MILESTONE 4

GROUP 7: Danae Morrison, Dylan Kim, Julien Ouellette and Sobechi Madueke

TABLE OF CONTENTS:

Game Design Documentation

Name: Sheepy Time
Duration: 30-45 minutes
Players: 1-4 players

Game Objective:
Be the first to reach your pillow using your winks (head) on the scoreboard by strategically jumping the fence, catching winks, avoiding nightmares, and utilizing dream tiles!

Milestone 4 Decisions and Deviations from Milestone 3 (How and Why)

General Note on MVC

To adhere to the MVC model for the game, we made sure that all of the game's classes besides the one responsible for initializing the game (Initializer) were separated into packages corresponding to models, viewers, and controllers. We did this to make it clear if any class communicated with another class that it should not be able to by the rules of MVC through the presence of an import statement from an inappropriate package. The classes in the controller package are allowed to import from the model and viewer classes, but classes in the model package should not import from the viewer and controller packages and classes in the viewer package should not import from the model and controller packages.

Initializing the Game and Shifting between Phases

Handling the Racing Phase

We thought it prudent to create the model class RacingPhase to store and make accessible the information required for the racing phase mechanics of the game. It makes use of other model classes such as Player, Nightmare, Deck, and DreamTileBoard and other variables that form the requirements of data to be used to enable the racing phase. In addition to the getter methods used to acquire state information, we also handled errors in the model classes. We did this so that it would be clear that the models are in charge of the logic of what is allowed or not allowed to happen in the controller class.

We decided to use RacingPhase to handle error messages, store data, and allow data to be retrieved, but not carry out tasks related to the racing phase itself because we wanted to separate the handling of the bigger actions carried out by the racing phase into classes themselves. These actions include playing cards (realized in CardPlayer) and using dream tiles (realized in DreamTilePlayer). To allow RacingPhase to make use of the CardPlayer class turned out to be a difficult thing to manage when the CardPlayer class required getting input from the user. Due to this, it became evident to us that we would feel more comfortable leaving the handling of playing a card through the CardPlayer class to the

RacingPhaseController class which can facilitate getting input from the user for the CardPlayer class without breaking the MVC structure.

Several viewers were used to bring the racing phase to life. These include RacingPhaseViewer and CardViewer. The two are separate because we wanted the complex functionality for printing the instructions of a card to be isolated from the rest of the print messages used to display information and obtain responses from the user. Within the RacingPhaseViewer class, methods were created to print off various messages to the user related to cards in a player's hand, the results of actions taken by players and the nightmare, and for asking for all input needed to be used in the flow of the racing phase.

Although some functionality is shared between aspects of the racing phase and the resting phase, primarily in the act of catching tokens, we separated the enactment of these aspects with the initial goal of keeping things distinct between the different phases. We prioritized getting the catchZ functionality implemented for the resting phase while we worked on other aspects of the racing phase. By the time we realized that some functionality could work better off as general classes that could be used by both phases as opposed to being catered to one phase over the other, time was tight. The final result is what we were able to come up with in the remaining time we had. If there is any aspect of the code that might not work totally correctly, it would be the catchZ option for the racing phase.

We are also a bit worried about whether the dream tile board is appropriately updated and sent across the different phases. *We were not able to test out the functionality of all the information that is shifted between the phases.

Playing Cards

As mentioned above, a good amount of the functionality of playing a card has been relegated to the RacingPhaseController for MVC purposes. For the sheep cards, there was user input and displaying information to worry about. The nightmare cards were easier to handle related to the MVC structure since there was no input to request- we just had to make sure that the game could output information to the user about the actions of the nightmare- i.e., a nightmare crossing the fence, players getting scared once and also getting scared awake.

Nightmare Card Functionality

We do not allow the nightmares to move backwards over the fence in the same way that the players are not allowed to move backwards over the fence. This means that, at the beginning of the racing phase, if a nightmare card is picked to move backwards by any amount of steps, the nightmare will remain off the board. The game is implemented only to allow two nightmares, Wolf and Bump in the Night, to be used in the game. This is because those two are what we felt comfortable implementing to allow for working gameplay.

Dream Tile Functionality

To implement Dream Tiles, we ended up using a DreamTile class as an abstract class with a few general methods, and then having each specific tile extending this DreamTile class to implement their own specific rules. We did it this way to solve OCP, as we can now add a DreamTile with whatever functionality we want simply by making a new class extending

DreamTile. Also, because of the complex behaviour the different tiles exhibit, it would've been hard to sort of "abstract away" all the different functionalities into variables that could be stored by all the tiles, so this interface turned out much better.

Discovered Code Smell and Noted Desired Changes

The reason why some code smells and other not quite so acceptable code still might exist in the code is not due to a lack of attention paid to identifying those smells, but a lack of time to address and correct them. Here's a list of detected problems that would be rectified had time permitted us to:

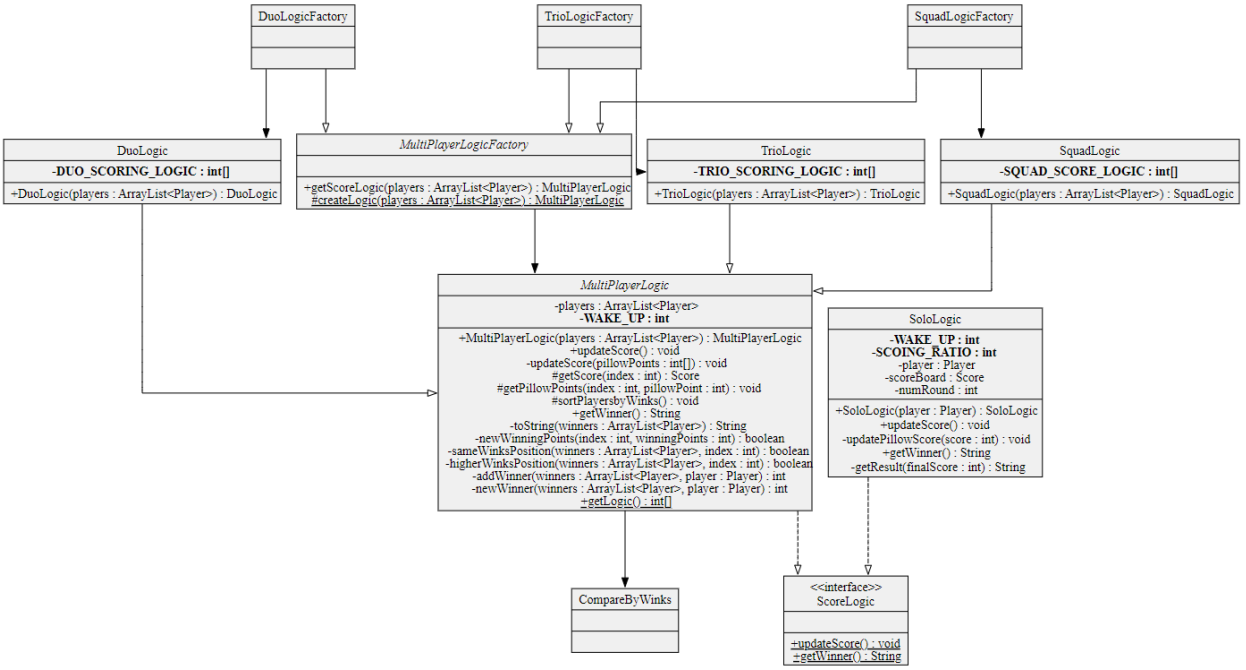
- DeckGenerator- For generating the sheep cards for two players, it is not necessary to fill a data structure to say that the cards are not nightmare cards and that they do not have any AND cards, so these booleans will always be false

Design Patterns

We made use of the builder and some factory design patterns throughout the code. The Card class uses the builder pattern because different kinds of cards make use of different values that can be attributed to a card- but never all of those values. The builder pattern allowed us to specify which parameters of a card were given values depending on which were relevant to that specific card while leaving the other parameters at default values that could be gleaned when the rules of a card are being printed or when a card is being played. Objects of the PlayerBoard and NightmareBoard classes are created with the use of the BoardFactory class, which determines what specific board should be created based on a string either saying "Player" or "Nightmare".

Subpackage: scorelogic &Factory Method Design Pattern

The score logic system(for Multiplayer, not solo) is closed for modification and open for extension to add new score logic, for example, FivePlayerLogic. However, this causes open for modification in the ScoreController class; If FivePlayerLogic is added, we need to open the ScoreController and add a new code, constructing the FivePlayerLogic object using the same factory system. We are aware of this problem and decided to prioritize resolving OCP in the factory system rather than in the Controller since the controller class is now considered as a client that is more reasonable to open for modification. The reason why the controller became the client, not the other model class, is that all ScoreLogic classes can return the winner of the game. Since there is a winner, this information would eventually have to go to the controller so that the controller passes it to the viewer, thus rather than having a middleman between the ScoreLogic object and controller, we decided to create the ScoreLogic object directly in the controller.



ScoreController

As we justified above, we successfully removed the if/else statement that violates OCP from the ScoreLogic through the factory method design pattern. However, that if/else statement hasn't gone entirely, it moved to ScoreController since we still want to get different objects depending on the number of players. We are aware that this is violating OCP because as we add new score logic, we wouldn't have opened the factory system, but the ScoreController class. We decided to prioritize resolving OCP in the factory system since ScoreController is considered a CLIENT from the perspective of the factory system. The reason that ScoreController became a client, not another model, is because the final result from the factory that we want is the String value of who is the winner or informing to proceed to RestingPhase if there is no winner. Since the String value needs to be printed only by the viewer, eventually the result would have to pass ScoreController. Therefore, we decided to let ScoreController become a client and directly associate with the factory to get rid of a MiddleMan code smell.

GameViolationException

Since this game requires a lot of user inputs, checking the validity of every single input is essential for the game. At first, IndexOutOfBoundException, NullPointerException, IllegalArgumentException and IllegalStateException were considered to be thrown, we were concerned that as a new functionality is added to the mode, a new Exception might have to be thrown, which requires Opening the Controller classes. We are aware of this problem and have decided to create a customized Exception called, GameViolationException, and extend RuntimeException; each Exception would extend GameViolationException. Even if there is already a pre-existing Exception that can be applied -for example, IllegalStateException might replace AlreadyOccupiedOnBoardException- we still created to resolve the

Open-Closed Principle. Abstracting all exceptions provides extra advantages when informing the user about the invalid inputs.

```
  throw new BoardIndexOutOfBoundsException(message:"Please type from 1 to 10!");
else if (!tileBoard.occupied(location)) {
  throw new EmptyBoardIndexException(message:"There is no Dream Tile on this location, please choose other location!"
else if (numZToken < 1 || numZToken > 2) {
  throw new IllegalZTokenAmountException(message:"You can put either 1 or 2 Z Tokens!");
```

```
} catch (GameLogicViolationException glve) {
    phaseViewer.showErrorMessage(glve.getMessage());
```

Those hard-coded messages will be accessible to the controller when it catches the exception and the controller simply passes that message to the viewer class to inform the user about the invalid input.

PhaseShiftController

In the PhaseShiftController, each RacingPhase and RestingPhase ends, their method(startPhase()) will return the DreamTileBoard object and this return object is passed to the alternate Phase as a parameter. Throughout the entire game, there is a couple of information that keeps updating and is used both in RestingPhase and RacingPhase, such as ArrayList<Player> and DreamTileBoard. Since ArrayList<Player> contains the actual reference of each Player object, it doesn't require to be passed and returned to preserve the information throughout the game. However, since DreamTileBoard is just a solid complex object, the update made during the RacingPhase would not be syncing to the DreamTileBoard object in RestingPhase. Thus, we decided to make the startPhase() method in each Phase required DreamTileBoard and return it back so that the information is successfully preserved.

RestingPhaseAction and RacingPhaseAction

Right now, RacingPhaseCatchZ and RestingPhaseCatchZ have a lot of codes in common. We are aware of this problem, so we considered extracting the behaviour and combining them into one CatchZ method that is applicable for both phases. However, unfortunately, we were lack of time, but also the overall structure and logic in RestingPhaseController and RacingPhaseController were too different for us to quickly extract.

UML Diagram & Sequence Diagrams

UML Class Diagram (use link in moodle for this)

Sequence Diagrams

Use Case Specifications

Use Cases

ID:	GameStart100Solo
Title:	Start Game of One Player
Description:	This use case has the goal of initiating the game for a user who is playing by themselves. This use case is always required in order to accomplish other goals related to playing the game as a solo player.
Primary Actor:	The potential game player
Preconditions:	The state of the game is empty. It's waiting on the would-be player to supply appropriate information.
Postconditions:	The system has initiated all the appropriate classes to correspond to the beginning of a game before the beginning of the first racing phase.
Main Success Scenario:	1. The game asks the user how many players there will be, between 1 and 4 2. The user responds with 1 3. The game asks the user to pick a difficulty from 1-3, 1 being the easiest and 3 the hardest 4. The user responds with the desired nightmare via a difficulty between 1 and 3 5. The system uses the information to create an appropriate deck, scoreboard, player objects, and other important details to prepare the game for a solo player entering the racing phase.

Extensions:	1. If a user enters a number of players that's not between 1-4, then they will be asked again to enter the number of players that will be playing 2. If a user enters an inappropriate response for nightmare type, they will be asked again to enter their desired nightmare
Frequency of Use:	Once, at the beginning of a game where there will be one player
Status:	Completed.
Owner:	Julien 1.0, Danae v 2.0
Priority:	Very important to the game's development

ID:	InitializingRacingPhase
Title:	Player Moves In First Racing Turn
Description:	The purpose of this use case is to fill a player's hand on the first turn of the racing phase and move the player on the board to finish the phase.
Primary Actor:	The players of the game
Preconditions:	The game has begun, and as of this round, the current player has not yet taken a turn.
Postconditions:	The racing phase is still in progress, meaning that at least one player is left (thus they will have two cards)

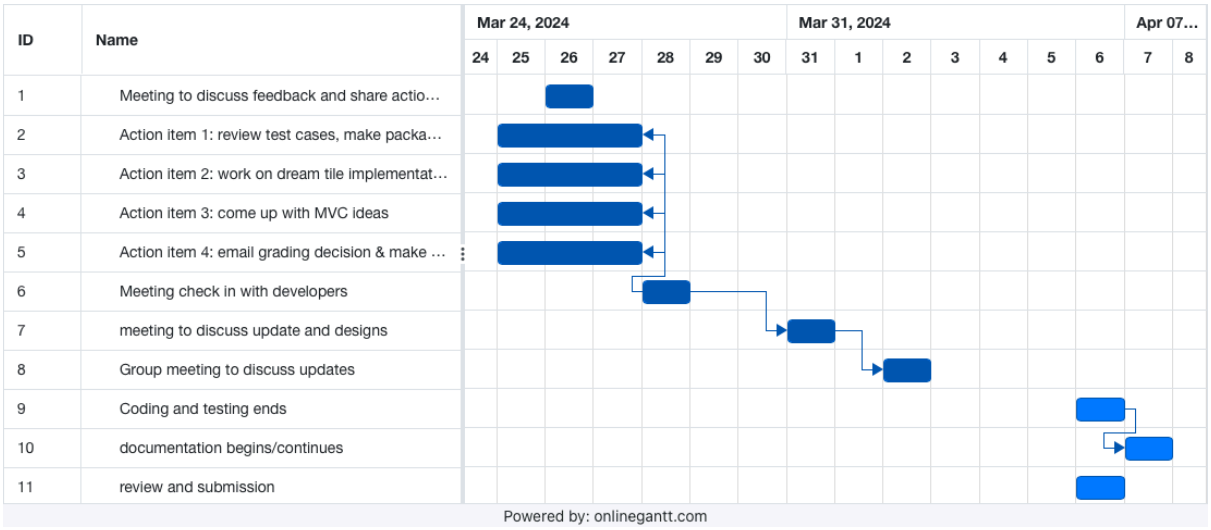
Main Success Scenario:	1. A sleep card is drawn by the game from the top of the deck, shown, and added to the player's hand. 2. After pulling a second sleep card from the top of the deck, the game adds it to the player's hand and shows it. 3. The player selects a card to play; 4. The user selects whether to move their character across the board. 5. The user is presented with the movement after the system executes the selected action and modifies the game's state. The card is placed in the pile of used cards. 6. A sleep card is selected by the algorithm from the top of the deck. After being shown to the player, it is placed in the user's hand. 7. The gamer's turn concludes.
Extensions:	1. Should the system draw a nightmare card from the deck, it will instantly execute the card's actions and add it to the used pile of cards. The user's hand will continue to be drawn cards by the algorithm until it contains two sleep cards. 2. The player experiences fear if they land on a position that has a nightmare on it.
Frequency of Use:	Once for each player at each racing phase of the game.
Status:	Developed
Owner:	Julien 1.0, Danae v 2.0
Priority:	Very essential to for the game to be played

Addressing feedback from milestone 3

- After receiving feedback we organised our code using packages.

- We fixed the code to adhere to MVC architecture by separating them into different classes (we used to have all in a method).
- Updated tests by correcting incorrect test applications and ensured it didn't call methods that don't exist.
- Provided Updated use case specifications

Gantt Chart



Deviations from schedule:

- The schedule only began after feedback was received.
- Meetings were held as scheduled, we all adhered to it, the only deviation was that we held a meeting on April 1st and not March 31st due to conflicting schedules.
- Documentation began as stated.
- Coding & testing went until the 6th.