DESIGN MANUAL

Team 1

Introduction

In this final project for CSCI 205 at Bucknell University, we created a GUI-based version of the popular game Mastermind. In the original version of this game, the user has to correctly make a four-colored peg sequence guess in 12 turns. We upgraded this game to include three difficulty levels - easy, medium, and master - and a custom level - the user can select the number of pegs in the secret code and number of guesses. Additionally, we created a sound on/off mode for the user to choose from. Finally, we integrated GIFs to animate the winning/ losing messages and added three themes: original, zen, pink, and wooden.

User stories

The user stories we have created for the project are shown in table 1 below. These stories are based on our conversations with the client (the professor) and our discussion as a team. Our priority is to deliver a smooth user experience rather than to add complex functionality, so we did not start the multiplayer mode feature but rather complete all the tasks and updates to better the experience. Some features we finished are not listed here, for example, the mark to indicate the current row, because as we tested the program, we came up with more ideas that are translated into tasks right away instead of user stories. As we developed our program, we have consistently tested and conducted user interviews to ensure the quality of the experience. Some additional features suggested by the users are written in table 2.

Table 1: User stories for the project with the status

User stories	**Tasks**	**Status**
As a user, I want to	start the game	Done
	visually see the board	Done
	choose the pegs by dragging/ clicking on them	Done
	submit my guesses for feedback	Done
	see the feedback of the guess	Done
	find out if I win or lose	Done
	change my answers before I submit	Done
	choose to play again	Done
	quit the game in the middle	Done
	enter the name at the start of the game	Done
	know the rules of the games	Done
	have congratulatory sounds/ boo if lose	Done
	change the board to accommodate people's vision	Done

	customize themes for games (big ideas)	Done
	have a hint when stuck	Done
	interpret the results (question mark symbols next to row) customize to each result	Done
	have confetti to celebrate when I win	Done
	change the maximum numbers of guess - changing difficulty	Done
	be able to go back to the previous screen	Done
	play with my friends/ against a computer	Not started

Table 2: User stories gathered from user interviews

User stories	**Tasks**
As a user, I want to	play the game using the keyboard
	take notes or able to mark if I think the peg is in the correct position
	have the peg be button-like when we press it
	have hints about the position of the peg
	have suggestions for the guesses
	have a step-by-step tutorial for beginners

Design

In order to identify the most important abstractions and high-level classes that will become part of our Mastermind game design, we created an initial set of CRC cards (Figure 1 shows some examples):

- CRC cards for backend logic of game
    - CodeMaker class to handle generating and comparing secret code
    - CodeBreaker class to handle acquiring code and sending feedback after comparison
    - Board class to control interactions with the user
    - Code class to represent user entered guess and secret code
    - Response class to represent feedback after comparison
    - Manager class to control interactions between different classes and game flow

(A)

Code	
Makeup of 4 pegs of 6 colors	Peg

Response	
Makeup of 4 pegs of 2 colors	Peg

(B)

Manager	
Responsibility	**Return type**
Start the game	
Welcome message to get name	Player, Board
Get user code	Player
Send the user code to code maker	CodeMaker/ Response
Display Response	Player
Restart/ End a game	
Win lose message display	Board

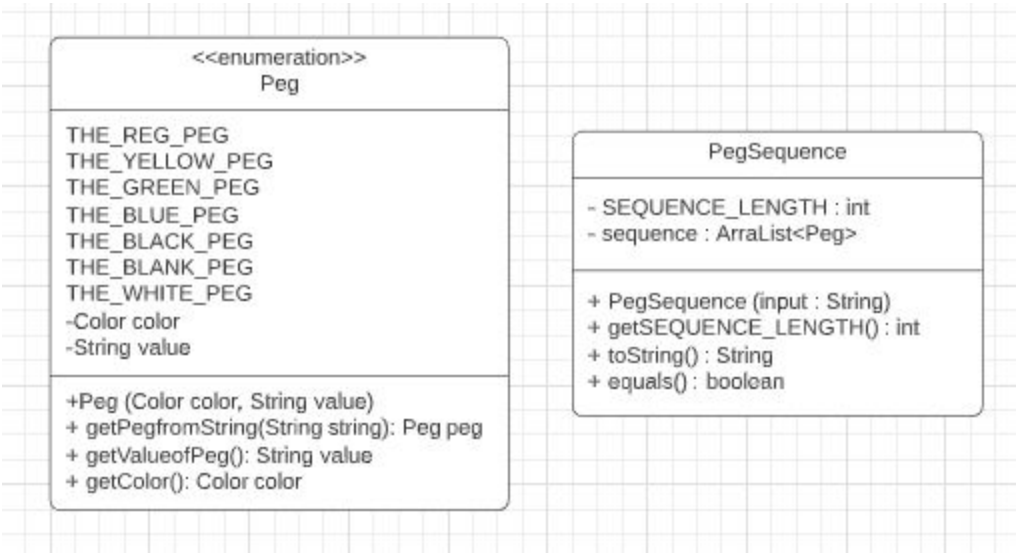
(C)

Board	
Responsibility	**Return type**
Display the code, peg, response	Manager
Get user input for name and guesses	Manager
Display result	Manager
Ask the user for again, quit	Manager

Figure 1: Some example CRC cards **(A)** Code and Response cards **(B)** Manager card **(C)** Board card

After reflecting on this design, we saw that there were many places for improvement. Firstly, we realized we could simplify the relationships between the code and response class. Thus, we created Peg enum to represent each component of the response and code sequences. Then, we created a PegSequence class to represent a sequence of Pegs, and then let the game use PegSequence objects for the purposes of representing Code and Response. This made more sense because Code and Response objects are both sequences of Pegs just used for different purposes in the game, and thus do not justify two different classes. Figure 2 shows some of our early UML class designs.

(A)



(B)

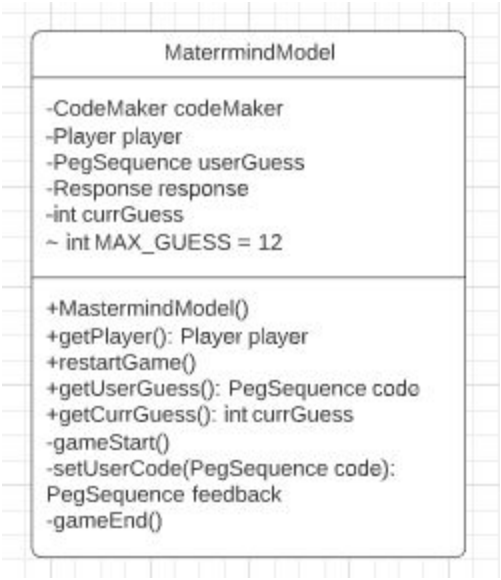


Figure 2: Early UML class design of **(A)** The Peg enum and the PegSequence class that will represent the code and response functionalities **(B)** The MastermindModel class that will handle the game logic. One thing to note is that we iterated on this design even after this. For example, in **(B)** we have the getUserGuess method, which we moved to the controller and kept the MastermindMain class only to handle the backend game logic

Since this was only the first stage of planning class design, the team was working in two sub-groups: Minh Anh and Anurag were creating CRC cards and Cuong and Lily were creating GUI interface wireframe using Figma. In order to integrate the backend logic and GUI interface, the team decided to use the MVC design pattern, which caused the backend logic to change significantly. For example, the new Manager class did not handle user input and display to the visual interface, but only controlled the game logic. The Board class was divided into multiple classes and placed in a view package. Finally, multiple controllers were created to coordinate the model and view classes. Some of these changes are shown in Figure 3.

(A)

Controller	
constructor take 3 view and the model	Constructor(3 view, model)
submit answer on click, check the curr number of guess, call set user code method from the model output the feedback the view should render it	clickPeg(): setOnAction
Handle the change back and forth between screen	theView.updatePegColors()
Get the input from the view translate that to the code understand by the model	
Get the output from the model and translate that to the view	
Delete button to delete entire row	
Quit call the game end method and quit java fx	
1st thing implement: click on the pegs, display the pegs that are clicked: updateViewAfterClicked(); checkAnsweronClick() updateFeedbackView(),	
DeleteOnePeg(), RuleScene()	

(B)

Figure 3: (A) An initial CRC card for the controller **(B)** UML diagram for the MastermindView class after the MVC design pattern was used. Note that this view class took on the responsibility of the Board class from one of the original iterations on CRC cards

Linking the GUI objects and the logic for the game was arguably the most important and challenging part of the project. Thus, we decided to make this our focus for the first two sprints. We had to use some creativity in this part of the project. For example, since the GUI required the color of Pegs but the model could not easily compare colors to create a response, we changed our Peg enum design so that each Peg has a color and a number value. The GUI would use the color and the model would use the number representation of the Peg.

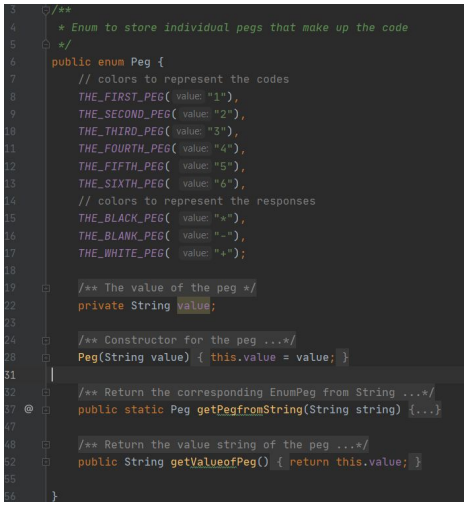


Figure 4: IntelliJ screenshot of the Peg enum showing how each Peg is represented as a number and a color.

Hereon, the user stories set by our Product Owner Lily primarily guided our class design. The second sprint was used to tackle the most important user stories that allowed the game player to start the game, see the board with the pegs, and be able to choose and delete the pegs to create a guess. In order to accomplish this, we primarily worked on the controller class design. We created the numerous buttons (Figure 5) in the view class and then designed the handlers for these buttons in the controller class. We also included extensive measures to not let users break the program. For example, the handleCheckAnswer() and handleDelete() methods ensure that the users can only check a complete answer (when four pegs are selected) and cannot delete an already submitted answer. In order to achieve this, we had to increase the coupling between the view and controller classes. For example, we had to create getRow() and getCol(int row) methods in the controller so that we could find which row and column was the user currently on while interacting with the GUI. With such changes, we were able to get a basic game of Mastermind running from the start to the end.

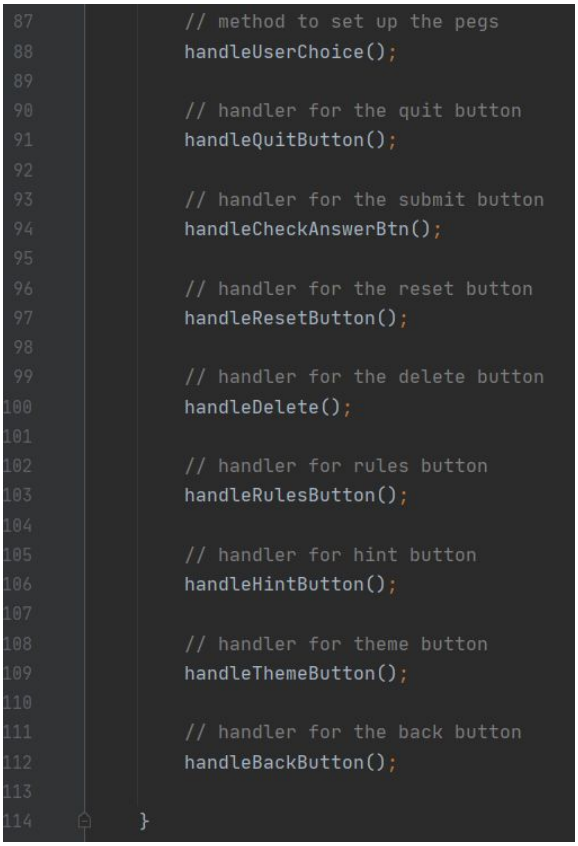
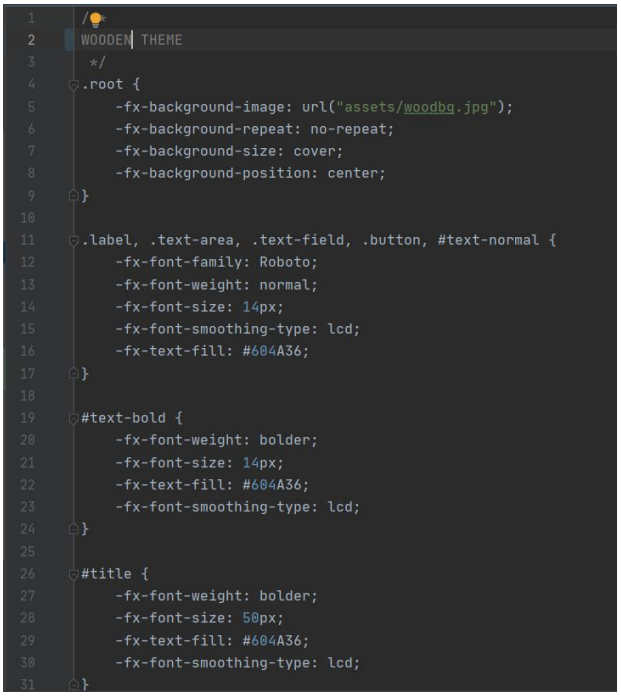


Figure 5: IntelliJ screenshots of the constructor of the controller class

In the next sprint, the major user story the team decided to accomplish was different difficulty levels: easy, medium, and hard. In order to create a screen with different modes, we chose to create different controller and view classes. This ensured that we maximized the cohesion within our controller and view classes, as each controller and view class had a specific role. While creating the different modes, we learned a very important lesson: the importance of not using magic numbers. In the initial design, even though we knew we had to create different modes later on, we used magic numbers to represent the

maximum number of guesses and other important values. Because of this, we ran into a slew of bugs as we created modes with different number of guesses and number of pegs. Thus, significant portions of sprint 3 and 4 were spent on replacing magic numbers with constants.

Another important design choice was the use of CSS sheets (Figure 6). This allowed us to gain significant control over the design of buttons, backgrounds, fonts, and colors. While this could have been done using JavaFX functions, CSS was chosen because we knew that a future user story was to create different themes. If we went the JavaFX route, then creating different themes would cause us to repeat large portions of code with just changes to how they were being shown in the GUI. However, by the use of CSS sheets, we were able to create an underlying game logic that was not dependent on the theme, which was controlled by different CSS sheets. This further decreased with coupling and increased the cohesion of our design.



```
/*
WOODEN THEME
 */
.root {
    -fx-background-image: url("assets/woodbg.jpg");
    -fx-background-repeat: no-repeat;
    -fx-background-size: cover;
    -fx-background-position: center;
}

.label, .text-area, .text-field, .button, #text-normal {
    -fx-font-family: Roboto;
    -fx-font-weight: normal;
    -fx-font-size: 14px;
    -fx-font-smoothing-type: lcd;
    -fx-text-fill: #604A36;
}

#text-bold {
    -fx-font-weight: bolder;
    -fx-font-size: 14px;
    -fx-text-fill: #604A36;
    -fx-font-smoothing-type: lcd;
}

#title {
    -fx-font-weight: bolder;
    -fx-font-size: 50px;
    -fx-text-fill: #604A36;
    -fx-font-smoothing-type: lcd;
}
```

Figure 6: Example CSS sheet for the wooden theme that we created for the Mastermind game

Once most of the user stories and product backlog was met, we set out four days for everyone to try and break the game. This allowed us to find numerous bugs in our code that continue to be a result of magic numbers. Once we fixed all of the bugs we were able to locate, we used the next couple of days to refactor and implement clean code practices. For example, only a few of our functions are more than 40-50 lines of code. Our variables are named such to minimize the use of comments. All of our methods are descriptive javadoc and if need be, we can also create an API for our game. Figure 7 shows one of our final UML diagrams to represent how the Main class handles all the coordination between sub-packages to make the game flow.

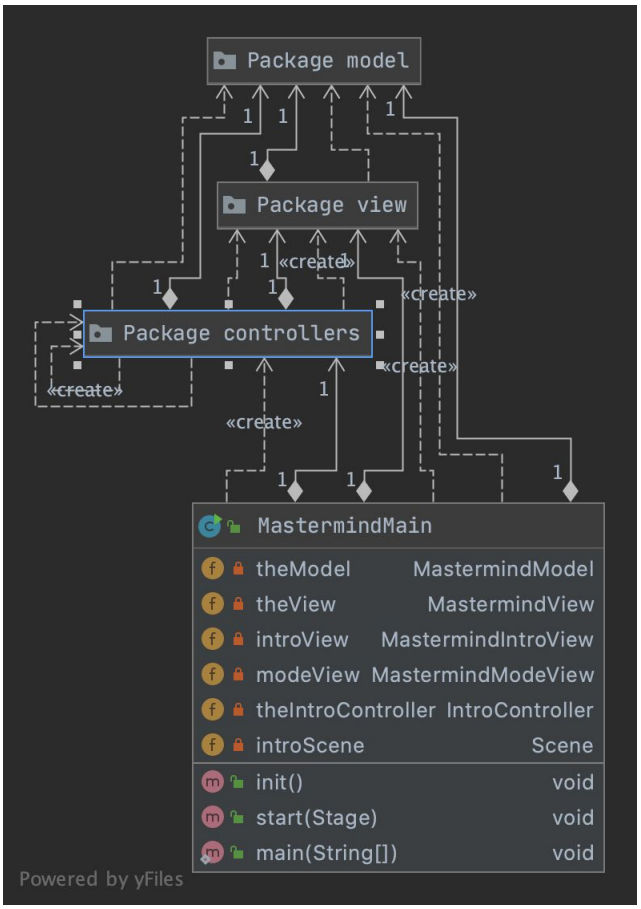


Figure 7: UML diagram showing how the MastermindMain class handles the coordination between the controllers, model, and view packages to make the game flow