FIT2099: Team 10 Design Rational

By : Ali Maskari , Sujin Lee & Madeline Ting

Requirement 1: Sujin

- **New map implementation**
  - In the main method of Application class, I have created a 'lavaMap' object which is a type of WorldMap. This lavaMap is a smaller version of gameMap with an extra feature, Lava Zone.
  - If the player steps on the Lava zone, the player will be damaged by 15 per turn.
  - Since Enemies cannot step into the Lava zone, I overwrote the canActorEnter method to return true if the actor is Player, otherwise return false (e.g. enemy).

- **Teleportation**
  - For the player to teleport to another map back and forth, the player has to use "Warp Pipe". This warp pipe is 1 in the lava map, and 3 in the game map. When initiating a warp pipe in the game map in Application class, it is given with multiple parameters; nextLocation - the next destination, nextWarpPipe - the warp pipe in the lava map (since there is only one warp pipe). Since the warp pipe is connected to another warp pipe, it has to have information about the other warp pipe.
  - However, when initiating the warp pipe in the lava map, we do not know where the next destination is yet before the teleportation is made from the game map to the lava map (as the player starts in the game map). It is certain that teleportation from the lava map to the world map is preceded by that from the world map to the lava map. So whenever the teleportation from the world map to the lava map, we record the previous location (the current location in the world map) and next warp pipe (the current warp pipe) so that next time when the player teleports from the lava map to the world map, the teleport action can refer to previousLocation and nextWarpPipe as nextLocation and nextWarpPipe.
  - After one turn passes, Piranha Plant will show up on the warp pipe. Once the player kills the piranha plant, the player gets to have JumpAction onto the warp pipe. After jumping onto the piranha plant, the player gets an option of teleport action. If the plant is on the warp pipe in another map, the plant may instantly be killed if the player teleports.
  - If there is a change in the design, let's say there are 3 warp pipes in each of the map, we can just simply save information about the previous warp pipe and next warp pipe like we did for the one in the lava map. It should be after some connections between the warp pipes are made; the open-closed principle has been kept since it safely can be implemented no matter what future design changes are.

Requirement 2: Madeline

- **Princess Peach**
  - Princess Peach is implemented as an actor, like Toad - who doesn't have any behaviours and mainly performs monologues. In order for the player to interact with Princess Peach, the conditions of allowableActions needs to be met; to be within the exits vicinity of Peach, and to have a Key stored within their inventory.
  - A EndGameAction has been created as it allows the player to initiate it as an action within their action list. When standing within Princess Peach's circumference with the Key in hand, the EndGameAction will be added into the player's action list. Upon selection, this will begin the game over sequence.
  - First, the player will be given the GAME_OVER capability status, which will call upon a check to see whether the player has this status - hence performing it's tasks.
    - The player's action list will be cleared completely.
    - Only ResetAction is added back onto the player's action list.

    - The game over message from Princess Peach will be returned and printed in the console.

- **Bowser**
- Bowser does not utilise WanderBehaviour as he stays put next to Princess Peach until the player stands within his exits circle.
- The fire that remains on the ground after Bowser's hits is created as an item that is dropped on the player's location every time the AttackAction is initiated by Bowser.
  Within the fire class, a tick is implemented to check whether an actor is standing in the same location, dealing 20 damage to any actor.
  A second tick is implemented to reset and remove the item whenever the counter (counterGround) reaches 3. This is because the fire only stays on the floor for 3 turns.
  Bowser has the Key added into his own inventory, which is dropped following his defeat by AttackAction's code. This item will appear in place of Bowser after knockout, which can be picked up by the player to shortly finish the game.

- **Flying Koopa**
- In order to avoid excessive repetition in our code, Flying Koopa was implemented through the pre-existing Koopa class.
    - A new constructor was created for Flying Koopa in order to accomodate for the differences between the Koopa varieties (eg. max HP, display character, status etc.)
    - Within the Mature tree class, after passing the 15% spawning chance, a new instance of the Koopa class will be spawned onto the map. However, depending on what side of the 50% chance is landed on, either the normal Koopa or the Flying Koopa constructor will be chosen - thus, allowing both to be spawned as separate enemies from the same class.
- To fix last assignment's implementation of Koopa, I created checks within the Enemy class's playTurn method to check for Koopa's with the KOOPA_DORMANT status. If fulfilled, then the following would occur;
    - All behaviours from this instance of Koopa will be cleared
    - Koopa's display character will be set to a 'D'
  Previously, there was an issue with Koopa still moving after being knocked out, as well as not changing display character. That issue has been resolved.
- As Flying Koopa can travel freely across walls and trees, I edited the canActorEnter() code within the HighGround class to accommodate Flying Koopa's agility and flying capability.
  I created and assigned Flying Koopa to a new status capability; KOOPA_FLYING, to easily call and identify this enemy within code via the hasCapability() boolean check.
    - *Can the actor enter ? Unless they have the KOOPA_FLYING status, they are unable to enter any high grounds.*
- **Piranha Plant**
- Piranha Plant spawns on top of the Warp pipe after 2 turns as an enemy without wandering or following behaviours as they need to obstruct the entrance to the lava map. Once Piranha hits 0 HP (!isConscious()), they will be removed from the map, revealing the Warp pipe set underneath them.

Requirement 3: Sujin

- **Bottle**

- New Bottle class has been created and it is instantiated in the player's inventory. The Bottle is not portable since it can't be dropped nor picked up. Bottle can keep water in the form of Stack, and each water has a unique effect. To keep the single responsibility principle, the effect of water is not implemented in the bottle class. It is implemented in the drinker interface.
- Not to violate LSP principle and to avoid using instanceOf, I created a BottleManager class. It has an attribute of HashMap that keeps the actor and the actor's bottle. Whenever the actor (the player for now) is created, the actor's bottle and actor is appended to the hashmap (registered) in the bottle manager.

- **Water fountain**

- The Fountain abstract class has been made; HealthFountain and PowerFountain extend the class. According to the dependency inversion principle, a higher module (fountain abstract class) does not know about lower modules (HealthFountain and PowerFountain).
- The Drinkable interface has been made; HealthWater and PowerWater implement the interface.
- The ground Fountain has an action of FillWaterAction. When the player steps onto the ground Fountain, the player gets to have the FillWaterAction. In the FillWaterAction, the action is executed by filling the water from the fountain in the bottle derived from the bottle manager (more specifically, the getBottle(Actor) method). getBottle method returns a bottle owned by the actor (parameter).
- The player implements the Drinker interface so that whenever the player drinks the drink in the bottle, according to the features of the drink, the drinker interface can implement those features; open-closed principle is kept since it is open to future changes - if more actors can consume the drink, they can simply implement a drinker interface.
- If the player wants to consume the drink (drinkable) in the bottle, the player uses the consumeAction. If the player consumes the bottle, it will execute the Drink method in the drinkable class (for example, Health water or Power water). Drink method gives drinkers some features such as heal and increaseAttack. For example, if the player drinks Health Water, the drink method in health water will execute the 'heal' method in the drinker class. The single responsibility principle is kept since each method, class and interface has only one job regarding the purpose of it. Their job is specialised.
- Interface segregation principle has been kept since we have a consume (drink) action for drinking the water, and as for implementing the features of water, we have another interface (drinker) in the drink method in the water class. The classes and interfaces implement methods that they only care about. They could be a larger interface - but they are separated into smaller interfaces.

Updates from previous Design: Ali

- Changed the spawning(creation) of the different age stages of the tree [Sprout,Sapling,Mature] to be only within the specific tree stage class. Eg in the older version of the game, tree stages were transformed from the tree class which did not represent a good design representation because it violated the principle of **single responsibility principle** which states that each class should be responsible for one thing in the system which makes code easier to test and maintain in the long run.

- Tree is kept as an abstract class to be able to show only the relevant details we wanted in each tree stage.
- Added a **Monologue.Java** class, because this class will be used to as a contolor for Toads messages being said randomly to minimise Toad.Java class's responsibilities (from the previous version of the game) and removed toads messages conditions of players occupation of a weapon Wrench or item Power Star from being inside the SpeakAction.Java class , to being within toad execution of his own message(see below).

Requirement 4: Ali

- **FireFlower.Java:**
  - To implement the **FireFlower.Java**, it will be extending the abstract class item.Java that was given to our team inside the base code, that is because the characteristics that the fire flower has from the requirement and my understanding of the game are more closely related to an item rather than the first thought of Tree.Java or Ground.Java, following the **abstraction principle** of OOP to be able to use to apply the same functionality given within the item class inside the fire flower such as the tick method that is associated with the current location of the actor.
  - For the player to be able to consume the fire flower, FireFlower.Java will implement the interface Consumable.Java that was developed and used in the older version of the game following the understanding of the **interface segregation principle,** motivates developers to implement multiple small interface if needed, if not they should use existing interfaces that do the same thing in order to minimise code written and produce a good design.
  - When the player picks up a fire flower he will have two options. The first
    The first option would be if he wants to keep it in his inventory or consume it and get FIRE ATTACK.

- **Fire.Java:**
  - A new feature was introduced in the version in the form of a weapon. **Fire.Java** , is a new class that extends the pre-existing abstract class of Weapon.Java given within the engine code, given that the player decides to consume the fire flower to gain the capability of attacking with fire.
  - It will implement item's both tick methods, one will be responsible of the damage it will cause an actor that is standing on it, meaning in each turn the actor is on top of the fire will be dealing a -20 in his life points till it gets removed from the game, the other tick method will be responsible of the passage of time of the fire attack capability that mario the player has, which is 20 turns.
  - By extending the weapon.Java class, it will be convenient for us to reduce the amount of new code written and use a built code for the implementation of Fire.Java class.

Requirement 5: Ali

- **Monologue.Java:**
  - As mentioned above, in this version of the game the monologue class is used to help developers control any NPC's conditions when speaking. This implementation allows us to push,get,remove,clear any monologue we would want in the future if more characters with conditions are introduced to the game, ensuring that the **Open-Closed Principle** in SOLID is followed as the rest of the other actor's code implementation

can easily use an instance of this class within their own class to compute any monologue related conditions(as for what happened with toad).

- Monologue.Java will have a function called pushMonologue() that will be responsible for pushing out the monologue based on their index position. This function will have two parameter like this:

  [PushMonologue(int index, String monologue]

  That will be responsible of adding m=the message that the character will
  Shout out.

- **Speakable.Java:**
  - Speakable.Java is an interface that will allow all NPC to implement its methods which is "speak ( )" that will enable them to accomplish the requirement of speaking, following the principle of **interface segregation** which in its context states that it's better to create lots of interfaces than having few better ones, which explains why the implementation of the pre-existing Behaviour.Java interface is left out of this equation.
  - All NPC implementing this interface and its speak() method will allow them to speak, as to have an excellent design we must leaning towards decoupling and design composition over inheritance.therefore, our design will avoids having one huge general purpose interface as what was given to us in the base code at the start of the project.