Design Notes

Sai Prithipa Ram Mohan

**Components**

**Player -** Player is the protagonist in the game and acts as the central character around which game revolves. A player object exposes behavior such as walk and jump. Player objects maintains three states - Super state, Normal State, Dead State. These states define the size of player and also notifies its observer of player being alive or dead.

**HealthBar -** HealthBar is a UI element of the game which maintains state of player’s health at any given time. HealthBar acts as an observer to player which sends notification to HealthBar to change its state if player is attacked by the enemy.

**ProgressBar -** ProgressBar is a UI element of the game which is used to maintain level progress and notify to its observer about the game progress state.

**PlayerName -** PlayerName as a UI component displays player’s name as selected by the user. This component stays statically at the upper right corner of the screen.

**HealthIcon -** This is a static UI component which acts as a label for HealthBar.

**CastleIcon -** This is another static UI component which acts as a label for ProgressBar.

**Design Patterns**

**Observer Pattern**

The observer pattern is used in the game to enable effective communication among game objects, specifically player and other game elements. For example, player notifies each observer if he attacks an enemy. This enables enemy object to trigger its disappearance from the world. This also triggers. The ScoreBoard Object to trigger an increase in the score.

**Classes:**

* **IPlayerSubject -** This class defines contract for Player as it acts as subject that would be observed by all other observers. As part of this contract, it exposes a method called attach(), which enables any other game object to add themselves as an observer to the player. Another method exposed as part of IPlayerSubject is removeObserver. This enables game objects to remove themselves from the observer list maintained by the player who acts as a subject.
* **IObserver -** This class defines contract for any game object who intends to act as an observer to IPlayerSubject. As part of this contract, game objects need to implement a method called playerUpdate(). This enables observers to get notification from Player who acts as a subject.
* **IProgressObserver -** This class can be implemented by any object which needs notification from player about the game progress. For example, LevelComplete object can implement this interface and implements its updateProgress() method. The player can then notify LevelComplete about when to end the level and move to another one.
* **IPowerObserver -** This class can be implemented by any object which needs to be notified when player touches any special elements such as LuckyCharm or Coin.

**State Pattern**

State pattern is used to maintain different state of the player object. These states are PlayerDeadState, PlayerNormalState and PlayerSuperState. State design pattern acts as a great way to efficiently change player’s size based on his state - “super” or “normal”. State pattern also acts as a notification trigger for GameOver component when player moves to PlayerDeadState.

**Classes:**

* **IPlayerState -** This class exposes method setState which can be used by any concrete implementation to modify player’s state.
* **PlayerDeadState -** This class changes the state of player to “dead” state. This state change triggers game over flow.
* **PlayerNormalState -** This class changes the state of player to “normal” state. This state change triggers size change of player from super to normal size.
* **PlaerSuperState -** This class changes the state of player to “super” state. This state change triggers size change of player from normal to super size.

**Mediator Pattern**

Mediator patterns helps in loose coupling between objects who needs common functionality. For example, to change state of a player, multiple objects need not hold an instance of all the possible state of the player. Instead, PlayerMediator can hold all the possible state of the player and expose a method which can be used by any game object to modify player state.

**Classes:**

* **PlayerMediator -** This class holds instance of all possible player states and exposes method to modify player state to any of allowed state. This class enables loose coupling among objects needing common functionality of updating player state.
* **HealthBar -** HealthBar object modifies state of player based on its health change. For example, when player’s health moves from 5 to 4, HealthBar updates the player’s state to PlayerNormalState.
* **MyWorld -** MyWorld object uses PlayerMediator object to set initial state of player to PlayerSuperState.

**Decorator Pattern**

Decorator pattern enables decorating any output depending on specified condition. In this game, decorator pattern is used to decorate the single instance of GreenfootSound object with different sound file input. For example, player makes different sound when he is attacked by an enemy versus when he attacks the enemy.

**Classes:**

* **Decorator -** This class exposes a generic decorator contract in form of method decorate().
* **AudioDecorator -** This class implements the decorator interface and triggers the sound decoration based on the input provided.
* **AttackAudioDecorator -** This class decorates the sound object with sound relevant to player attacking enemy.
* **AttackedAudioDecorator -** This class decorates the sound object with sound relevant to player getting attacked by the enemy.