Design Document for Milestone 4

Response

For JUnit tests, we've been able to correct more than half of them. (All of the controller and dreamtiletest). We also removed the commented out debugging code we could find.

For the BumpInTheNight, NightmareWolf, SinisterSpider classes, we removed them as they were string classes, and modified NightmareReferenceCard appropriately to fulfill the necessary duties of selecting a Nightmare.

- Abstract card no longer insists on a parameterless constructor.
- We've improved the names for the CardBuilder field names
- For the over-engineered methods, we have to use "this" as that pertains to the Builder pattern, where we return this during construction.
- For CardDeck, we removed the magic numbers and attempted to reduce the chaining methods.
- We made checks to prevent the game from crashing when wrong inputs are received
- We changed the CardBuilder's variables "and" and "moveOr" to better names.
- We removed as many unused imports as we could find
- Removed the player's color attributes as we are only doing a Single Player game for now, so no need to differentiate players.
- We removed Drawable and DrawFromDeck as they were not necessary.
- We removed the misuse of generics
- For CardController's parsing, we changed the method name to better reflect the action.
- Removed magic numbers by using Card Factory, and now we have constants for all the numbers. For the DreamTileDeck we removed the hard coded list of Dream Tile Names to DreamTileFactory so that we segregate the OC violation to the factory. For Card1, Card2, they are all just Card now, so no more temporary variables.
- We improved the user journey to be more from the user perspective.
- Made all user journeys and use cases for 1 player.
- To address that there were missing aspects in the code; we have tried to ensure that all the code we've written is used and everything works as expected. In our design document we are also trying to ensure everything that requires addressing is addressed.

- We stated in our last design document that visitor pattern may be able to resolve some of the instanceOf uses in our implementation. The reason being is that visitor pattern allows for identifying the type of object in the visited class, hence alleviating the use of instanceOf in that particular class. The way this is done by defining an interface for the visitor and having each class in the structure implement an accept method that takes the visitor as an argument. Then, polymorphism can be used to detect the type of object, rather than instanceOf. Note that we did not use the Visitor Pattern.
- Feedback states opinions should not be used to justify design. Duly noted.
- We haven't implemented every dream tile as some of them require multiplayer.
- Displaying card in the cardView and GameView is only an issue if we need to show the cards for more than one player at any given time. This is due to the fact GameView only has the current player's cardView's.

This is however not an issue for Single Player.

- We opted to use instanceOf in Actionable (line 178) because certain objects on the Board are scareable, and we needed to identify them. Scareable is an important type as it determines which tokens can be scared, and allows for extension of future tokens that can be affected by IScare objects.
- For Casting, we used method overloading and polymorphism in the Fence class. This allowed us to remove the casting and instanceOf.
  We also removed casting and instanceOf by creating two decks of cards (one for Sheep cards and one for Nightmare Cards), and randomly drawing, using mathematics for probability of randomness, from the appropriate deck.

For design principles/patterns/decisions, here's what we used

- CardBuilder (Builder Pattern) -- because we found cards can vary in their attributes and in order to remove telescoping constructors.
- NightmareCardBuilder (Builder Pattern) -- same as above
- CardFactory (Factory Method Pattern) -- To fix OCP by not creating the cards in the Card Deck -- instead, moving the violation to the factory.
- DreamTileFactory (Factory Method Pattern) -- To segregate the OCP violation to the factory class, rather than violating OCP in the DreamTileDeck Class through the creation of Dream Tiles there.
- GameView (Composite Pattern) -- allows us to better organize the User Interface and treat all the components uniformly.
- GameInformation and GameController (Facade Pattern) -- both serve to simplify access to their subsystems by having less interfaces for the required operations. GameInformtion serves to access model components from the controllers without having to directly interact with each model component. GameController serves to interact with other controllers through the views without actually having to interact with them; GameController also allows for a simplified way to interact with the model for playing the game.

For the SOLID Principles, here are some examples of us applying SOLID. We've applied it to several other classes too.

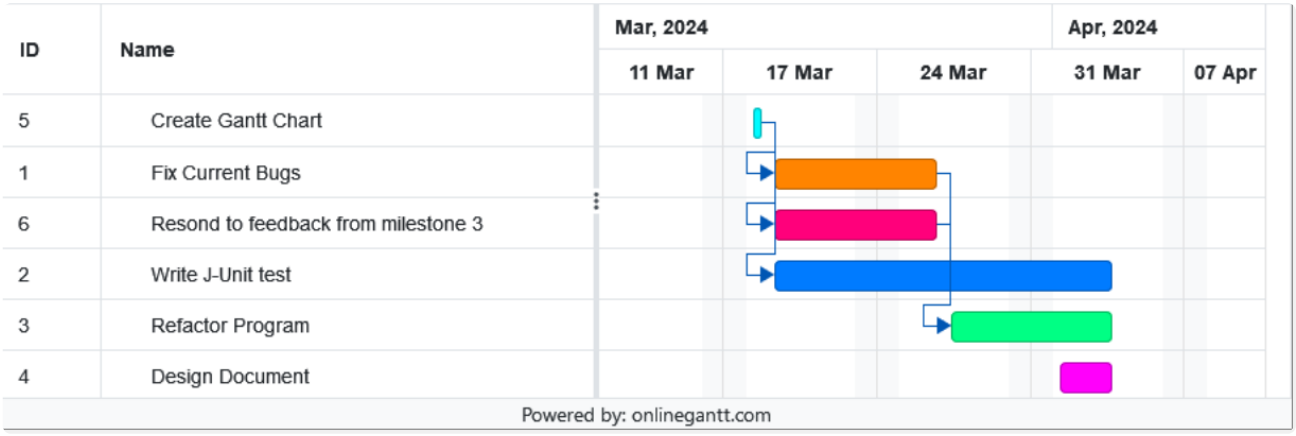
- For Single Responsibility, CardDeck is an example of satisfying SRP as the responsibility of the class is to merely hold the cards and provide a way to draw from the deck.
- For Open-Closed Principle, CardFactory is an example where we moved the OCP Violation from CardDeck to the Factory.
- For Liskov Substitution, SheepToken, WinkToken, PillowToken and NightmareToken are an example where we have multiple implementing classes of supertype IToken. We are able to use them in the Boards, treating them as their super type when it comes to their movement around the board.
- For Interface Segregation Principle, an example is the IScare Interface -- only some classes need implement the ability to scare: some Dream Tiles, and the Nightmare Token.
- For Dependency Inversion Principle: Almost all of our concrete classes **that are being referenced by other classes** implement an interface. This allows the referencing classes to be dependent on an abstraction, rather than a concretion. Henceforth, the dependency is inverted, as required.
  An example of this is the IToken Interface. Many classes reference the subtypes of IToken. However, we ensure that they reference an abstraction rather than a concretion by making them reference the interface types, satisfying DIP by definition.

Additional Design Decisions

We've made significant refactorization in the GameController class, as well as GameInformation class. There's also been several refactorizations throughout other classes too.

- In GameController he previously had everything in nested functions with very few methods. In this milestone we've extracted many methods and moved the ones related to the model into GameInformation. Any extracted methods that were related to game flow stayed in GameController. Now our GameController class is far more modular.
- In GameInformation we added methods to reduce chaining. As I said above we also added methods extracted from GameController to reduce bloat in that class.
- Some methods in GameController were also modified to accommodate for single player mode. We made sure to make single player a feature that can be turned off, rather than limiting our game from a future multiplayer mode.

Schedule and Adherence



However, the due date changed to April 8, which resulted on us working up until April 8. Generally, the schedule was followed, except for the part where one of the group members became very ill and was unable to contribute (during the first few days).

Diagrams

We've added two new sequence diagrams: InitializeGame and Play_Game_full. The earlier sequence diagrams should still adhere to the overall logic of the game.

The UML Class Diagram should also be in the github repository.