Chatbots in Software Engineering

A Case Study using ChatGPT-4o

Software Engineering Project

Eda Nur Demir

Ahmet Burak Ekmekcioğlu

August 2024

Supervisor:

Prof. Dr. Timo Kehrer

Software Engineering Group

Institute of Computer Science

University of Bern, Switzerland

Abstract.

The rapid development of AI Technologies, particularly ChatGPT, has led to increasing curiosity about their use in software development. ChatGPT, has shown potential for generating code based on user prompts. This project explores ChatGPT-4o's ability to independently create a fully functional Sudoku game. Two distinct approaches are analyzed: one that resulted in partially functional code, requiring additional modifications, and another that produced a complete application despite facing several issues during development. This study assesses the role of ChatGPT-4o in automating software creation, examining both its capabilities and the limitations that necessitate human intervention for solving more complex problems. The results emphasize the potential of AI in software engineering while also recognizing its current boundaries.

Contents

1. Introduction

It has been a long-known fact that ChatGPT abruptly came into our lives and made huge changes on how we create, whether it is code or content for social media. What makes ChatGPT so special is that among all chatbots, it stands out due to its user-friendly interface, enabling users to ask questions and receive meaningful answers [1]. It has become a valuable tool for tasks like debugging, generating code snippets, or even fixing user-provided code. Therefore, ChatGPT serves as an assistant in the coding process, allowing users to produce functional software by assembling these code snippets.

This paper aims to explore whether AI chatbots can effectively generate functional software. By requesting specific code snippets from the chatbot and piecing them together, users can theoretically create fully functional software[2]. In this study, we replicate a prior investigation conducted with ChatGPT-3.5, which evaluated its role in software development. Our goal is to examine how using ChatGPT-4o changes the results and enhances the overall development process. The methodology and execution of the case study will be explained in the subsequent sections. Finally, we will assess the findings to determine the viability of relying on AI chatbots like ChatGPT-4o for software development with minimal user effort—where the user provides only the required functionality rather than writing any code snippets themselves.

2. Methodology

In this paper, we are going to focus on how ChatGPT handles a specific task that we divide into several steps. The goal is to run the ChatGPT generated source code for a specific application by solely prompting the AI.

The aim is to analyze how effectively the source code that is generated by ChatGPT performs and how much work is required to run the complete application. The specific case that is examined in the paper is sudoku, a puzzle game. The case study includes two different versions, each utilizing distinct prompting methods which results in different outcomes and conclusions.

Lastly, the AI-generated code will be run on a Windows machine, using Visual Studio Code, with necessary Java utilities.

2.1 Why ChatGPT4o?

An initial study was done using ChatGPT-3.5, in a bachelor thesis presented at the University of Bern. The goal of this study is to find out what happens when a similar if not the same methodology is used on ChatGPT-4o.

One may ask why ChatGPT-4o was used instead of the legacy model of ChatGPT-4. The reason is that as the developers of ChatGPT, OpenAI, suggests that the GPT-4o model is the “most optimal" so far as to solving complex tasks. That is why we have chosen this model so that we can push the model to its current limits of computation.

2.2 How this paper relates to the prior study

As mentioned before, this paper essentially replicates the methodology used in Nikolic’s paper. The only difference is that we use ChatGPT-4o. The prompts that we have fed ChatGPT are the same, so that we could control for the inputs. Moreover, the results of these prompts somewhat differ from those of the prior, which will be discussed later.

3. Execution

This section provides a detailed breakdown of how the entire case study was carried out. The explanations for the prompts used in both versions and the modifications required to make sure the resulting code compiled and executed properly are provided in the sections subsequent.

3.1 Sudoku Version 1

This is the first version to generate the code for a complete sudoku game. In order to observe what ChatGPT-4o would produce and how helpful it would be, the prompts were designed to be as brief as feasible. As in the prior paper, we provided a less detailed and less structured input, so that the chatbot can generate an output with a minimal amount of assisting feedback from the user.

3.1.1 Prompts

The first prompt of the conversation in version one can be seen in Figure 1 below, the same as in the paper prior.

> How would I go about creating a sudoku game in java using Object orientated programming and having a working GUI

Figure 1: First prompt that was entered in ChatGPT for version one.

The initial prompt is made of a simple phrase where the authors ask ChatGPT how the user would create a classic sudoku game using OOP in Java. As seen in Figure 2 below, the tool generates a step-by-step list of how the initial task can be completed.

> Creating a Sudoku game in Java with object-oriented programming (OOP) principles and a graphical user interface (GUI) can be broken down into several steps. Here's a high-level guide on how to approach it:
>
> **1. Design the OOP Structure**
>
> Break down the problem into classes, with each class representing a key component of the Sudoku game. Here's an outline of what the core classes might look like:
>
> **a. `SudokuBoard` Class**
>
> This class will represent the 9x9 Sudoku grid. It will store the state of the board and provide methods for interacting with it.

Figure 2: First part of the result of the initial prompt that ChatGPT-4 generated, being a to-do list on how to implement a Sudoku game in Java.

First it starts by giving the instructions for designing the OOP structure. ChatGPT-4 gives the explanations of the classes and then the sample code snippets one by one as seen in Figure 2. Then, it provides a sample code where the board class is defined. These code snippets are available on GitHub.

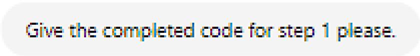


Figure 3: The figure shows the prompt used to generate the code snippet for the specific step in the to-do list from the result of the initial prompt

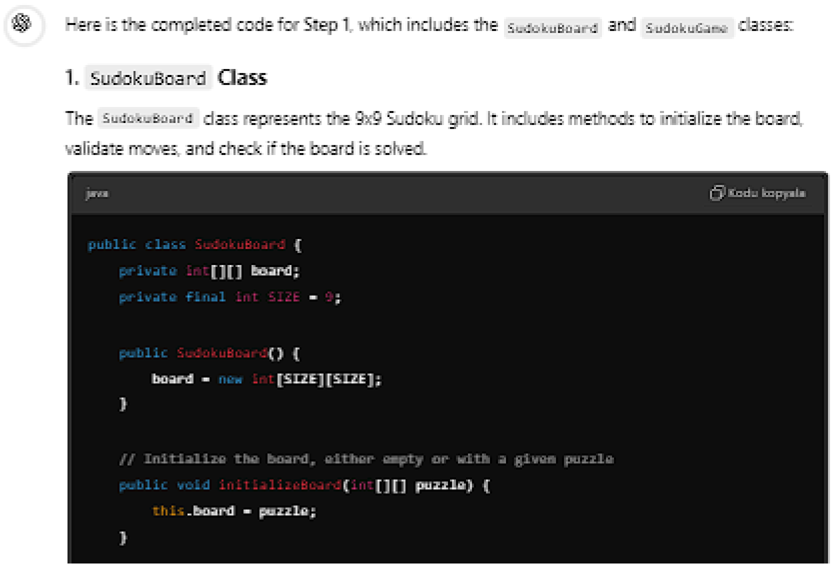
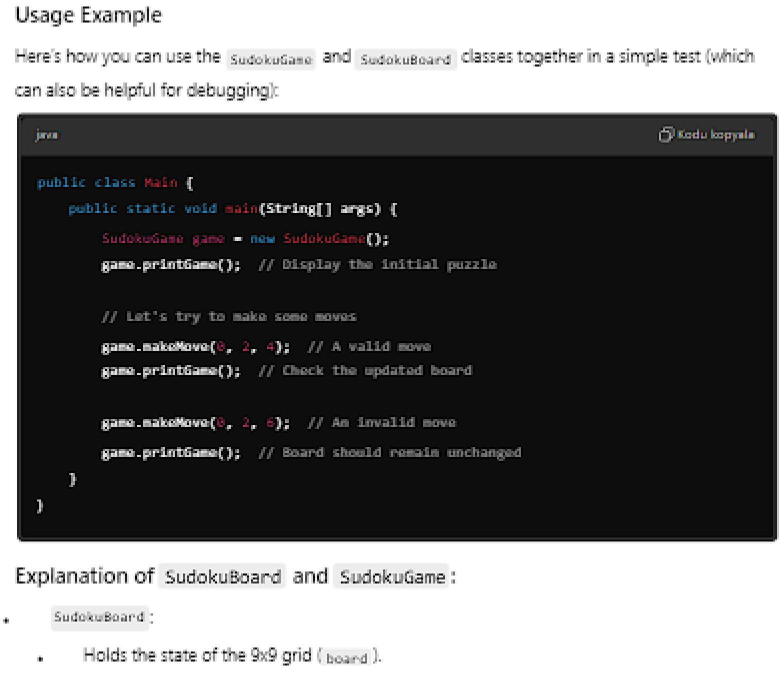


Figure 4: SudokuBoard class, part of the generated code for step 1 in the to-do list

This process is repeated for the remaining steps in the to-do list, concluding by asking the ChatGPT-4o to combine the snippets into a functioning application. The full log of prompts and results is once again available on GitHub. Once all the code is generated, the final task is to assemble it so that the source code compiles and runs successfully. Note that we did override the step 5, which ChatGPT-4o said is to run the game, by prompting it to combine to code and make it working.



Usage Example

Here's how you can use the `SudokuGame` and `SudokuBoard` classes together in a simple test (which can also be helpful for debugging):

java Kodu kopyala

```java
public class Main {
    public static void main(String[] args) {
        SudokuGame game = new SudokuGame();
        game.printGame();  // Display the initial puzzle

        // Let's try to make some moves
        game.makeMove(0, 2, 4);  // A valid move
        game.printGame();  // Check the updated board

        game.makeMove(0, 2, 6);  // An invalid move
        game.printGame();  // Board should remain unchanged
    }
}
```

Explanation of `SudokuBoard` and `SudokuGame`:

- `SudokuBoard`:
  - Holds the state of the 9x9 grid (`board`).

Figure 5: Additional explanation of what the code of the second prompt does.

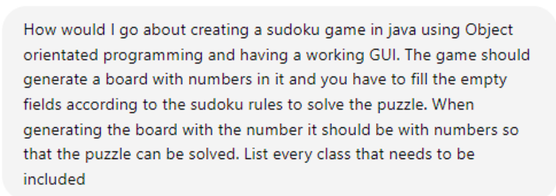
As seen in the Figure 5, ChatGPT-4 gives the explanation of what generated source code does as well as with an usage example.

3.2 Sudoku Version 2

For the second version, more specific, outlining detailed requirements are used. The main difference between these two versions is the initial prompt.

3.2.1 Prompts

Below is the second prompt fed into ChatGPT-4o.

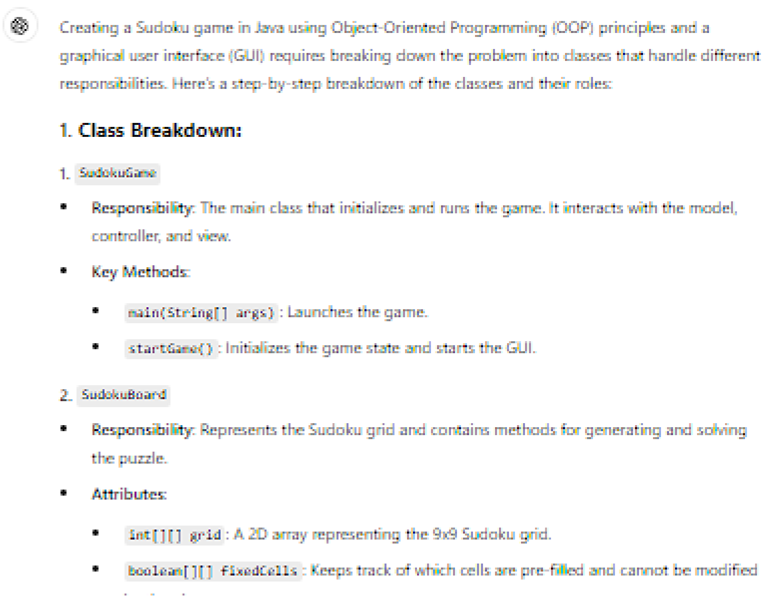


How would I go about creating a sudoku game in java using Object orientated programming and having a working GUI. The game should generate a board with numbers in it and you have to fill the empty fields according to the sudoku rules to solve the puzzle. When generating the board with the number it should be with numbers so that the puzzle can be solved. List every class that needs to be included

Figure 6: The initial prompt for the second version.

The first sentence is the same as the prompt from version one. However, in the second version, additional requirements are now included after the initial sentence to specify what ChatGPT-4o should generate.

Results of the prompts were satisfying, as the bot immersively outlined all the classes that needed to be implemented, and also steps to achieve that goal. We then continued to ask the bot to write down the codes for the respective classes, one by one.



Creating a Sudoku game in Java using Object-Oriented Programming (OOP) principles and a graphical user interface (GUI) requires breaking down the problem into classes that handle different responsibilities. Here's a step-by-step breakdown of the classes and their roles:

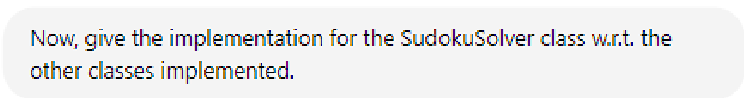
1. Class Breakdown:

1. `SudokuGame`
- **Responsibility**: The main class that initializes and runs the game. It interacts with the model, controller, and view.
- Key Methods:
  - `main(String[] args)` : Launches the game.
  - `startGame()` : Initializes the game state and starts the GUI.

2. `SudokuBoard`
- **Responsibility**: Represents the Sudoku grid and contains methods for generating and solving the puzzle.
- Attributes:
  - `int[][] grid` : A 2D array representing the 9x9 Sudoku grid.
  - `boolean[][] fixedCells` : Keeps track of which cells are pre-filled and cannot be modified

Figure 7: The partial 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.

Remaining prompts required the bot to provide us with the classes it outlined. In addition, we specifically asked the bot to implement the classes in consideration of the previous work before.



Now, give the implementation for the SudokuSolver class w.r.t. the other classes implemented.

Figure 8: Similar prompt to the ones from the first version with an extra statement to respect the previous classes.

As in version one, this process continues until all class code is generated. In the final step, the tool is asked to compile and combine the classes so it runs with minimal user modification.

4. Results

We will provide in this section the results of the codes generated by ChatGPT, as well as some comparisons between versions as to their software metrics evaluated in the prior study.

4.1 Insight to the General Scale of the Applications

The prior study suggests dividing the full code into classes and examining their number of attributes and of methods, lines of code and class size (defined as NOA+NOM) [3]. In our case however, specifically for the second version, this might lead to a confusion because the bot generated the completed code as a whole class named SudokuGame, which include static classes that help with the other functionality of the program. Thus, it is best for us to understand them as separate classes, having their own attributes and methods. In addition, we highlight that the constructor methods are as well counted. Below are the figures for both versions.

Version 1				
Class	NOA	NOM	LOC	CS
SudokuGame	6	9	221	15

Version 2				
Class	NOA	NOM	LOC	CS
SudokuGame	7	5	142	12
SudokuBoard	2	13	118	15
SudokuSolver	0	2	33	2
SudokuValidator	0	5	56	5
SudokuController	3	7	50	10
Total	12	32	399	44
Average	2.4	6.4	79.8	8.8

Figure 9: Excel sheet indicating class NOA, NOM, LOC and CS for both versions.

Notice that in the first version generated, there exists only one class, with all attributes and methods included. There are 6 attributes and 9 methods in this class, including the main function, and it takes only 221 lines of code for the whole program to run. On the other hand, in the second version, there are 5 classes, 4 of which are statics of SudokuGame class, and it takes 399 lines of code excluding the import statements. On average, we have 2.4 attributes, 6.4 methods and 79.8 lines of code per class. It is evident that the second version has less NOA and NOM per class compared to the first. One may also notice that the second program is twice as big as the first, 221 to 399 lines of codes.

4.2 Results from Version 1

Figure 10 is a screenshot we get when we execute the first version of the code.

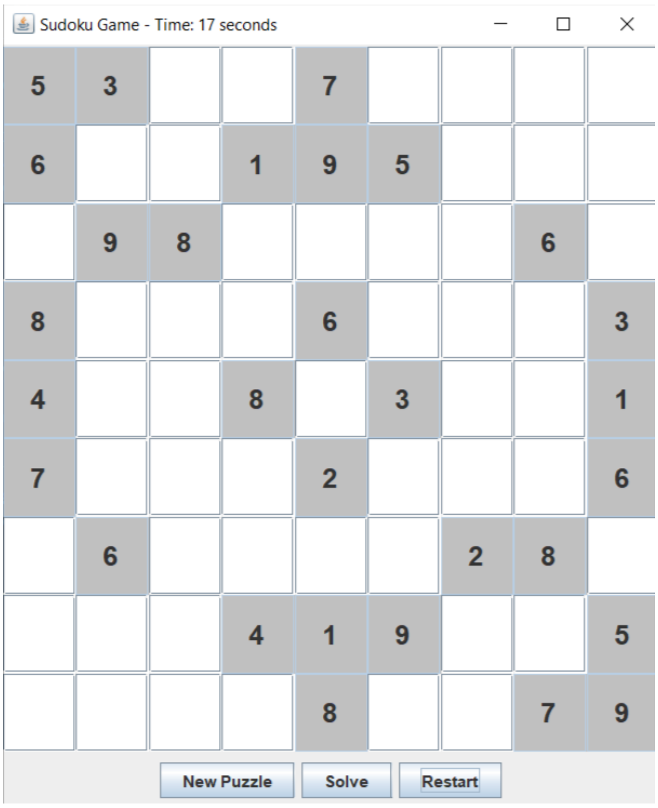


Figure 10: Results of the first version of the game

One may immediately notice that we have a full 9x9 grid, some cells pre-filled with numbers, 3 interactive buttons, and empty fields that can be filled with user input. The grid seems to respect the rules of 3x3, rows and columns. It also utilizes a timer on top. The first problem, however, is that none of the buttons below are functional; therefore, the player cannot interact with the game even if they solve the puzzle. The second is that since the board is hard-coded, every time a new game starts, players get the same board.

4.3 Results from Version 2

Figure 11 is the screenshot when we hit the "Run" command and click on "New Game" in the second version of the code.

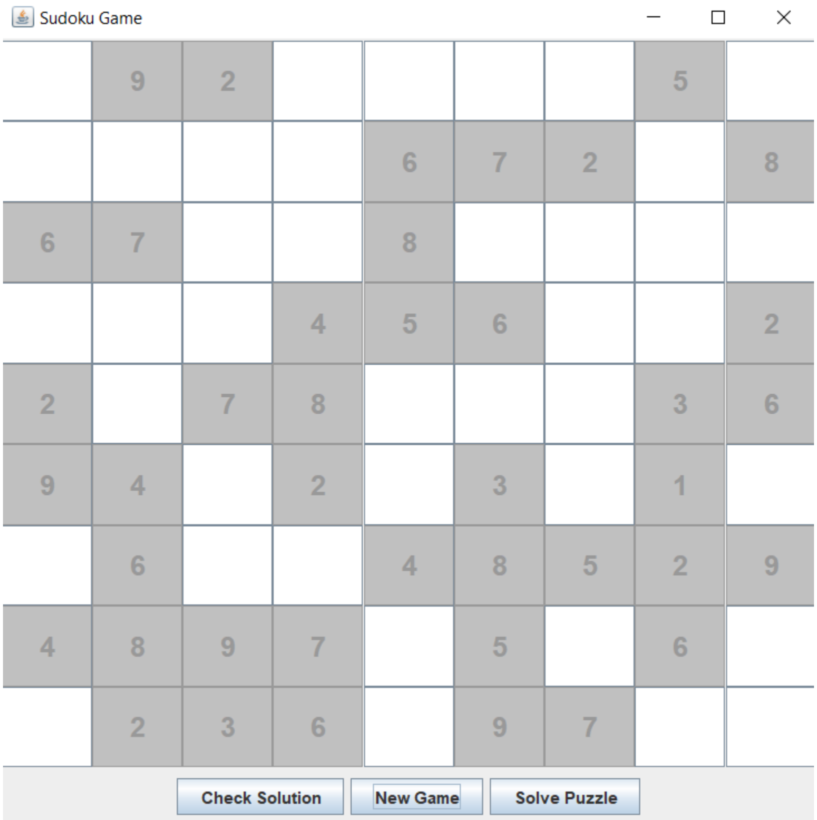


Figure 11: Result of the second version

In this version of the game, we have again an interactive pre-filled board, and 3 functional buttons for checking the solution, creating a new board, and solving the puzzle. As the first version, it respects the rules of the game, while prohibiting input of the same numbers on the same column, same row, or the same 3x3 grid. Additionally, it includes a feature of removal of numbers from cells filled by a mistake, by the user; as well as a solver method, which ultimately solves the puzzle, in the course of game, or at the start.

5. Evaluation

In this section, we will follow some of the evaluation criteria used in the prior paper, skipping quantitative measurements of the prompts and discussion of some of the problems encountered during the chat with the bot. The reason for the prior is that our chat did as well follow the same course with the bot as in the previous study, and for the latter is that we have not faced the troubles regarding code generation, whether it is not fully implemented nor the bot cut the conversation in the middle [3].

5.1 Code Quality

This section will evaluate the generated code using the same criteria presented in the previous paper by Nikolic. They have before explained and used the Chidamber and Kemerer metrics, as well as the metrics shown in section 4.

5.1.1 Evaluation of the CK-Metrics

Figure 12 indicates the CK metrics for both versions below.

Version1						
Class	CBO	DIT	LCOM	NOC	RFC	WMC
SudokuGame	1	6	4	0	34	15
Version2						
Class	CBO	DIT	LCOM	NOC	RFC	WMC
SudokuGame	5	6	4	0	41	12
SudokuBoard	3	1	0	0	20	15
SudokuSolver	3	1	0	0	4	2
SudokuValidator	3	1	0	0	4	5
SudokuControlle	2	1	0	0	18	10
Total	16	10	4	0	87	44
Average	3.2	2	0.8	0	17.4	8.8

Figure 12: Results of the application of CK criteria

The comparison between versions reveals significant differences in key object-oriented metrics, which reflect changes in complexity, cohesion, and coupling. Focusing on the Coupling Between Objects (CBO), Version 1 has a CBO of 1 for the SudokuGame class, while Version 2 exhibits a total CBO of 16, with an average of 3.2. This increase indicates that the classes in Version 2 are more tightly coupled, meaning there are more dependencies between them. Higher coupling often leads to more complexity and makes maintaining the codebase more challenging. In contrast, Version 1 has much lower coupling, which suggests fewer interdependencies, leading to easier maintainability.

Next, the Depth of Inheritance Tree (DIT) in both versions presents some contrasts. Version 1's SudokuGame class has a DIT of 6, which remains the same in Version 2, but the total DIT across all classes in Version 2 is 10, with an average of 2. This lower average indicates that the inheritance tree is more evenly spread in Version 2, leading to a flatter and potentially more manageable hierarchy, reducing inheritance-related complexity.

When evaluating cohesion using the Lack of Cohesion of Methods (LCOM) metric, Version 1's SudokuGame class has a score of 4, while Version 2 maintains this value for SudokuGame but shows an overall improvement with a total LCOM of 4 and an average of 0.8. This reflects better cohesion across the system in Version 2, implying that methods within the same class are more related, which typically enhances class understandability and maintainability.

Finally, Response for a Class (RFC) has increased in Version 2, with SudokuGame's RFC rising from 34 to 41. The total RFC for Version 2 is 87, averaging 17.4. This increase indicates that more methods can be potentially invoked, which could signal more complex interactions within the system, potentially making Version 2 harder to manage and debug. In summary, while Version 2 shows better cohesion and a more balanced inheritance structure, its increased coupling and RFC suggest heightened complexity in class interactions.

5.1.2 Additional Metrics and Evaluation on the Code Quality

In this subsection, we will review the metrics from section 4.1,Figure 9, in more detail. Additionally, what these values suggest about the overall code quality of the application will be provided.

The first version of the application shows that the SudokuGame class is relatively large, with a high number of lines of code (LOC) and cyclomatic complexity (CS), as seen in Figure 9 of section 4.1. This suggests that the class is handling many responsibilities, making it difficult to maintain and potentially violating the Single Responsibility Principle (SRP). The class has a reasonable number of methods, but the high LOC and complexity indicate that these methods are lengthy and potentially hard to understand. This could result in challenges for debugging, testing, and future modifications. The concentration of logic within the SudokuGame class points to a "God Class" issue, where too many responsibilities are held by a single class. Refactoring to split the logic into smaller, more focused classes could have improved the structure and maintainability of the code.

In contrast, the second version presents a more modular structure, where responsibilities are distributed across multiple classes. This is reflected in the lower total LOC and cyclomatic complexity compared to the first version, as shown in Figure 10 of section 4.1. The SudokuGame class, while still central, has been simplified with fewer methods and reduced LOC, indicating better management of responsibilities. Despite the improvements, the SudokuBoard class emerges as the most complex, with the highest cyclomatic complexity and method count, which suggests that further refactoring may be needed to reduce its complexity.

While other classes like SudokuSolver and SudokuValidator remain simple with lower LOC and cyclomatic complexity, the overall structure of version 2 is more modular, making the code easier to maintain and scale. However, some classes, such as SudokuBoard and SudokuController, still carry higher complexity, indicating opportunities for further optimization and refactoring.

In conclusion, the second version represents a clear improvement in code quality over the first. The more even distribution of responsibilities across classes, reduced overall LOC, and lower cyclomatic complexity make the second version more maintainable and in better adherence to SOLID principles. Nevertheless, some classes still show signs of complexity that could benefit from further simplification and reorganization.

5.2 Bug Fixing

As mentioned, there have been serious problems during code generation, as it did in the previous paper. The first version of the game, as it came from the AI, worked without any intervention, although it was not by any means playable nor functional [3]. As to the second version, there were some issues running the first-ever generated code; therefore, we had some intervention. First of the problems we had was that the button for a new game was not working properly: it was emptying all the cells from the board in every click, eventually leading to a fully empty board. Others were the non-randomized board problem and error notification problem. Both are fixed in the respective outputs. In addition, two last modifications were done, one was to grant the player the ability to erase what they may have written on the board, and also the functionality to solve the puzzle immediately. The bot successfully implemented these modifications, and came up with a working and playable program.

5.3 Testing

As in the previous paper, we have also conducted our testing manually. One can think of a first test such that we can try filling cells with numbers violating any of the cube, row or column rules. Thankfully, both versions of the game achieved success in this test.

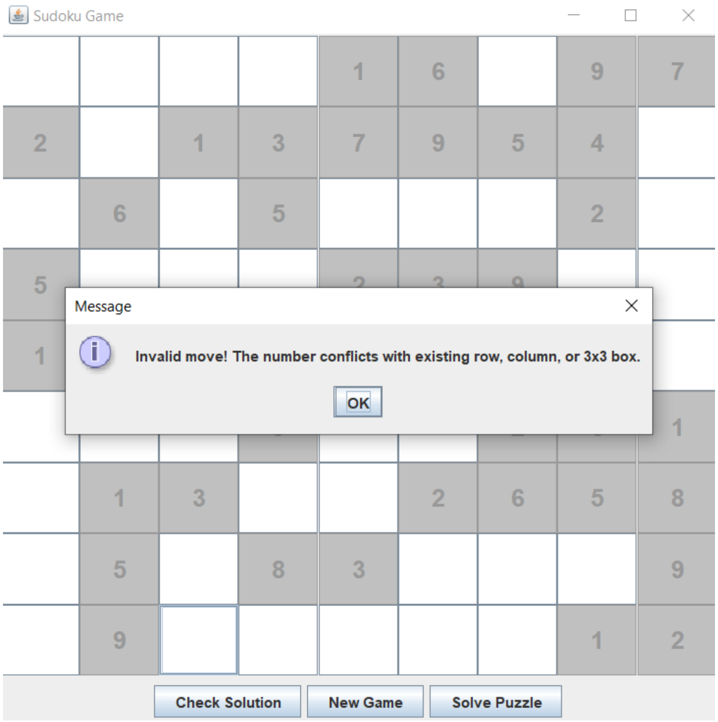


Figure 13: Error message shown when the player fills bottom-left cell with 9, violating 3x3 rule.

Moreover, all three buttons are also tested; “Check Solution” button effectively validates if all the cells are filled without violations; “New Game” button, every time it is clicked, generates a new randomized board; and “Solve Puzzle” button immediately finds a suitable solution for the puzzle. In summary, Version 2 of the game seems to have passed our manual tests, as we did not encounter any bugs or unexpected behavior during the course of execution.

6. Discussion

The evaluation's main conclusions are outlined in this section, along with a discussion of how the two Sudoku versions differ in terms of effort and prompts and how that affected the quality of the code. Moreover, in the following subsection, we compare this study with the prior to see how using ChatGPT-4o changed the results.

6.1 Summary of Key Findings

The key findings from the evaluation of both versions of the Sudoku game reveal notable differences in complexity, modularity, and functionality. The first version had a single large class with limited functionality, including non-functional buttons and a hard-coded board, resulting in a static user experience. It was less interactive and difficult to maintain due to its monolithic structure. The second version showed improvements with five distinct classes, providing better organization and modularity. Functional buttons for solving, checking, and generating new puzzles enhanced interactivity. However, this version also introduced increased complexity, with higher coupling between classes, as indicated by the code metrics. Despite the added complexity, the second version was more maintainable and scalable, passing manual tests successfully with working features like mistake correction and a solver function. Although some classes showed higher complexity, the second version offered a more complete and playable game with greater code quality and functionality compared to the first.

6.2 Comparison of Results of Both Studies

The previous study, as well as this one examined the chatbot's capacity to produce functional code with minimal user input, though the versions of ChatGPT and the prompting techniques differ. In the GPT-3.5 study, simple prompts were used to see how well the AI could independently create a Sudoku game. The results, however, were mixed: while GPT-3.5 managed to produce some code, the first version resulted in a fully solved, non-interactive grid, and the second version was playable but had numerous bugs and incomplete code, requiring significant human intervention. Although the prompts were kept the same in both studies to control, GPT-4o performed better overall, producing more modular and functional code, even though it still encountered bugs, particularly in the first version. The second version created by GPT-4o was fully interactive, with functional buttons for generating new puzzles and checking solutions. The GPT-4o code was more organized and modular compared to GPT-3.5, which often produced monolithic structures. While GPT-4o demonstrated better handling of complexity and interactivity, both versions required user intervention to fix bugs and ensure functionality. Overall, GPT-4o showed superior performance in generating more complete and maintainable code, though neither version escaped the need for debugging and refinement.

6.2 Implications

The implications of this study highlight the importance of modularity and structured code in software development. Version 2, with its improved class distribution and functional enhancements, demonstrates that dividing responsibilities across multiple classes leads to better maintainability and scalability. However, this comes at the cost of increased complexity, which can affect future modifications and debugging. These findings suggest that while modularity enhances functionality, it must be balanced with careful management of class interactions to avoid excessive coupling. Developers should focus on refining complex classes and optimizing code structure for easier long-term maintenance.

7 Conclusion

In conclusion, this case study highlights the potential of AI chatbots like ChatGPT in developing functional software, solely using the code generated by the tool. Authors successfully created a working Sudoku application by assembling generated code snippets with minimal understanding of the underlying logic. However, challenges such as missing implementations and bugs were prevalent, and the overall code quality was lacking. While ChatGPT can address minor bugs in isolated code snippets, it struggles with complex applications involving multiple classes, necessitating user intervention.

For example, developing a chess game, which requires sophisticated algorithms for move generation, game state evaluation, and opponent strategies, would likely present even greater challenges for ChatGPT due to the complexity of the game's rules and logic. The well-defined nature of Sudoku contributed to the study's success, but many software projects have unique, undefined requirements that may pose greater challenges for ChatGPT.

Given the limitations of time and scope in this software engineering project, further research with diverse and complex projects is needed to better understand the capabilities and limitations of chatbots. This project only aims to analyze the differences between two different versions by using ChatGPT-4.

References

[1] OpenAI. “”Introducing ChatGPT”. In: Introducing ChatGPT - Blog (2022). Accessed 20-08-2023. URL: https://openai.com/blog/chatgpt#OpenAI.

[2] 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.

[3] Filip Nikolic. “Chat Bots in Software Engineering: A Case Study using ChatGPT”, Bachelor’s Thesis. In (December 2023). Accessed 05-09-2024. URL: https://github.com/F-Nikolic/Bachelor_Thesis/tree/main