FIT2099 Assignment 1: Design Rationale

Lab 6 Team 3

REQ1: Trees

The tree stages were implemented as 3 subclasses that inherit from the abstract class **Tree**. Since all tree stages are expected to grow to a new stage after a set amount of turns, The number of turns required to grow should be stored as an instance variable as part of **Tree**. Once the requisite amount of time to grow has passed, a new object of the next tree stage may be returned. Once 5 turns has passed to grow for the Mature stage, the growth timer may be reset back to 5 to for it's to regrow a new sprout.

As sprouts require fertile land to grow, we have chosen to give **Ground** objects that are fertile the capability **FERTILE** in the enumeration **GroundTraits**. This is as future **Ground** tiles may also be fertile tiles, requiring a unified identifier for all such tiles. This allows us to follow the *"Don't repeat yourself"* design principle through use of public constants. **Ground** already features a **CapabilitySet** which we can use to store the enumeration value.

To check for adjacent **FERTILE Ground** for **Mature**, we will check the current location's **Exits** if they have the capability of **FERTILE**. This is as we know the surrounding exits are adjacent to the location. We can then randomly select one of the **FERTILE** locations and replace its **Ground** with **Sprout**.

Pros

- Safeguarding implementation of grow() and growTurns attribute.
- Able to identify all growth stages as a Tree class.
- Checking **Exit**s ensures **Location**s are adjacent to **Mature**

Cons

- Additional code required to implement abstract class.
- Requires **Mature** to handle setting ground of **FERTILE** at location.

Assignment 2 updates

- Added class Utils for random number generation for spawning and growing events. This is to follow the *Single Responsibility Principle* by having dedicated class responsible for random number generation (RNG) as opposed to Tree and its subclasses implement their own RNG.
- Added abstract method .spawn() to Tree. This requires subclasses of tree to handle spawning of either enemies or items at each turn.

REQ2: Jump

JumpActorAction extends **MoveActorAction**: We want **Player** to have an action to move to another location in the **GameMap**, the **MoveActorAction** has the methods to enable **Player** to have this ability. This will overide the execute() method to include the success rate check for jumps and if **Player** has the **TALL** status. It will also override the description() method of the action.

JumpManager is associated with **(interface) Jumpable**: We want **JumpManager** to store the **Jumpable** grounds.

(abstract) Tree and **Wall** implements **(interface) Jumpable**: We want only **Tree** and **Wall** grounds to be jumpable but not others, such as **Dirt**. Hence, we use interface. This will also add the **JumpActorAction** to the **ActionList** to allow **Actor**'s with the **CAN_JUMP** status the ability to jump to the **Jumpable** terrain.

Pros

- This follows the SOLID principle *"the Dependency Inversion Principle"* allowing for other objects (i.e. **JumpableManager**) to depend on the abstraction **Jumpable** rather than the **Ground** classes that are jumpable.

Cons

- Use of abstraction - slower time complexity and more use of resources (less efficient).

Assignment 2 updates

- Replaced **Jumpable** interface w/ abstract class **HighGround**, where **Tree** and **Wall** grounds **extend** from.
- Removed **JumpManager** (redundant)

REQ3: Enemies

Goomba and **Koopa** both extends the abstract class **Enemy**, and **Player** extends the abstract class **Friendly**. As we know enemies have different methods compared to friendlies, e.g enemies are able to wander and follow the player but player can't. Indeed, both enemy and friendly extends actor. Enemies would have the **AttackAction** applied to its action list by default for the **Player** to attack it. This can be overriden for different behaviour such as **Koopa**

Once a **Koopa** is damaged enough, it will be given the capability **DORMANT** to signify it cannot be attacked or move. We can choose to add the **AttackAction** to the **Koopa** action list if it is not **DORMANT**, preventing other **Actor**s seeing the option when next to it.

AttackBehaviour, **WanderBehaviour**, **StationaryBehaviour** and **FollowBehaviour** all implement **(interface) Behaviour**, so can act these behaviours without the player's input. The **AttackBehaviour** should take in a target **Actor** upon construction and stored as an instance variable like **FollowBehaviour**. This would allow us to set enemies to attack the **Player** when nearby.

Player holds **Wrench** If **Player** is holding **Wrench** in the inventory, then **SmashShellAction** can be called to kill **Koopa**. This will drop a **SuperMushroom** on its location upon death.

The **Goomba**'s kick and **Koopa**'s punch will be implemented by overriding the **getIntrinsicWeapon()** method of **Actor**. Here we would specify the attack it performs by setting the verb and damage *(the hit rate is the default 50%)* of **IntrinsicWeapon**'s.

Pros

- Easier to understand and follow, e.g. an **Action** is used on the object being attacked
- Separating Actors into either enemies or friendlies (abstract classes) allows simple, clear divide between two Actors with different methods.
- IntrinsicWeapon best describes the type of attack for **Goomba** and **Koopa** and enables us to set the damage.

Cons

- More space and time complexity is required for abstraction.
- Cannot override the hit rate of **IntrinsicWeapon** for future enemy implementations as it is default to 50%.

Assignment 2 updates

Main implementations * **FollowBehaviour** is added to enemies only when they call **AttackAction** in their last turn, through checking the parameter lastAction. I found this was the most optimal way enemies follow other actors as they already engage in fights with any actor close in proximity. * Used **Utils** static method **nextChance()** to create 10% chance of Goomba calling **SuicideAction** to be removed from GameMap. * Koopa cannot be removed from map due to creation of **INDESTRUCTIBLE** status * Koopa becomes **DORMANT** (Status): **AttackBehaviour**, **FollowBehaviour**, **WanderBehaviour** behaviours are stripped from Koopa.

Changes * Removed **StationaryBehaviour** * Enemy and Friendly, along with their subclasses have been divided into two packages, Friendlies and Enemies. * New Action **SuicideAction** implemented, results in removal of Actor from GameMap. * **SmashShellAction** is now responsible for dropping SuperMushroom upon Koopa's death.

REQ4: Magical Items

A new abstract class **MagicalItem**, which is inherited from the base class **Item**, has been added. **PowerStar** and **SuperMushroom** are implemented as inheriting **MagicalItem**. This is done to differentiate 2 types of **Item**: item that can be equipped as a weapon - **WeaponItem**; item that is not a weapon - **MagicalItem**. **MagicalItem** is consumable, which provides status to the **Actor**.

Eating a **SuperMushroom** will increase the **Player**'s max hp with **Actor**'s increaseMaxHP() method. This will also give the **Player** the status capability **TALL** to signify he has eaten the mushroom *(This is indicated with the icon M)*. Upon taking damage, if **Player** has **TALL**, it will remove the capability.

The **PowerStar** item will be given the **FADING** status. This counter for its duration would be stored as part of the **PowerStar** item, decreasing every tick until its removal.

High ground may check to see if the **Player** is standing on it and has the **INVINCIBLE** effect. If so, it will convert to dirt and drop a coin. Upon taking damage, we check the **Player** has the **INVINCIBLE** status to see if we are dealt damage.

The **AttackAction** may also check for the **INVINCIBLE** status of the actor to instantly kill a target when attacking after checking that **Player** successfully hits.

Pros

- More defined purpose of items
- Avoid the need of implementing 2 interfaces
- Future-proof for adding new features, like an inventory system

Cons

- Increased code sized
- Potentially harder to debug

Assignment 2

For the **PowerStar**, its effect, instead of just 1, is divided into 4 statuses:

- **HIGHER_GROUND**: allow actors to move onto higher ground without jumping and destroy that ground
- **COIN_FROM_DESTROYED_GROUND**: every destroyed higher ground drop $5
- **IMMUNITY**: actors receive no damage
- **INSTA_KILL**: actors can kill an enemy instantly upon a successful attack

By separating a compound effect into individual atomic statuses, it will be clearer to us what effect(s) a status provides. For example, the **INVINCIBLE** status from the initial design carries similar meaning with **IMMUNITY**, but it doesn't carries meanings of "Killing enemies instantly" nor "Walk to Higher Ground and Destroy it" etc. It would be more future-proof, as we may have new items providing lesser effects, maybe just **IMMUNITY** and **INSTA_KILL**. The same rationale also applies to **SuperMushroom**, which inital status **TALL** is divided into **TALL** and **EASY_JUMP**. A **ConsumeAction** class is created for **MagicalItem** to provide actions for the players, which is actually required and was missing in the inital design. A **StatusManager** singleton class is introduced to manage all actors' effects with duration. This is done to avoid modifying the based code of Actor and writing smelly code.

REQ5: Trading

Two new classes, **Toad** and **Coin**, are added in this section. **Toad** serves as the item shop in this game, and **Coin** is the currency for buying items from **Toad**. **Coin** is inherited from **Item** as, like all items, coins can be picked up from the ground. However, it is not inherited from either **WeaponItem** or **MagicalItem**, as it can not be equipped as a weapon or consume to gain status. **Coin** should have 1 Integer attribute to represent its value and 1 static String attribute to represent its visual to be displayed. **Player** should also have an integer value tracking the amount of currency they have.

Pros

- No additional class to manage currency of only **Player**.

Cons

- Not as simple as each item having an Integer attribute representing its cost/player having an Integer attribute representing how much the player have.
- Requires other classes to implement class attribute to store total amount of currency.
- (currently, this is exclusive to player).

Assignment 2

Wallet class is created for managing credit of actors, and **WalletManager** singleton class are created for managing wallets of actors. **Item** no longer has **Coin** as attribute.

REQ6: Monologue

GameMap and **Location** keep track of where **Toad** and **Player** are. If close enough, **SpeakAction** is available as an option: grabs monologue from **Toad**, checks if **Player** contains **Wrench**, **Status** of a PowerStar or else the **CapabilitySet** of the Actor e.g cannot be Goomba and decides what to say. Uses **Display** to print monologue string.

Player holds **(abstract) WeaponItem**, and is extended by **Wrench**. Use of **(abstract) WeaponItem** from engine, ease of implementation of new weapons in the future. **(enum) Status** able to contain the different statuses for **Player**

Pros

- Choice to utilise enumeration will avoid the excessive use of literals, and hence improve maintainability and extensibility of the code in the long-term.

Cons

- **Toad** isn't directly responsible for calling **SpeakAction**, but rather relies on **GameMap** and **Location**, e.g, **Player** must close enough in coordinates to **Toad** for **SpeakAction** to be available in console (could be a possible way for Toad to directly call **SpeakAction**).

Assignment 2

- To allow for a NPC to talk, we have decided to use the interface **Talkable**. This requires the implementer to have the method **getMonologue()** which is responsible for getting the monologue spoken by the implementer. This meets the SOLID principles *Interface Segregation Principle* as it separates the single action of monologuing as its own interface. In addition, we are also able to uphold the **Open-Close Principle** as we can continue to add additional objects that can talk without modifying those that use the **Talkable** in their methods (**SpeakAction**).
- Having **Toad** store the **SpeakAction** as part of their action list isn't a con as this utilises the existing action system of engine which checks adjacent tiles for available actions for the **Player**.

REQ7: Reset Game

A **Resettable** interface and a **ResetManager** class are created. All classes that can be reset, like **Player** and **Enemy**, implements this interface. And **ResetManager** manages the reset process. This follows the *Dependency Inversion Principle* which depends on the *Resettable* interface. Classes that implement *Resettable* will only depend on it.

We will iterate through all items that are in the ResetManager and call their reset methods. They are then items deleted will be removed from the manager.

- For trees, the reset() method will perform a 50% chance to change the location **Ground** to **Dirt**.
- For each enemy, we will remove it from the **GameMap** with .
- For the **Player**, we will check if they have status **TALL** or **INVINCIBLE** and remove the capability if they have it. We will also have it set the **Player** current HP attribute equal to the max HP attribute to fully heal **Player**.
- For coins, we will use its **Location**'s removeItem() method to remove it from its location.

Pros

- Easier to debug this features as all the reset method are implemented in this class
- Able to specify which items are reset with the **Resettable** interface, following the *Interface Segregation Principle* to solely be responsible for what occurs to the item during a reset.
- ResetManager tracks *Resettable* instances allowing us to query it to get all resettable instances.

Cons

- Increased code size and complexity

Assignment 2 updates

- Use of the interface **Resettable** assists in following the **Open-Closed Principle** as this allows us to continue to add more things that can be reset without modifying previous implementations. As such, we are able to call the **resetInstance()** method of a **Resettable** without modifying the caller of the function (**ResetManager**).
- The **RestManager** also follows the **Dependency Inversion Principle** by storing all **Resettable** instances which we may interface with to work with all implementers. This then assists us during the **ResetAction** as we can iterate through to reset each entry.

- For Coins and Trees, we require to remove/change them at their location. However, the **resetInstance()** method does not receive the location as a parameter. Our workaround to uphold the polymorphism of the method is to give the Tree/Coins the capability **RESET** to mark them for reset. Upon **tick()** being called, we may then remove/update them from their location as we receive it as a parameter there. This is also the case for **Enemy** where it is removed during the **playTurn()** method being called.
- Inclusion of the **Utils** class which is responsible for random number generation follows the **Single Responsibility Principle** as this allows other classes to call it when a random number is required, rather than each class storing a **Random** class as an attribute.