Candy Land - Design Manual

Developed by Alexa Horvath, Claire Engel, Viveka Kurup, and Julia Calderone

Introduction

To create our Candy Land game, we used JavaFX to create the board and visual aspect of the game and a combination of Java classes to create the background functionality. We created a SceneController, which has the ability to switch between game screens, show the card picked by the user, and move the pawns around the board. We created a class that drives the gameplay internally, which uses multiple other classes to represent the different parts of the game, including cards, players, pawns, and spaces. These classes relate to each other in the way that the parts of the game interact with each other in real life, for example, players only have access to cards through the deck.

User Stories

We created user stories to drive the design of our game. The focus of these stories was the user interface, and making sure the game is easy to play and visually appealing.

- User Story: As Benjamin Williamson the Playful Child, I want an easy, colorful game to play with friends so that we can all be entertained. Persona Tale: "I love playing board games and wish they were more accessible online!" Benjamin is an eleven year old who really enjoys playing games of all kinds. He just got a new computer and wants more fun games.
- User Story: As Marijana Leclercq the Mother, I want a simple kid friendly game so that my children can stay engaged and have fun. Persona Tale: "I need an easy to play

multiplayer game to distract my children and keep them entertained" Marijana is a 34 year old single mother of 2 young children (4 and 7): Timothy and Jessabel. She needs an easy distraction for her kids while she completes housework and cooks meals. She wants a kid friendly game for all ages and genders, so that she can enjoy playing as well.

- User Story: As Astrid Mortensen the Twitch Streamer, I want advanced visuals and unique features so that the viewers enjoy watching. Persona Tale: "I love creating a streaming platform to introduce gaming to newer generations." Astrid is a 16 year old online gamer. Her platform is to entertain and educate primarily kids and preteens with appropriate and fun video games. She often plays with her best friend Rick, so they are looking for new multiplayer games. Although she has a younger audience, she prefers less childish looking games.

Object-Oriented Design

In order to create our object-oriented design for our candyland game, we began by reading the directions of the game to find potential classes and methods that we may need. By highlighting important nouns mentioned in the directions, we created a list of classes that we wanted to implement. During our actual implementation, we realized that we didn't need some of the classes that we initially thought we would, and we also needed other classes that we didn't think of before. From the initial list, we then created these CRC cards as the first step towards our object-oriented design. These are the cards that became actual working classes in our solution.

Players	
Responsibilities: - Has gingerbread pawn - Pick card - Move	Collaborators: - Spaces (what space the player is on) - Cards (what card they picked)

Spaces	
Responsibilities: - Track player on space - next/previous space	Collaborators: - Players (which player is on the space) - Special Spaces (subclass)

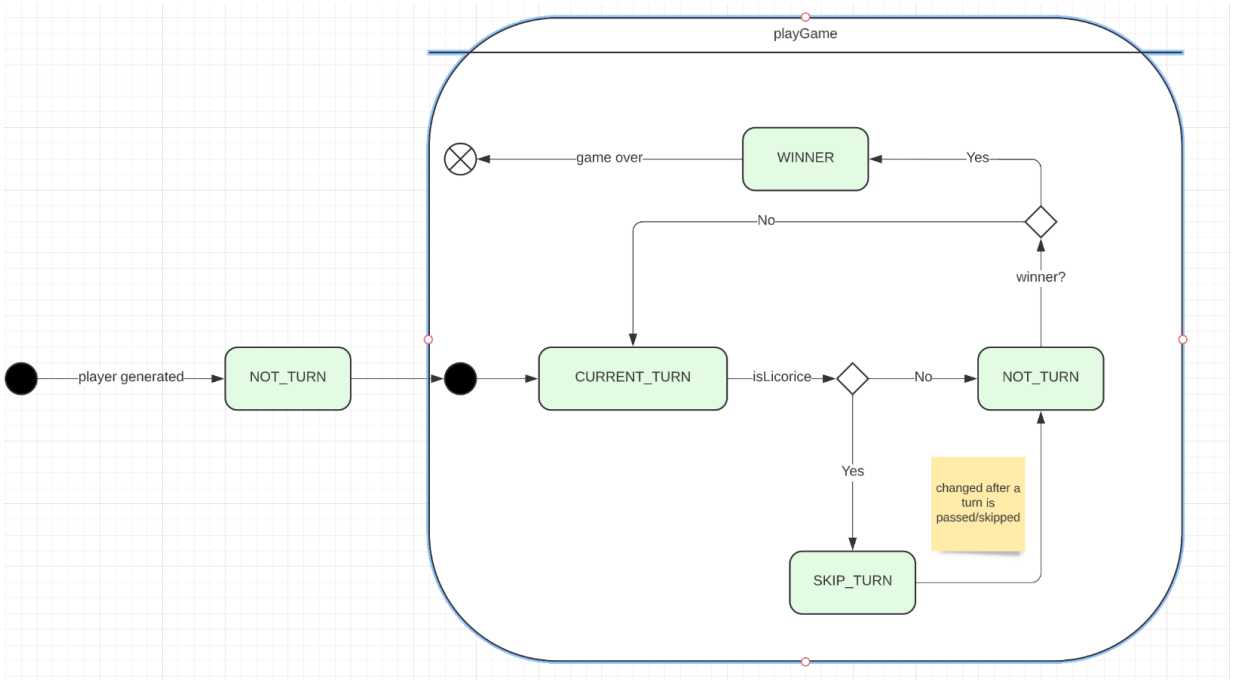
Card	
Responsibilities: - Generates cards that contain the color and number of spaces to be added to the deck	Collaborators: - Colors (attribute/enum) - CardDeck

CardDeck	
Responsibilities: - Creates a deck of cards - Allows a player to randomly pick a new card	Collaborators: - Card

Gingerbread Character Pawn	
Responsibilities: - Represent each player and show where they are in the game	Collaborators: - Player (which player is using the pawn)

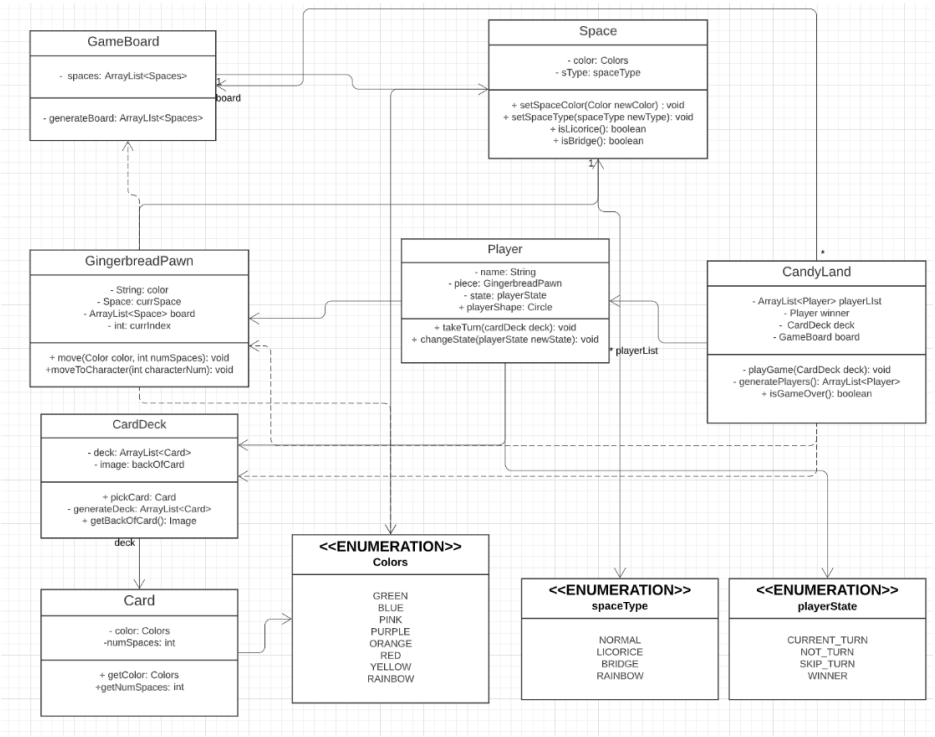
Game Board	
Responsibilities: - Which pawn is on which space - Tracks all of the spaces/special spaces	Collaborators: - Gingerbread pawn - Spaces - Special spaces

UML State Diagram:

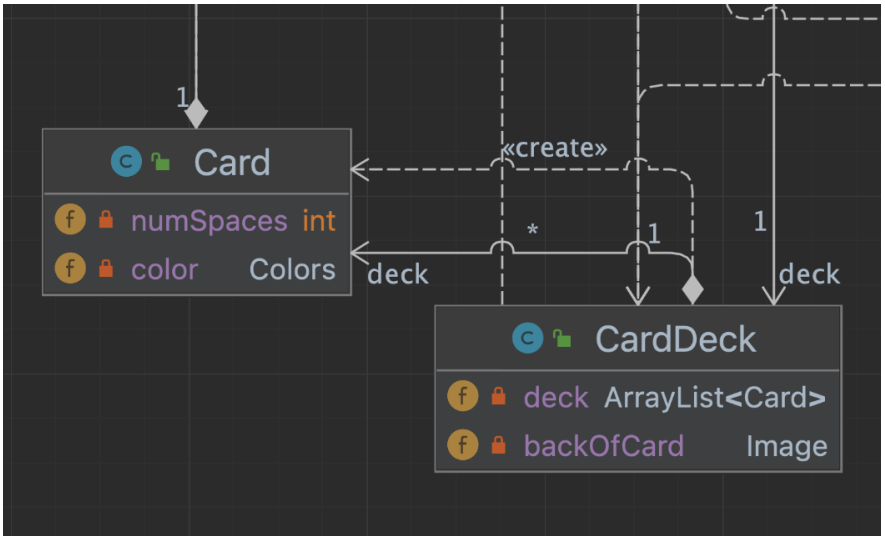


This UML state diagram shows the progression of the state of a player throughout the game. The player has one of four states: NOT_TURN, CURRENT_TURN, WINNER, or SKIP_TURN. These states ensure that the player can only pick a card and move their pawn when it is their turn in the player rotation. Additionally, they allow for players to miss a turn when their pawn is on a licorice space and continue gameplay after one turn is skipped. The SKIP_TURN state preserves the licorice obstacles from the original game, keeping our version as similar as possible. Finally, when a player wins, the change of the state of that player to WINNER ends the game. These states ensure that the gameplay continues and rotates through each of the players until one player wins, and continues accurately with the original game.

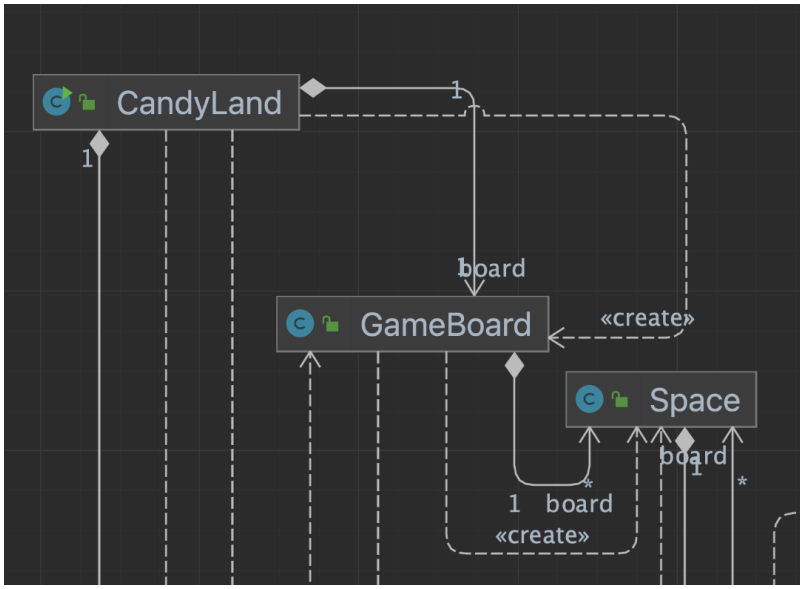
UML Class Diagram:



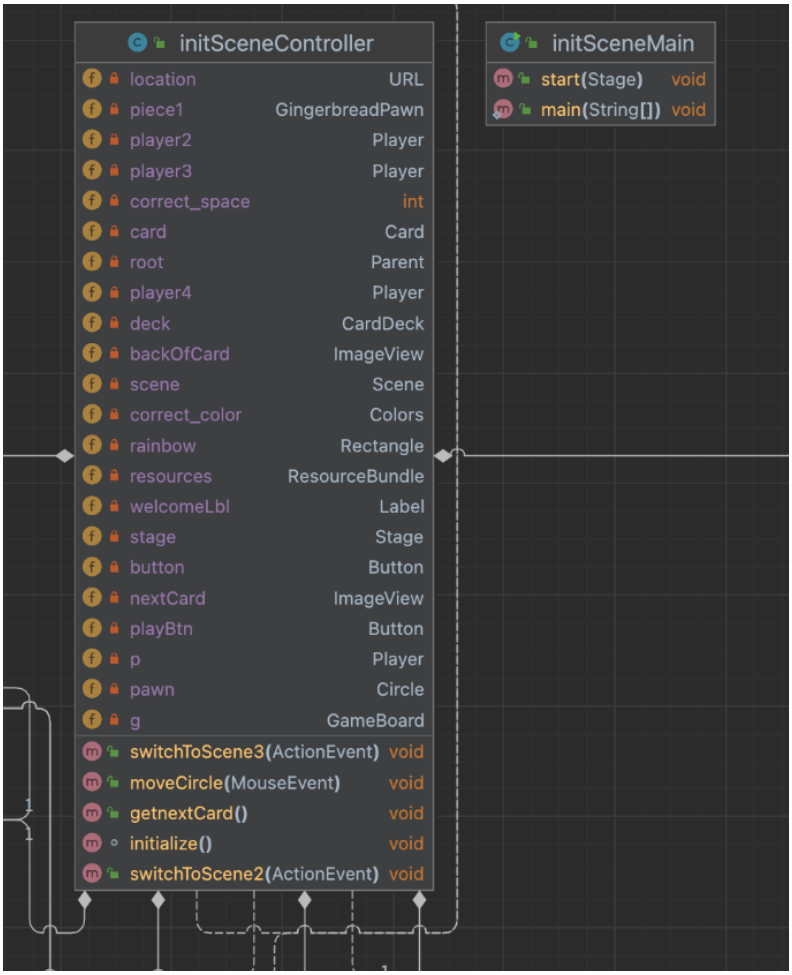
As we continued creating and updating our classes, we ended up with this as our final UML diagram. We used enumeration classes to keep track of the colors of the different spaces, the types of the different spaces, and the state of the player. The colors of the spaces are used in order to ensure the user moves the pawn to the correct space based on the card they drew. The space types are used when special actions are needed based on the space. Most spaces are normal, but when the pawn lands on a licorice or bridge space, they either skip a turn or move across the bridge. The rainbow space type allows us to determine the winner of the game, as when a player reaches the rainbow space, they win the game.



This is a screenshot from the UML diagram generated by IntelliJ. Our CardDeck class contains a list of Card objects representing the deck from which the player draws a card each turn. This class is the only class that creates Card objects, which simplifies the access to cards to go through the CardDeck class. Since players only draw Cards from the deck, the only access to Card objects needed is when they pick a card, which goes through CardDeck. Even though players do need to keep track of the card they picked, the Card is still accessed through the pickCard() method in the CardDeck class



This second screenshot from IntelliJ shows our CandyLand class. This is the class that we designed to run the game, and it is the only class that creates a GameBoard. We chose to do this because there is only one board needed for the game. This board is made up of Space objects, which are only created in the GameBoard class. Since there is only one GameBoard, and it is the only class that stores Space objects, they are only created in that class. This simplifies the creation of Space objects and the GameBoard object and limits access needed from other classes.



This screenshot shows our initSceneController and initSceneMain classes. These classes make the connection between the back-end classes and the front-end visuals that we created using JavaFX. The scene controller switches the scene, draws a card for the player, and allows

the player to move their pawn, ensuring that they move it to the correct spot based on the card that was generated.