. However, at the time this paper is written, there now exists an additional feature that detects when such an event would happen and then asks in the form of an interactable button if the generation should be continued. The problem with adding the code remains, but at least the user can ask the tool to keep generating the code without explicitly requesting the chatbot.

5.3.2 Missing Code Implementations

The most bothersome problems during the process were missing code implementations in the responses. When asking the AI to generate the respective code for a specific class, it would sometimes leave out the actual implementation of specific methods or logic and instead write a comment on what needs to be done at that particular location in the code. This usually happens if the generated code is a long one. A noticeable example of this problem can be seen in Figure 19.

```
private int getNumSolutions() {  
    // TODO: Implement a Sudoku solver to count the number of solutions  
    return 0;  
}
```

Figure 19: The `getNumSolutions()` method in the "Generator" class has no concrete implementation. Instead, all the bot gave was a comment on what the method was supposed to do.

To combat this, the author tried following up on the "empty" methods by providing the same method as input and then asking if the chatbot could complete the specific method. Figure 20 illustrates this particular event, while Figure 21 shows the response.

It can be noticed that the code is complete this time, but some things still need to be fixed. After trying to include the missing content in the current code, the compiler immediately spotted that the new code has methods from the solver class that do not exist, such as the `solver.hasMoreSolutions()` method. Even though the logic of the code might be correct, it does not consider the current implementation of certain classes. It introduces new methods or, in rare cases, the same methods but with a different name, which is tedious since the user has to spot these problems now and try to fix them by asking for new code, etc. This led to even more methods that did not exist and created more bugs in the code. After some time and many prompts later, as well as manually fixing the bugs, the code finally could adequately compile and run.

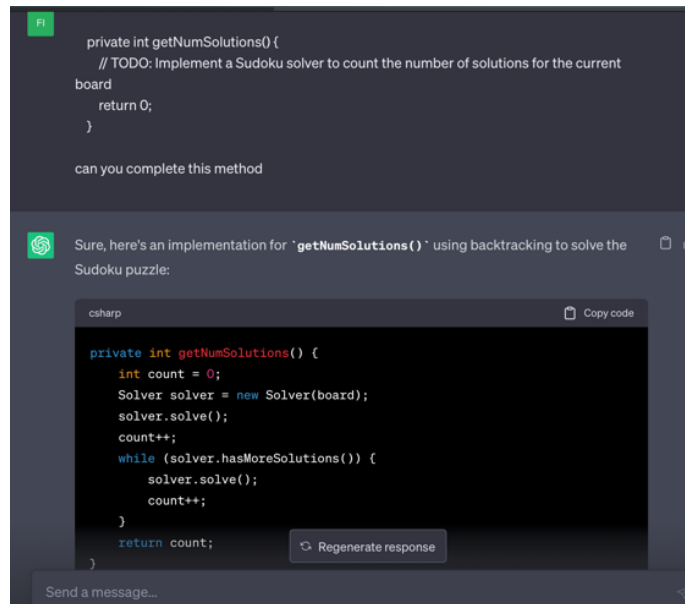


Figure 20: The follow-up prompt asking the tool to generate the missing code in the `getNumSolutions()` method.

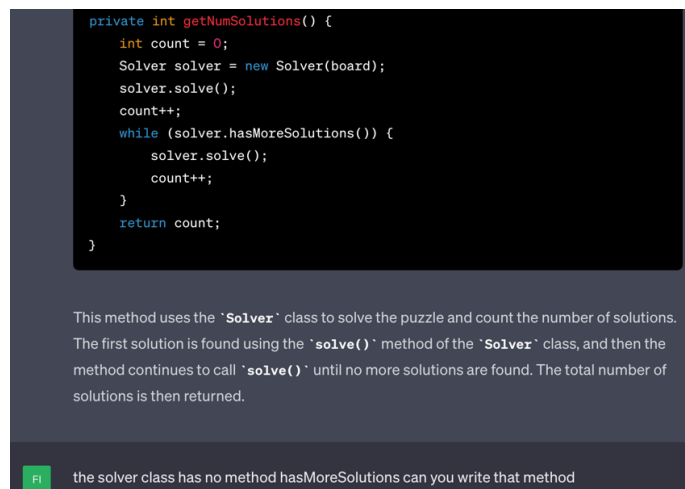


Figure 21: This figure shows the response of the AI after being asked to generate the missing code for the `getNumSolutions()` method in the "Generator" class. Below the reaction is an additional prompt asking for the code for another absent implementation.

5.3.3 Bug Fixing

The bugs mainly consisted of wrong variable names across the classes and missing methods, especially after asking for missing implementations to be generated. The author tried to let the chatbot fix the bugs, and for some, it worked. Still, for most, there had to be a self-intervention where the author had to manually fix the bug by simply matching the method and variable names, which was also much faster than trying to make the AI handle it.

Rare, occasional logical errors also occurred during the project, where some code functionality was wrong or not working as intended. Interestingly enough, the first big problem that occurred was the same as in version one: the game generates an already fully solved board. The author tried letting the AI fix the bug by pointing out that the desired functionality still needs to be met and repeating that the puzzle should not be fully solved so the user can play the game. After some back and forth and tedious prompting, the chatbot finally understood the problem and tried to modify the code to fix it. This led to new issues, like, again, some methods that do not exist, but after asking the chatbot to generate the code for these methods, the code could be recompiled. But in the end, after all these prompts, the code was still not working as intended, so the author manually fixed the bug by commenting out some logical flaws in the implementation of the "generate()" method in the "Generator" class.

Another big problem was that as soon as you got one cell right, the game treated it like the whole puzzle got solved, even though there would still be other cells with no values left. This time, instead of asking the AI, the author used a debugger tool and fixed the bug, which was way faster and easier.

Note that the source codes on the Git Hub contain the final code of both sudoku versions, with both including comments in the code explaining what was modified so that the application could run in the end. Since there are also comments made by chatGPT to explain what is happening in the code, to distinguish between comments from chatGPT and the user, the comment would start with the keyword "Author:". Furthermore, instead of deleting non-working or faulty code, it was commented out to comprehend better what happened during the manual bug fixing.

5.4 Testing

Note that the functionality of both versions was tested manually. Since the second version is playable, the tests mainly focused on the second application. The application underwent multiple tests with irregular inputs such as letters, numbers not in the range of one to nine and special characters. Inserting any input that was not a number from one to nine would generate an error message. Figure 22 shows an example of such an error message.

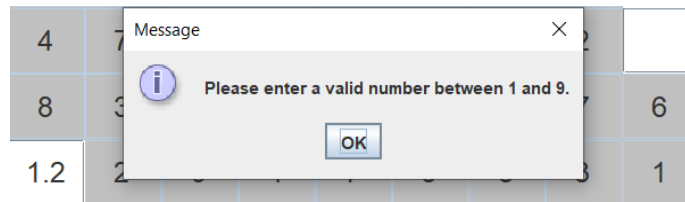


Figure 22: Error message for a wrong input, in this example being the number 1.2 which is not a number between one to nine

Appropriate tests for the additional features, such as creating a new game or having a specific number of attempts left, also went through manual tests by simply playing the game multiple times. For example, to test the number of tries left, it was only necessary to give wrong inputs and watch if the counter decreased and the game lost when no tries were left anymore.

Typically, for these functionalities and properties of the application, appropriate test cases could also be written that would verify that the application works. However, for different reasons mentioned in the following sentences, this was not done. Since the code was generated by artificial intelligence and not written by the author, it was not easy to write proper test cases without reading through the entire code and trying to understand it first. The required tests to check if the application works correctly were simple because we have chosen a more straightforward case study, so these tests could also be done well enough manually. The author also tried to let chatGPT handle and generate the test cases. However, even with the provided context of the initial code, the generated test could not correctly be integrated into the existing source code without forcing the author to interfere with the source code too much and having to write extra methods and alter the code too much. So, in the end, for those reasons and time constraints, testing the application manually was the solution the author chose.

6 Discussion

In this section, the key findings from the evaluation will be summarized and then discussed concerning the two sudoku versions and how they were affected by the differences in prompts and the effort needed to create the application as well as how it impacted the code quality. In the end, the last subsection will discuss about the threats to validity of this project in correlation to this specific case study.

6.1 Summary of Key Findings

It was shown that the first version has fewer prompts than the second version, which are also shorter on average and thus less detailed. The prompts from the first version can be categorized as "follow-up", "initial", and "continue" prompts. At the same time, the second version also includes prompts specifically for bug fixing and missing implementations, where the "bug-fixing" prompts would usually contain code snippets as input.

Furthermore, unlike the first, various problems were encountered in the second version that complicated the process of assembling the generated code snippets, like missing implementations and such. In the end, the new prompts asking for bug fixes and generating missing code would, as explained in section 5.2, lead to even more bugs than before, ultimately almost forcing the author to fix them manually himself to get a running application, while the first version was running after minutes of assembling the code. However, in the end, even though it took longer, version two worked as intended and was far more functional than version one.

Overall, the second application was larger than the first application, both in terms of functionality and in the general scale of the application. The second application had more lines of code, more methods, and bigger classes; it was also more complex. However, this fact recognizes that even though the second application was on a bigger scale, the general code quality was not much better than the first version. Both versions have many faults, be it too many methods in a class or too long or complex. These factors lead to low maintainability, readability, and re-usability, making the code for both applications unsuitable for genuine software. However, since the project was on a small case, the AI's efforts for the second version were enough to make the application work, even though some things had to be changed manually.

The increased complexity and low readability also make it hard to develop suitable test cases for the application without changing too much of the original code, leading to the author testing the application manually.

6.2 Implications

By going through these key findings, one can imply that version two had a better final result because of the more detailed and longer prompts. The initial prompt was way more thorough in version two, being around 3.5 times longer than the initial prompt of version one. This allowed for a more solid base to follow up with and get better responses from the chatbot since the AI had more requirements to work with. On the other hand, these longer prompts also caused some problems, which slowed the overall development in contrast to version one, which did not have these difficulties. Furthermore this better result only applies to the functionality of the application. The overall code and its quality did not improve compared to the first version and are still bad for both when looking at the discussed metrics in the previous section.

Overall, it can be implied in the case of this particular study that the code generated by the more detailed prompts is more prone to having bugs when combining the code into a working application, while not having a much better general code quality, but also yields better results regarding the final product.

6.3 Threats to Validity

The following subsections discuss the internal and external validity of the case study to provide an understanding of the limitations connected with the methodology of this project.

6.3.1 Internal Validity

The case study's reliance on the author's encounter with chatGPT raises the possibility of sample bias because it suggests that the conclusions reached might not accurately reflect the experiences of a wider variety of users. Making sudoku may have affected how broadly applicable the results are. The author's specific request for a sudoku application may not represent other programming tasks. Furthermore, since the author is familiar with ChatGPT, it could lead to a learning curve bias. Since the author requested code generation, any bias in understanding or navigating the chatbot may have influenced the evaluation.

6.3.2 External Validity

The study's findings could be tainted by outside variables like upgrades or modifications to ChatGPT's features. Furthermore, the fast-evolving technology in chatbots may impact the relevance of the findings. The study's limited generalizability may stem from its emphasis on the author's interactions with ChatGPT for Sudoku code generation. It shall be explicitly acknowledged that the results are unique to this task and person, and care will be taken when extrapolating the findings to a larger setting. Lastly, the prevailing time constraints may have affected the depth of the code quality analysis and the evaluation of the results.

7 Conclusion and Future Work

In conclusion, this case study has demonstrated the potential of using AI-powered chatbots like chatGPT with the help of developing working and functional software, with the code being almost entirely generated by AI alone. With minimal knowledge of the logic behind the code, the author made a fully working Sudoku application by simply putting together the generated code snippets and making them work together. However, it is also important to note that the project had challenges. Missing implementations, as well as bugs, were typical throughout the process and the overall code quality leaves a lot to be desired. While chatGPT can fix minor bugs in unrelated code snippets, it could not fix the code of a complex application with many different classes working together as one, so in the end, this had to be done by the user. This implies that the user should have at least some coding knowledge when working with the chatbot.

Sudoku is a well-defined game with clear rules written all over the internet. This played a big part in contributing to the success of the study case. But not all software projects are straightforward, like sudoku. Most projects have unique requirements that are not predefined, like, in this case, the rules of a puzzle game. Therefore, the application becomes more complex and diverse, which may make it harder for chatGPT to come up with valuable and working code.

As this case study represents a bachelor thesis, there were limitations regarding time and scope. To better understand the limits and potential of chatbots in software engineering, there needs to be further case studies on a larger scale involving projects with more diverse and unique requirements so that more data can be gathered to form a more precise conclusion.

To close this thesis, the author hopes that this paper can be used as a foundation for future works in exploring the topic of chatbots in software engineering, which will lead to even more insightful investigations regarding the potential and limitations of these tools.

The complete source code of both sudoku versions and the entire chat logs of the chatGPT conversations can be found on [Git Hub](#).

List of Figures

1	First ever prompt that was entered in chatGPT.	6
2	The result of the initial prompt that chatGPT generated, being a to-do list on how to implement a sudoku game in java using object orientated programming.	7
3	The figure shows the prompt used to generate the code snippet for the specific step in the to-do list from the result of the initial prompt.	7
4	The generated code for step 1 in the to-do list and an additional explanation of what the code does.	8
5	The initial prompt of the second version. This time, a more detailed start with clearer requirements for what the application should do.	9
6	The result of the initial prompt in version two, displaying a step by step to-do list and the classes needed to implement the sudoku game.	10
7	Similar prompt to the ones from the first version but adds the additional statement to respect the previous classes.	11
8	Another example of the extra statement.	11
9	Various software metrics after analyzing the first version of the application. The columns of the table represent from left to right in order: The class in question, the number of attributes in the class, the number of methods in the class, the number of lines of code and the total class size by combining the number of methods and attributes of the class. Each column also displays the total and average count of the given metric.	12
10	Various software metrics after analyzing the second version of the application. The table reads the same as in figure 9	13
11	Result of the first version of the sudoku game	14
12	Result of the second version of the sudoku game	15
13	The whole conversation of version one. It is manually filtered to show the input prompts and no responses. One can notice the "User" keyword above each text to indicate the start of a prompt.	16
14	This diagram illustrates the differences in prompt length using the number of words as a metric. The green bar represents the maximum number of words in a prompt, the blue bar the minimum and the yellow bar the average number of words through all prompts of their respective version. It can be seen That version two has, on average, longer prompts than version one.	18
15	This diagram illustrates the differences in prompt length using the number of characters as a metric. The bars represent the same meaning as in Figure 12, the difference being that this time, the number of characters is used instead of the number of words. The result remains the same as in Figure 12.	19

16	Obtained CK metrics after analyzing the source code from the first application.	21
17	Obtained CK metrics after analyzing the source code from the second application.	21
18	This figure shows a snippet of the response stopping during code generation, followed by the prompt asking for continuation. The bot would then respond by generating the code from where it left off.	24
19	The getNumSolutions() method in the "Generator" class has no concrete implementation. Instead, all the bot gave was a comment on what the method was supposed to do.	25
20	The follow-up prompt asking the tool to generate the missing code in the getNumSolutions() method.	26
21	This figure shows the response of the AI after being asked to generate the missing code for the getNumSolutions() method in the "Generator" class. Below the reaction is an additional prompt asking for the code for another absent implementation.	26
22	Error message for a wrong input, in this example being the number 1.2 which is not a number between one to nine	28

References

- [1] S.R. Chidamber and C.F. Kemerer. “A metrics suite for object oriented design”. In: *IEEE Transactions on Software Engineering* 20.6 (1994). Accessed 21-11-2023. URL: <https://ieeexplore.ieee.org/document/295895>.
- [2] OpenAI. “”Introducing ChatGPT”. In: *Introducing ChatGPT - Blog* (2022). Accessed 20-08-2023. URL: <https://openai.com/blog/chatgpt#OpenAI>.
- [3] Oracle. “Package javax.swing”. In: *Oracle* (). Accessed 10-12-2023. URL: <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>.
- [4] Justine Winata Purwoko et al. “Analysis ChatGPT Potential: Transforming Software Development with AI Chat Bots”. In: (2023). Accessed 13.01.2024. URL: <https://ieeexplore.ieee.org/document/10327087>.
- [5] Viriya Taecharungroj. “”What Can ChatGPT Do”?; Analyzing Early Reactions to the Innovative AI Chatbot on Twitter”. In: *Big Data and Cognitive Computing* 7 (2023). Accessed 20-08-2023. URL: <https://www.mdpi.com/2504-2289/7/1/35>.