Design Document

One-Player Sheepy Time Java Game

Group 4
Adil, Derrick, Tan, Tolga
Milestone 4
COMP 2711

Introduction

In our third milestone of our project, we have implemented a one-player version of the Sheepy Time Java board game. In Sheepy Time, each player is a Dream Sheep. Player will jump over the fence placed on the board (between the 10^{th} square and the 1^{st} square) to collect winks, avoid nightmares, and try to reach the maximum score he/she could reach to win the game.

Changes

Changes from last milestone has been documented at the end of this document under Updates section. And errors that were pointed out in our feedback are changed in the document.

Scheduling

We created our gantt chart, to start milestone 4. Our manager created the chart, assigned roles for each developer and tester. we decided to stick to the schedule as much as possible to finish refactoring in time. As a manager, I think we followed the schedule very well, and did a great job of communication.

One Player Version

Sheepy Time has two different versions. A One player version, which allows one player to play by themself, racing against the nightmare and to collect the most winks possible. Some differences between the One Player version from the Multiplayer version are:

- In the one-player game, after each card is played by the player and another card is drawn which results in the players hand containing two sheepy cards, the card on top of the deck reveals itself. If the card is a nightmare card, its resolved. If it's a sheepy card, it gets discarded. This makes the one-player game more challenging.
- In the one-player game, winning is different than the multiplayer version. Since you are racing against a nightmare, scoring more points means a more successful game for the player. Scoring rules are explained in the user journeys.

Project Design:

Our project prioritizes categorizing controllers, models, and views as we've been told. We created separate folders for each of them. When models, controllers, and views are interacting with each other, they must import the corresponding folder, this way we can clearly see which component is interacting with which component of the project. There were lots of changes to class structure compared to the last milestone, and we will try to explain all those changes below:

User Input

For dealing with user input, we decided to create a separate class. **UserInput** class is using a scanner to get user information. Different methods are included in user input, such as getting player name, getting sheep color, pick an option for a card, getting the selection of card in the players hand, basically has a separate method for each action that user should pick an option or enter some information for.

Controllers

Controllers are responsible for the translation from model to view, and vice versa. We removed all the game logic handling from our controllers. Instead, we designed our controllers in the way that they should be, getting user input, updating the model, and updating the view. My wording might not be correct, but by any means, our controllers are not handling game logic. For the fourth milestone, we changed two controllers from the original Milestone 3 submission. We decided to create following Controllers:

- **DeckController**: for controlling and handling the cards in the deck model and updating the view.
- **CardController**: is removed from our project since it is not being used in any aspect of the game.
- **GameBoardController**: for controlling the GameBoard model. Which helps changing game boards state,
- **GameController** was keeping all the controllers in the game as a instance. Main Control panel of the game. We refactored initializeGame method into Main class, to remove game controller from being the main control panel of the game.
- **NightmareController**: for controlling nightmare model and updating the nightmare view.
- **PlayerController**: for controlling player model and updating the player view, if the player model updates.
- **ScoreboardController**: for controlling scoreboard. And helps to update scoreboards aspects.
- **TileController**: for controlling tiles.

Design Patterns
We decided to implement multiple customized factories for creating objects like cards, tiles, nightmare, players, scoreboard. But from our feedback, we were informed that our factories weren't actual factories. So, for this milestone, we renamed the classes. The main idea of implementing creational design patterns was for helping us creating the cards and tiles. We wanted to incorporate couple design patterns, like Factory and Builder, for strengthening our program, and to follow the solid principles. Our main concern was Open-Closed Principle, and by Factory and Builder patterns, we avoided our concerns.

Here are all the patterns we implemented and their uses:

- **DeckBuilder**: forming the deck with creating cards.
- **NightmareFactory**: creates the nightmare depending on user input.
- **TileDeckBuilder**: forming the deck of tiles with creating tiles.

For the next section of Creational Classes, we named them as Creators. They are not like the design patterns we saw in our class. They are essentially for creating the respective controllers.

- **GameboardCreator**: creating the game board controller.
- **NightmareDeckCreator**: for creating nightmare cards and putting them into the deck.
- **ScoreBoardCreator**: creating scoreboard and the controller for it.
- **PlayerCreator**: creates the player object, and its controller.

Model

For our projects model, we needed more classes that we expected. Our initial start was trying to take our last milestones CRC cards, and make it work. Since our CRC Cards were not designed well enough, we encountered some problems. First off, we want to explain how we designed Cards and Tiles, since those parts are always fixed in our program, we created separate folders for cards and tiles. For the fourth milestone:

Cards

We structured our card design, by creating multiple card classes under the card interface for separating cards actions. So, for example, if a card's description says, "Move 5 spaces or catch 1zz", this card is created as **DualActionChoiceCard** class object. Here are all Card classes implementing **Card** interface, and how they work:

- **Card**: Card interface has three methods. One for getting description of the card, one for executing the action of the card, and one for checking if it's a nightmare

card or not, since all the sheepy cards and nightmare cards carry the same responsibility, we decided to construct our interface this way.
- **DualActionCard**: and combo card object for cards which has “and” in their description.
- **CatchZZZCard**: for card object which has “catch ZZ” in their description.
- **GainWinksCard**: for card object which gives you the option of gaining winks.
- **JumpNightmareCard**: specific card class which description of it lets nightmare to jump around the board.
- **MoveNightmareCard**: for the card type that description of it lets nightmare move around the board.
- **MoveSpacesCard**: for the card type that description of it lets the sheep to move around the board.
- **DualActionChoiceCard**: for the card type that description of it contains “or” word, which lets the player to choose an action.
- **ScareNightmareCard**: for the card type that description of it contains “scare” word, which allows the nightmare to scare adjacent sheep.

These are all the card classes we decided to design. By using the architecture, its easily accessible to check the type of the card. And we can differentiate which card type executes the action immediately, and which type of cards waits for user input for executing the chosen option. We created these Cards in a different folder, so only the parts of the program that must deal with the cards should import it and use it. Cards are getting created at the start of the game, and deck forms by keeping all the cards created by using these classes. Since it’s a one player game, we only programmed our design to create cards that have “2” number on them in the real board game.

Tiles

Tiles, are accessible for player to place it on the game board, during the resting phase, to activate their own unique boost. Resolving tiles, and their actions was a hard obstacle to deal with. Like what we did with the Cards, we decided to specifically create one class for each tile, since each tiles have different boosts in the game that can be activated. Of course, each tile class created in the tiles folder, will implement Tile interface. And we keep all the tiles in our Tile Deck class, to form some type of “tile market”, for our player to add a tile to the game in the resting phase. Since our game is a one player game, we didn’t need to incorporate all the tiles that are in the actual board game. As the rulebook informed us, we only needed to create these tiles below:
- **ActionHeroTile**
- **CoolKidsClubTile**
- **DoubleDutchTile**
- **FinalSprintTile**
- **IntenseDreamTile**
- **LoneSheepTile**
- **PerfectLandingTile**

- **RestingSpotTile**
- **SecondWindTile**
- **StepBackTile**

Each tile activates their own special boost, which is defined in the activateEffect() method that each tile object has.

For our updated version, we decided to add a **TileManager** class, for reducing the responsibilities of the **Gameboard**. **Gameboard** was managing the tiles on the board internally, but now with adding **TileManager** class, and extracting **Gameboard'**s methods which are updating and altering tiles to **TileManager**, got rid of some code smells. Responsibility of **TileManager** class is placing and retrieving tiles on the game board.

Now, since we figured out the ways to create and store cards and tiles, with their own effects, let's talk about rest of our model components, which form the game.

Nightmare

Nightmare is the villain of the game, since it's a one player game, we are playing against the nightmare only. Our idea of a nightmare is changed since the last milestone. Our initial **Nightmare** class was an interface, but since all the nightmares are sharing the same attributes and same methods, we decided to make **Nightmare** an abstract class. This way, we got rid of the duplicate code inside the **Wolf**, **Spider**, **BumpInTheNight** classes. Here are how nightmare classes are structured:

- **Nightmare**: contains three methods, for getting name of the nightmare, which is also the name of the class that nightmare concrete object is initialized. And two more methods for description and difficulty of the nightmare, which is for viewing to the user.
- **Wolf**: First level of difficulty, most likely to be chosen for a beginner game. Variation of the nightmare.
- **Spider**: second level of difficulty. Another variation of a nightmare
- **BumpInTheNight**: third level of difficulty.

By allowing our objects that are inheriting from the **Nightmare** to keep their name, difficulty, and description helps us with the view component of the program, for printing out information of the nightmare when it is chosen, and when it is encountered.

Deck

Deck structure is based on a queue of cards. In our previous milestone, we had the idea of two lists for keeping track of played cards, and active cards. But as its stated from the feedback response, we didn't need that since the deck doesn't need to be empty at any time, not like the real game version. So, we reduced our deck to keep

only one queue, which, by the help of **DeckController**, hands cards to the player. We still did keep the idea of shuffling the deck. Instead of popping and pushing the card back to the deck, we create the deck when all the cards from the queue is gone. We didn't want to push and pop, since we essentially focused on cards not coming back in the same order. Rest of the methods in the deck are for doing actions on the queue, adding card to the queue, checking if it is empty, or drawing the card.

Movable

Movable interface is for the objects on the game board to implement. Our **Gameboard** will only hold movable objects as instances. By the name you can tell, objects that implement the movable interface are allowed to move. **Movable** interface has 6 methods: getName() for getting the name of the Movable object; becomeScared() for scaredness update the scaredness level of a sheep; crossFence() for updating the winks of a player; isScared() to check if a sheep of a player is scared or not; becomeBrave() is for increasing the scaredness level of a sheep by one; and, lastly, wakeUp() for ending the racing phase.

MovableManager class was added with the new updated version. **MovableManager** class is purely designed for managing the **Movable** objects on the board. Instead of **Gameboard** class handling the Sheeps and Nightmares on the board, our **MovableManager** will deal with that with the next update. We created this class by extracting methods from **Gameboard** to **MovableManager**.

Player

Player class implements **Movable**. **Player** class is designed to hold the players hand, which is two sheepy cards. **Player** class has lots of responsibilities, from playing cards, gaining winks, moving pillow position, holding its own **Sheep** as a variable, and catching ZZZ's. We are aware putting this much responsibility is probably violating SRP as well, but as the core component of our game, **Player** needs to have these instance variables to hold. We decided to break some properties, because our **Player** class is technically the "database" of our game. All the information that the player holds are connected to each other, so even though we are breaking SRP, we are making sure that each variable that Player holds is essentially for the player and player only.

Logic

To handle the game logic parts of our game, we decided to have the **GameLogic** class. The **GameLogic** class is responsible for executing the game, which has the loop that the game is running on. To handle this big of a structure, we couldn't find any better way than having one class that deals with controllers. The GameLogic class contains four methods. playGame() is basically the while loop that the game keeps

following until the end of the game; playPhase() is the new method we added to our program, which takes in a Phase object parameter, and returns true if the game is over; checkWinCondition() method to use inside the playPhase() method, to communicate with scoreboardController to check if the player satisfied the winning conditions; and resetGame() method, is for resetting the gameboard and the scoreboard.

There are two more “Logic” classes. The first one, **RestingPhaseLogic**, which contains the actions that should take place in the Resting phase. Placing the tile, catching a zz on an existing tile, and placing the top tile on the game board, for one player game, as it is explained in the rulebook.

The second logic class, the **RacingPhaseLogic** class, which used to have one method for making a Racing move, now has 9 methods. We had one long method, to execute the racing move a player has played. Now with extracting methods, and fixing the long method, we removed the code smells. Our methods are essentially doing the exact same thing as the last method, but in parts. We combined all into playRacingMove() method.

These classes are created to specify the different moves in resting phase and racing phase.

Phase

With our new milestone, we added the **Phase** interface. The **Phase** interface, contains two methods, for playing a move for each phase. Also, we introduced the **RacingPhase** and the **RestingPhase** classes, which are implementing the **Phase** interface. Our idea for this interface and classes, is for reducing responsibility of the **GameLogic** class. Classes that implementing the **Phase** interface, are creating their respective logic class object inside their method, and calling the method that plays the move. For example, for the **RacingPhase** objects executeMove() method, **RacingPhaseLogic** object is created, and playRacingMove() method is called.

By creating this **Phase** interface, we reduced the responsibility of **GameLogic** class, and with this approach, we adhered Open-Closed principle with our phases.

Scoreboard

The **Scoreboard** class is designed to keep the list of players as instance. **Scoreboard** class has simple responsibilities, moving the pillow of a player, adding winks to a player’s wink score, waking a player up. **Scoreboard** as it is in the real game, is for keeping the pillow and wink token to check winning conditions.

GameBoard

The **Gameboard** class is completely refactored, after the addition of **MovableManager** and **TileManager** classes. We extracted the responsibilities to the Manager classes, and now our **Gameboard** is only keeping instances of the Manager classes. Each method in this class, are calling managers, to execute actions. As we described above, **MovableManager** class's responsibility is handling the movements of movable objects on the **Gameboard**, and **TileManager** class's responsibility is handling the tile actions like placing a tile, or activating a tile on the board.

Views

The overall View is text based. According to the technical specification for the project, each view displays out the corresponding model's information, which is passed by the corresponding controller. Here are views in our project:

- **CardView**
- **GameBoardView**
- **GameView**
- **NightmareView**
- **PlayerView**
- **ScoreBoardView**
- **TileView**

All these views are designed to print out the information of the object that is passed by the controller. In every round, we focused on printing out as much information as possible to keep the player engaged, and not to make them feel lost during the game. We print out a welcome message with the **GameView** when the game is starting, and rest of the views are called to print out information after each move is played.

From the last milestone, we missed a mistake in our views, which was importing models. With our feedback response, we removed the importing of the models. Our views are now purely independent from the models, and only taking primitive type parameters in their methods.

Testing

For testing program, we decided to create separate folders for each part of our project to test. Our testing folders include.

- Card Tests: testing the card interfaces methods. executeAction() Is the most important one.
- Controller Tests: testing if controllers are doing their jobs to help models update, and help views to print out information.
- Factory Tests: were removed since all the design patterns are renamed and moved into the model folder. Now the testing of design patterns are included in Model Tests folder.

- Model Tests: testing the model component of our project, like player, scoreboard, gameboard, nightmare, and every other logic our game should have. Testing the edge cases, to see if we will encounter with unexpected errors.
- Tile Tests: like card tests, testing all the tiles, and their action, to see if the action is correctly applied to the game.

For testing part of our view class, we included screenshots from our running program, to show what is expected from our program, and from our view classes. Each view classes responsibilities are listed below:

- **CardView**: should display a card information. What actions could be taken with the card should be printed out. (Move 5 or catch zzz)
- **GameBoardView**: should display game board state, and information about every object on the board with help of individual views. (tiles, movables, nightmares)
- **GameView**: should display welcome message when the game starts, also should display goodbye message when the game is over.
- **NightmareView**: should display current nightmares information on the board.
- **PlayerView**: should display players information (name, sheep, hand information, amount of zzzs)
- **ScoreBoardView**: should display the current state of the scoreboard (game state, player score, wink position, pillow position)
- **TileView**: should display the tile information.

Updates

In our previous submission, we had some problems with MVC architecture, testing, and a little bit of Solid violations. We approached our mistakes, by fixing the small mistakes we had. Most of the updates are updated in the document in respective places where the update has happened, but we added this little section to give a quick overview of what we changed. Here are some changes from the last milestone:

- Testing:
    - 3/178 tests were failing on MAC and Linux systems only. Those were the view tests, and we knew views might be challenging to test. Even though the expected output and actual outputs are identical, we were still getting errors on Linux and Mac systems, so we commented those out for you to inspect.
- Models:
    - **GameBoard** refactoring. We refactored **GameBoard** class, by creating a **TileManager** class, for reducing responsibilities in the **Gameboard's** methods. We basically extracted the methods of **GameBoard**, into **TileManager** classes methods, and created an instance of the **TileManager** inside **GameBoard** for our board to access all the methods.

After this success, we did exact same steps, to manage the movable objects on the gameboard. We extracted the **GameBoard** classes methods where the movable object is managed, into a new class named **MovableManager**.

- **Phase** interface. We created a phase interface, **RestingPhase** class, and a **RacingPhase** class, which is implementing that interface. The main idea for this is to help us with refactoring the **GameLogic** class, since it was acting like a god class and needed some extractions.
- **RacingPhaseLogic** class changes. We fixed long methods by extracting methods. We created 7 new methods, and we use these methods inside the playRacingMove method for better design. Now instead of 1 method in this class, we have 7 methods that will call each other inside playRacingMove method.
- **RestingPhaseLogic** class changes. We fixed magic numbers, also we fixed the long method by extracting the same way we did with the **RacingPhaseLogic**. Instead of having one large method for playing a resting move, we extracted some parts of the method into 6 different methods.
- **Nightmare** Interface to abstract class change. Since all the nightmares are sharing all the same variables, we decided to get rid of the interface, and make **Nightmare** class an abstract class. This way we remove the duplicated code inside the classes who inherits **Nightmare** class.
- After all these changes, we had small bugs for skipping over phases, so we fixed that by adding a loop inside the **GameLogic** class.
- From the feedback, we were informed that we should change the names of our Card classes. Using and, or for class names were bad naming, so we changed **AndComboCard** to **DualActionCard**, **OrComboCard** to **DualActionChoiceCard**

- Controllers:
  - We removed **CardController** class, and respective test class since we realized we are not using the controller in any part of our program.
  - **GameController** refactoring.
  - **GameController** and Main changes. Instead of having the GameController deal with all the “god” work, we transferred it into **Main** class. We removed initializeGame method from the **GameController**, and transferred everything that was done in that method to **Main** class. This way **GameController** class can now act as the actual Controller.
- Views:
  - All our views were importing the respective models, for communicating. This was a big mvc violation. We removed all the imports that views have, and made them completely independent. Our views are now only getting primitive type parameters, for printing out and updating information about the game, and none of the views are talking to the models anymore. We

did this by making controllers send the models information to the view, by using getters, instead of sending the whole model to the view.

- Design Patterns:
    - From our feedback, we were informed that none of our Factories were factories. So, we worked on fixing all the Factories and give them correct names. New names of our Factories are:
        - **DeckBuilder**: This class builds the deck by using different methods inside the class, and returns a finished deck, deck controller, and a deck view.
        - **NightmareFactory**: This class naming stood the same, but we extracted some lines. Before extraction, this class wasn't a factory, but now, it returns nightmare depending on the user input.
        - **TileDeckBuilder**: Same idea with DeckBuilder. These classes were using builder design patterns, but we named them as factories. We changed the naming.
    - Rest of our factories weren't really a design pattern we learned in class, they were more of an Initializer, or Creator classes to create the controllers of the game. So, we decided to rename those classes into "Creator" classes.
        - **GameBoardCreator**
        - **PlayerCreator**
        - **ScoreboardCreator**
    - After finishing updates, we moved our design patterns, and creators into model folder, to getting rid of factories folder. We also did the same thing for tests, and removed factory tests, since most of our designs are not factories anymore.

Conclusion

These are all the updates that happened in our program, to attempt eliminate code smells. MVC structure wasn't respected at first submission, but we eliminated some important mvc violations in this milestone. As we described every class and its effect on our project above, and our project still works successfully as expected, we can conclude our Design Document.

Use case specifications, Schedule and the Gantt chart, Feedback Response from the past milestone feedback, Sequence Diagrams and UML Class diagram is added below this document.

Feedback Response

From milestone 3 feedback, we had some successful parts, and some unsuccessful parts of our program design. These are the changes we will have in the next milestone submission:

- Game can run without crashing. Good Sign.
- UX can be reduced, but we decided to keep the wall of text, so this way user can
- Public instance variables are all changed back to private, it was an honest mistake that we missed, but now we fixed all the public variables.
- Static methods were not needed, and we will change all the static methods to normal methods.
- We fixed all the components that didn't pass the tests. Since we updated our program with removing code smells, we changed the test as well. What classes and what test are changed accordingly, is explained in detail in our design document.
- For the java doc part, we put all four of our names, since we gave each other ideas and talked a lot about how we can improve the program. Adil and Derrick are the developers, and they did most of the coding, but they wanted to put our Manager and Testers name for showing that us as a group were involved with all the aspects of the program and helped each other identifying and solving errors. We can change it back to only Adil and Derrick, but we decided to keep all the names since we believe all of us did more than their role jobs to finish this program.
- For controllers, our main reason to have that many controllers was to not violate mvc structure. Since controllers can talk to each other, we decided if we have as much controllers as we want, we can operate the communications between them.
- We removed the methods that were not using in the program. ExecuteCard Action method is removed.
- Our factories were not actually factories, so we fixed all the factories. Changed some of them to Composite, Builder, and Ones we wanted to keep as factories are fixed and made into actual factories.
- We mostly did our refactoring to fix mvc without breaking the game. Removed all the controller responsibilities inside the models.
- Removed all the view and model communications.
- Model classes are not importing views anymore.
- View classes are not importing models anymore.
- We refactored our game logic class into multiple classes, to reduce code smells.

Schedule

Milestone 4
Group 4

March 29

- Assigned Roles
- Created gantt chart. (Derrick)
- Read and talked about feedback, wrote a feedback response. (Derrick)
- Started on Fixing View and Model communications. (Tan)

March 30

- Had another meeting for refactoring our code smells.
- Changed GameLogic class for reducing the controller responsibilities that It has. (Tolga & Tan)
- All the view and Model communications are deleted. Views are not importing models anymore. (Tolga)
- Changed Testing methods, to make It appropriate with our new changes in our program. (Adil)

April 1

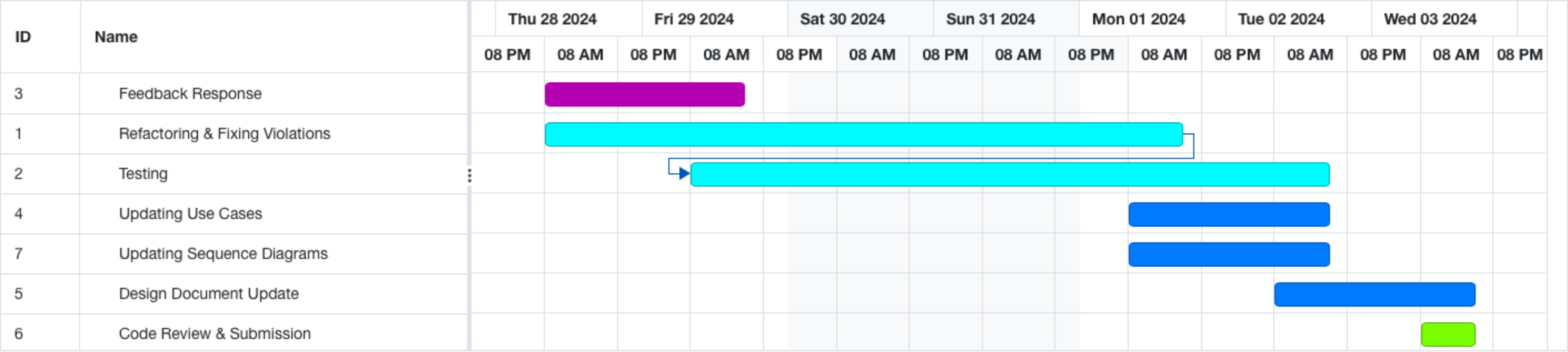
- Finished Refactoring. (Tan & Tolga)
- Started working on design document. Incorporated the changes from our program to the document. (Derrick & Adil)
- Updated Use Case Specifications, and Sequence Diagrams. (Derrick & Adil)
- Auto generated UML Diagram from the code provided.

April 2

- Code review. (All members)
- Final test runs are done. Tests are working perfectly.
- Took one last glance to check any Solid violations we break during refactoring.

April 3

- Submitted milestone.



Use Case Specifications: Sheepy Time

In this document, there are 8 high priority use case specifications for the sheepy time java game. JUnit testing for our program is going to be based on some of these use cases. All the use cases are for ONE PLAYER GAME only.

Use Case 1:

ID:	**1.0**
Title:	**Start the Game**
Description:	Initiates the Sheepy Time java board game.
Primary Actor:	Player
Preconditions:	User must have the required java files to execute the program. 1) Game is not running. 2) Game has not been initialized.
Postconditions:	The game has running and initialized. Ready for player to do the first move.
Main Success Scenario:	1. User starts the software. 2. User enters his/her name. 3. User enters the color of preference. 4. User chooses one of two nightmare options. 5. Game is set and ready for user to play.
Extensions:	3a. Input from user is not a color. 1. Ask user to enter a valid color. 4a. Input from user is not a nightmare. 1. List of nightmares prints on the user view. 2. User enters correct nightmare from the list.
Frequency of Use:	1 time use. Beginning of the game.
Status:	Done
Owner:	Tan
Priority:	High priority. Correct Game initialization is a must to play the game.

USE CASE 3:

ID:	**3.0**
Title:	**Play a Sheepy Card**
Description:	Player plays one out of two sheepy cards in their hand. Effect of the card applies immediately.
Primary Actor:	Player
Preconditions:	1) Game is running. 2) Game has been initialized. (See Use Case 1.0) 3) Sheepy Cards have been dealt. (See Use Case 2.0)
Postconditions:	The Player's intended sheepy card is played, and actions are resolved.
Main Success Scenario:	1. Player chooses one of two sheepy cards in their hand to play. 2. Card action is applied. Card gets discarded from player's hand. 3. A new card is added to player's hand. 4. Player reveals the top card of the deck.
Extensions:	2a. Card chosen by player is a multiple action card: 1. If it's an "or" card, player chooses one of two actions. 2. If it's an "and" card, both actions are applied. 3a & 4a. Card is a nightmare card: 1. Game resolves the nightmare card immediately. (See Use case 4.0) 2. Another card is added. 3b & 4b. Card is a sheepy card: 1. Continue the steps with the success scenario.
Frequency of Use:	Frequently, at least once in every round.
Status:	Done
Owner:	Tan
Priority:	High priority. Card resolution is one of the most important aspect of the game.

USE CASE 2:

ID:	**2.0**

Title:	**Deal the Cards**
Description:	Initializes the player's hand by giving the player cards until two sheepy cards form in their hand.
Primary Actor:	Software. (Deck Controller)
Preconditions:	1) Game is running. 2) Game has been initialized. (See Use Case 1.0)
Postconditions:	The Player has two Sheepy Cards in their hand. Ready to proceed with the game.
Main Success Scenario:	1. Top card of the deck is returned to the player's hand. 2. Player forms their hand with two sheepy cards.
Extensions:	1a. The top card of the deck is a nightmare card. 1. Game resolves the nightmare card immediately. (See Use Case 4.0) 2. Repeat 1 from Main Scenario. 1b. The top card of the deck is a sheepy card. 1. Repeat 1 from Main Scenario.
Frequency of Use:	1 time use. Beginning of the game.
Status:	Done
Owner:	Tan
Priority:	High priority.

USE CASE 4:

ID:	**4.0**
Title:	**Nightmare Card Resolution**
Description:	In any part of the game, if a card added to players hand by the controller, with the draw action, is a nightmare card, we need to resolve them immediately. This use case follows the resolution steps. Nightmare cards are different than sheepy cards, which are meant to move, jump, the nightmare, or scare the sheep's around the nightmare, based on the cards description.
Primary Actor:	Software
Preconditions:	1. Game is initialized. (Use Case 1.0) 2. Game is running.
Postconditions:	1. Nightmare card is resolved, action applied to the game. 2. If nightmare crossed the fence, phase ends.
Main Success Scenario:	1. Cards get dealt by the program (Use Case 2.0) 2. A nightmare card is revealed. 3. The description of the card (move, jump, scare), applies to the game immediately.

	4. Player is handed another card from the deck.
Extensions:	4a. Card handed from the deck is a nightmare card. 1. Repeat from 3. Again.
Frequency of Use:	Frequently, most likely to happen every other turn.
Status:	Done
Owner:	Tan
Priority:	High priority.

USE CASE 5 :

ID:	**5.0**
Title:	Gaining Winks
Description:	Winks are players progress on the scoreboard. Every time players sheep jumps over the fence, player gains 5 winks. Player has a wink and a pillow token, which is kept track on the scoreboard, whenever the players wink is past their pillow, game ends and solo scoring calculation starts.
Primary Actor:	Player
Preconditions:	1. Game is running and cards are dealt. 2. Players sheep is close to jumping the fence
Postconditions:	1. Sheep jumped the fence.
Main Success Scenario:	1. Player plays card. (Use case 3.0) 2. Players sheep is moved on the board depending on the cards information. 3. Sheep crosses the fence. 4. Player gains 5 winks. 5. Player chooses to Call it a night, or not. (Use case 7.0)
Extensions:	2a. Player's sheep goes to the same square the nightmare is located. 1. Sheep gets scared (See Use Case 6.0)
Frequency of Use:	Frequently, possibly once in every four cards played.
Status:	Done
Owner:	Tan
Priority:	High priority.

USE CASE 6 :

ID:	**6.0**
Title:	Sheep Gets Scared
Description:	Sheep's have two lives, if during the racing phase, a nightmare scares the sheep twice, the player needs to wake up.
Primary Actor:	Player
Preconditions:	1. Game is in racing phase.
Postconditions:	1. Racing phase continues. (Scared = 1) 2. Or the racing phase ends. (Scared = 2)
Main Success Scenario:	1. Player plays a card. 2. Sheep moves on the board. 3. A Nightmare card is drawn. 4. Nightmare moves to the square which sheep is in. 5. Sheep gets scared.
Extensions:	2a. Sheep moves on a square that nightmare is in. 1. Sheep gets scared. 2. If the sheep was scared before. RacingPhase ends. 4a. If the sheep was scared before. 1. Racing Phase ends. End of round starts. 4b. If the sheep is getting scared for the first time. 1. Sheep loses one life.
Frequency of Use:	Not so frequent. Happens mostly when nightmare and sheep are next to each other.
Status:	Done
Owner:	Tan
Priority:	High priority.

USE CASE 7 :

ID:	**7.0**
Title:	Player Calls it a Night
Description:	Calling a night means players sheep is removed from the gameboard. After calling a night Racing phase ends.
Primary Actor:	Player
Preconditions:	1. Game is in racing phase. 2. Players sheep crosses the fence.
Postconditions:	1. Game is finished, and a message is printed if player won. 2. Or the game is not finished, proceed to Resting Phase.
Main Success Scenario:	1. Player gains winks. 2. Player have two decisions.

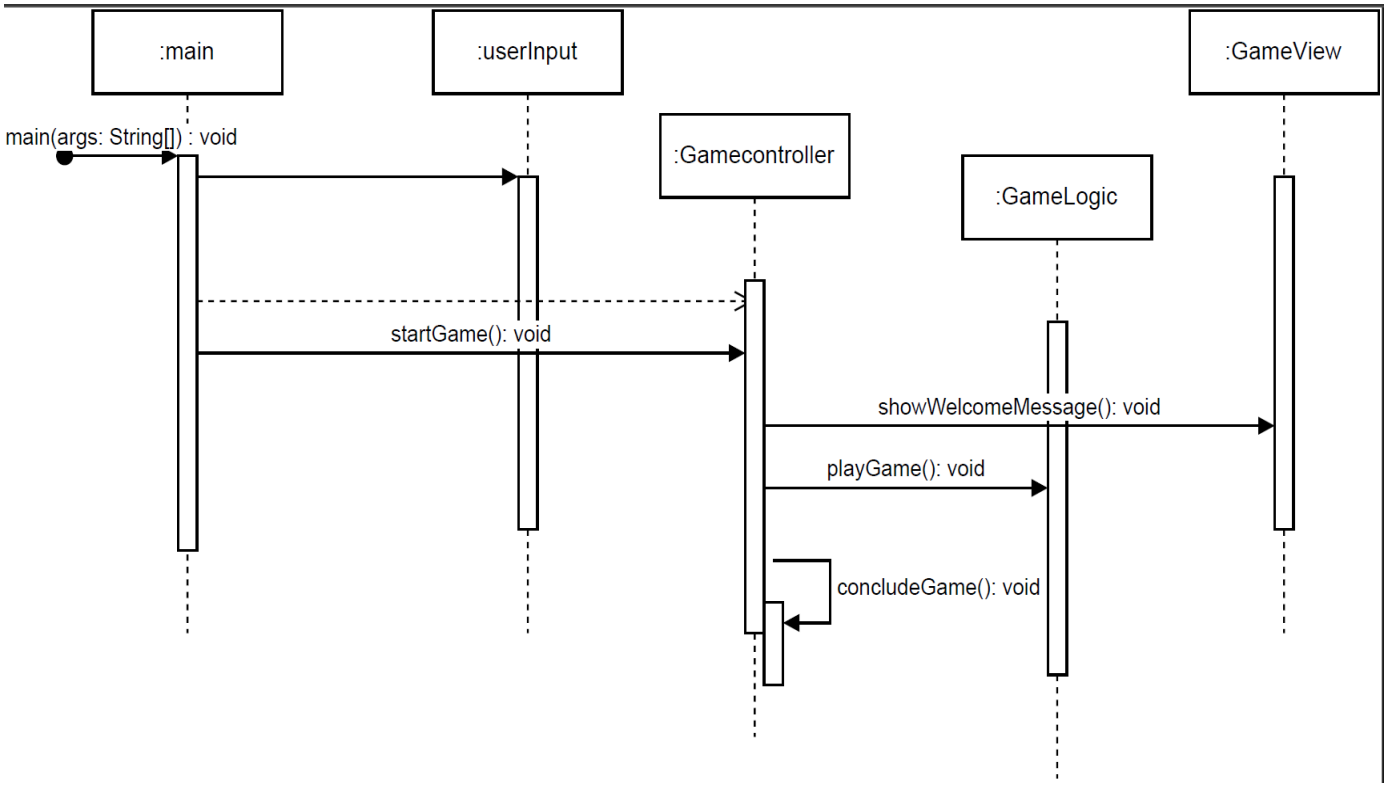
	3. If player calls it a night, players wink and pillow tokens locations are checked. 4. If player doesn't call it a night, racing phase continues. 5. Player plays another card.
Extensions:	3a. Player calls it a night, and their wink number is past their pillow number. 1. Solo Scoring calculation happens. (Use Case 8.0) 2. Player's score and a goodbye message print to the screen. 3b. Player calls it a night, but their pillow number is past their wink number. 1. Players pillow token is moved down 1 spot for every 5 winks gained during that Racing Phase. 2. If the player is forced to wake up by getting scared, (See Use Case 6.0) pillow number doesn't change. 3. Resting phase starts.
Frequency of Use:	Medium frequency.
Status:	Done
Owner:	Tan
Priority:	High priority.

USE CASE 8 :

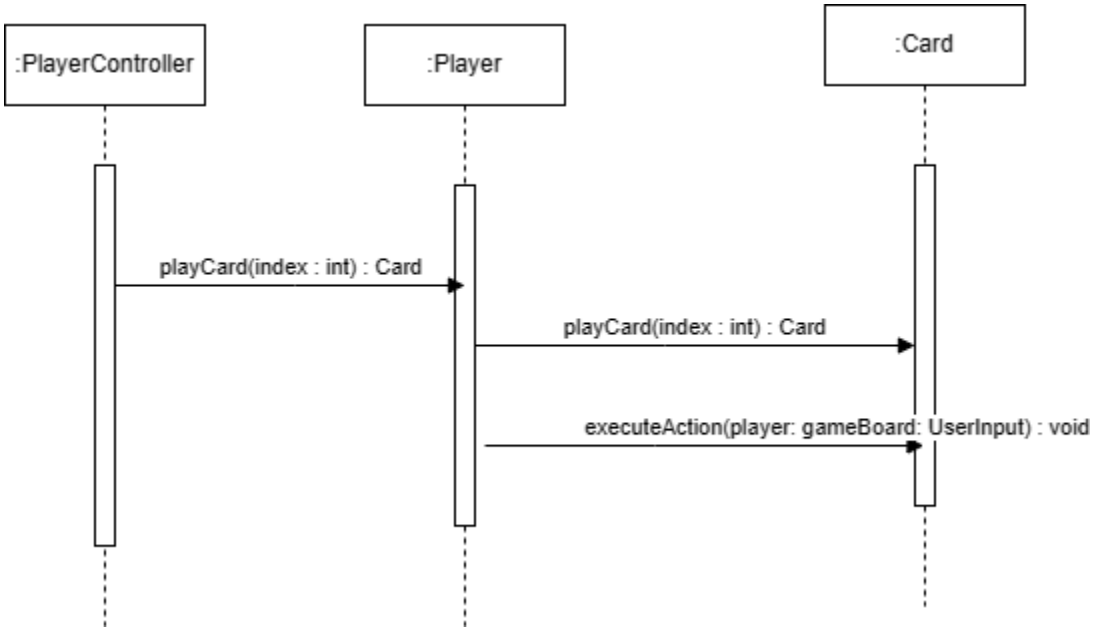
ID:	**8.0**
Title:	Resting Phase.
Description:	After Racing phase and end of round actions end, player moves on the resting phase where the player can add new tiles to the board, or add zz tokens to the tiles are already on the board.
Primary Actor:	Player
Preconditions:	1. End of Round phase ends.
Postconditions:	2. Racing Phase starts
Main Success Scenario:	1. Player performs a Resting phase action. 2. Top of the dream tile stack is placed on the lowest number available place on the board.
Extensions:	1a. Player adds a dream tile. 1. Player picks a dream tile from the dream tile market which will be printed out the screen for player. Places the tile on the board. Tiles have different boosts for players if its activated. 1b. Player adds zz tokens to an existing tile on the board. 1. Player chooses to give up one of their tokens to place it on a tile.

	2. If in the racing phase, player steps on this tile, they will have an option to activate the boost on the tile.
Frequency of Use:	Every time after an End of Round phase, if the game is not concluded.
Status:	Done
Owner:	Tan
Priority:	High priority.

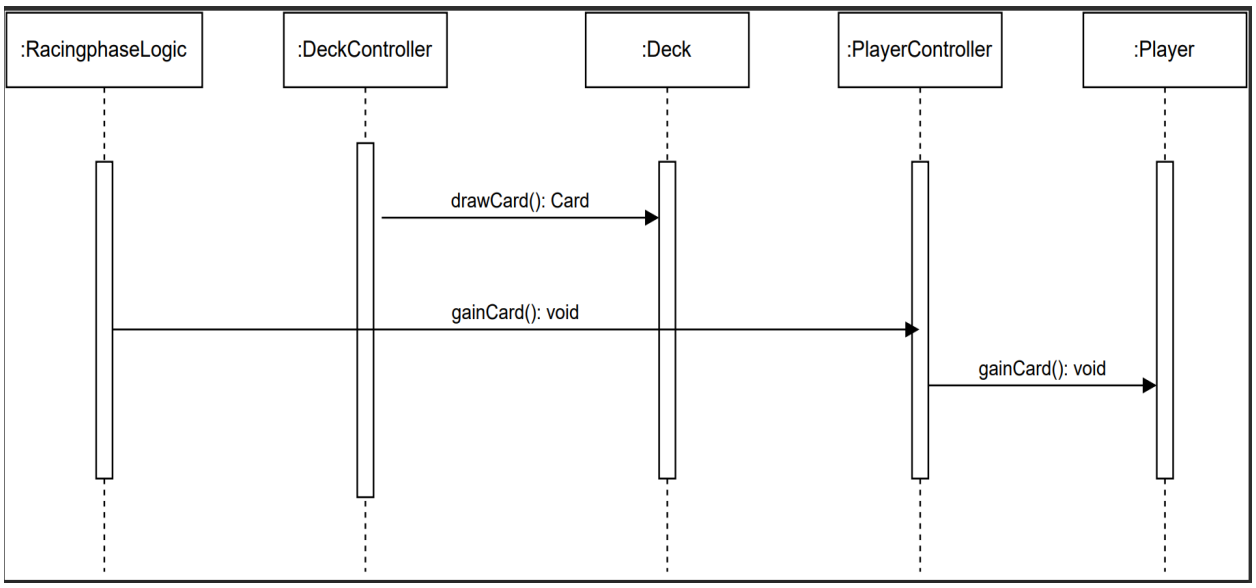
Start Game:



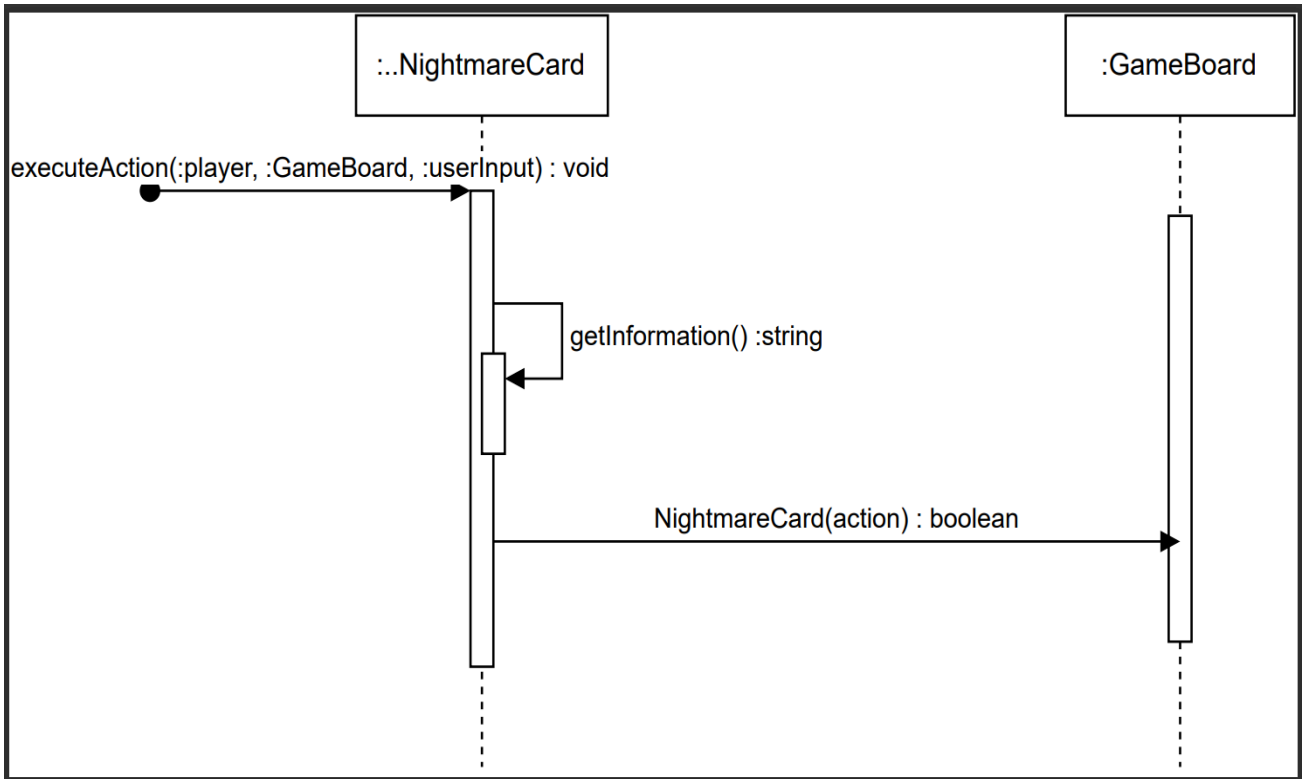
Play Card:



Deal Card:



Nightmare Resolution:



Gain Winks: