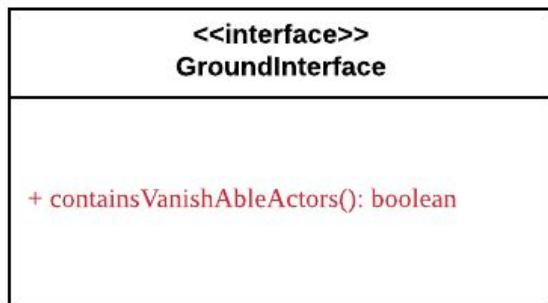# FIT2099 - Update Design Rationale for Assignment 3

Name: Isaac Lee Kian Min
StudentID: 31167462
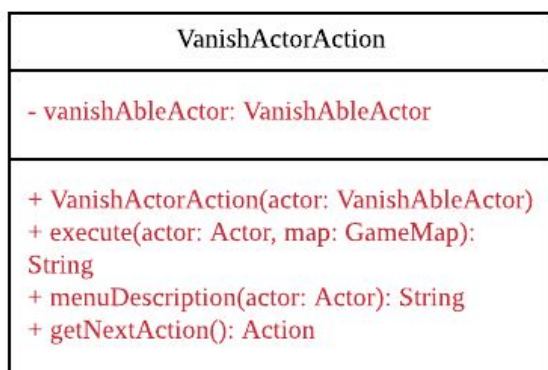
Old Classes Modified

The words in **red** indicate a new variable/method being added into the class, whereas words in **black** indicate extending an already existing method.

```
                ┌─────────────────────────────────┐
                │         <<interface>>           │
                │        GroundInterface          │
                ├─────────────────────────────────┤
                │                                 │
                │                                 │
                │ + containsVanishAbleActors(): boolean │
                │                                 │
                │                                 │
                └─────────────────────────────────┘
```

- A **containsVanishAbleActors** method to determine whether the ground houses actors that are able to vanish/disappear.
- Designed it this way because:
    - To reduce downcastings, which causes dependencies between classes
    - Make use of polymorphic code.

New Classes Added/Extended

The words in **red** indicate a new variable/method being added into the class, whereas words in **black** indicate extending an already existing variable/method.

```
        ┌────────────────────────────────────────────┐
        │             VanishActorAction              │
        ├────────────────────────────────────────────┤
        │ - vanishAbleActor: VanishAbleActor         │
        ├────────────────────────────────────────────┤
        │ + VanishActorAction(actor: VanishAbleActor)│
        │ + execute(actor: Actor, map: GameMap):     │
        │ String                                     │
        │ + menuDescription(actor: Actor): String    │
        │ + getNextAction(): Action                  │
        └────────────────────────────────────────────┘
```

- A **VanishActorAction** class extends an **Action** class to make an actor that is able to vanish disappear from the current game map.
- A **vanishAbleActor** attribute to determine the actor to be vanished.
- A **VanishActorAction** constructor to create a VanishActorAction object.
- An override **execute** method that makes the vanishAbleActor disappear.

- An override **menuDescription** method that returns a description of the vanished action.
- An override **getNextAction** method to determine whether an actor remains vanished or not.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Encapsulates the action of vanishing into one container class.
    - Use the Single Responsibility Principle as this class's only concern is to cause an actor to be vanished from a game map.

```
                VanishGround
─────────────────────────────────────────────
- vanishAbleActorActionHashMap:
HashMap<VanishAbleActor, Action>
- randomLocationGenerator:
RandomLocationGenerator
─────────────────────────────────────────────
+ VanishGround(actor: VanishAbleActor,
action: Action)
+ containsVanishAbleActors(): boolean
- containsMamboMarie(): boolean
+ tick(location: Location)
- reappear(location: Location)
```

- A **VanishGround** class extends a **Dirt** class which houses the actors that are able to vanish from the game map.
- A **vanishAbleActorActionHashMap** attribute to determine the actors that vanished at this location and their last actions.
- A **randomLocationGenerator** attribute to help generate a random location.
- A **VanishGround** constructor to create a VanishGround object.
- An override **containsVanishAbleActors** method to determine whether this ground houses vanished actors.
- A **containsMamboMarie** method to determine whether this ground contains MamboMarie.
- An override **tick** method that gives this ground the flow of time, while also determining whether any vanished actors will reappear on the game map.
- A **reappear** method that determines whether any vanished actors wants to reappear on the game map.
- Designed it this way because:
    - Extension of existing classes to make use of existing mechanisms for accessing a ground, thus reducing possibilities of duplicate codes.
    - Use Open-Closed Principle by overriding superclass methods to implement own behaviours, thus encapsulating this unique behaviour in its own class.

```
                    <<interface>>
                    VanishAbleActor

        stillVanished(): boolean
        getActor(): Actor
        getVanishAble(): VanishAbleActor
```

- A **VanishAbleActor** interface that allows an actor in the game to have the ability to vanish from the game map and reappear by implementing it.
- A **stillVanished** method to determine whether the actor remains vanished.
- A **getActor** method to get an Actor reference of this VanishAbleActor.
- A **getVanishAble** method to get a VanishAbleActor reference of the actor.
- Designed it this way because:
    - Provides a blueprint in the form of an interface for all actors to be able to vanish from the game map.
    - Allows for reuse in the case where we create another new character that can also vanish.
    - Provides a layer of abstraction to avoid downcastings by just having a one general reference (e.g. reference a subclass that implements this interface with the superclass), making use of polymorphic code.

```
                    ZombieWorld

        - quitingGame: boolean

        + ZombieWorld(display: Display)
        + processActorTurn(actor: Actor)
        + stillRunning(): boolean
        - checkForMamboMarie(): boolean
        - quitGame(): boolean
```

- A **ZombieWorld** class that extends the **World** class to implement certain behaviours.
- A **ZombieWorld** constructor to construct a ZombieWorld object.
- An override **processActorTurn** that checks whether the player wants to quit the game on top of processing the actor's play turn.
- An override **stillRunning** method checks certain conditions to determine the current state of the game (i.e. check whether specific actors are alive) on top of checking whether the player is still alive.
- A **checkForMamboMarie** method that determines whether MamboMarie still exists in the game.

- A **quitGame** method used as a menu to prompt the player on whether it wants to quit the game.
- Designed it this way because:
  - Extends existing class to reduce code duplication by inheriting from parent class.
  - Override to implement own behaviours and implementations, thus encapsulating inside its own class.

MamboMarie

+ VANISH_PROBABILITY: double
- justVanished: boolean
- playTurns: int
- behaviours: Behaviours[]

+ MamboMarie()
+ playTurn(actor: Actor, lastAction: Action, map: GameMap, display: Display)
+ stillVanished(): boolean
+ getVanishable(): Vanishable
+ getActor(): Actor

- A **MamboMarie** class that extends a **ZombieActor** and implements a **VanishAbleActor** to imitate the Mambo Marie character.
- A **VANISH_PROBABILITY** static attribute to determine the probability of Mambo Marie remaining vanished from the game map.
- A **justVanished** attribute to determine whether it just disappeared from the game map.
- A **playTurns** attribute to keep track of the number of turns played.
- A **behaviours** attribute that determines the behaviours of it.
- A **MamboMarie** constructor to create a MamboMarie object.
- A **playTurn** method to determine the actions played based on its behaviours.
- An implemented **stillVanished** method to determine whether the actor remains vanished.
- An implemented **getActor** method to get an Actor reference of this VanishAbleActor.
- An implemented **getVanishAble** method to get a VanishAbleActor reference of the actor.
- Designed it this way because:
  - A static attribute used to remove embedded literals while at the same time provide constant access to all objects of such since its attribute is the same among all instances.
  - Extends an existing class to reuse some methods defined in the superclass, reducing duplication of codes.

```
┌─────────────────────────────────────┐
│           ChantBehaviour            │
├─────────────────────────────────────┤
│                                     │
├─────────────────────────────────────┤
│ + getAction(actor: Actor, map: GameMap): │
│ Action                              │
└─────────────────────────────────────┘
```

- A **ChantBehaviour** class that implements a **behaviour** and is a nested private class within the **MamboMarie** class.
- An implemented **getAction** method to determine the actions based on its behaviour.
- Designed it this way because:
    - Inner class inside the MamboMarie class because the chanting behaviour of summoning new zombies is unique only to Mambo Marie's chant.
    - Implements an existing interface to avoid downcastings and provide a centralized way to access and invoke its method.

```
┌─────────────────────────────────────┐
│          CreateZombieAction         │
├─────────────────────────────────────┤
│ - numberOfZombie: int               │
│ - randomLocationGenerator:          │
│ RandomLocationGenerator             │
├─────────────────────────────────────┤
│ + CreateZombieAction(actorNumber: int) │
│ + execute(actor: Actor, map: GameMap): │
│ String                              │
│ + menuDescription(actor: Actor): String │
└─────────────────────────────────────┘
```

- A **CreateZombieAction** that extends an **Action** and creates a specified number of **Zombie** actors in the game map.
- A **numberOfZombie** attribute that determines the number of Zombies to be created.
- A **randomLocationGenerator** attribute to help generate a random location.
- A **CreateZombieAction** constructor to create a CreateZombieAction object.
- An override **execute** method to create a specified number of Zombies.
- An override **menuDescription** method to return a description of how many Zombie's created.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Encapsulates the action of creating new zombies into one container class.
    - Use the Single Responsibility Principle as this class's only concern is to create new zombies within the game map.

```
                    Vehicle
  ─────────────────────────────────────────

  ─────────────────────────────────────────
  + Vehicle()
  + addAction(action: Action)
```

- A **Vehicle** class that extends an **Item** and is used to travel across different game
  maps.
- A **Vehicle** constructor to create a Vehicle object.
- An **addAction** method that adds a specified action into its allowable actions list.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Use the Single Responsibility Principle as this class's only concern is to mimic
      the behaviours of a vehicle.

```
              RandomLocationGenerator
  ─────────────────────────────────────────
  - rand: Random

  ─────────────────────────────────────────
  + getRandomLocation(actor: Actor, map:
  GameMap): Location
  - getRandomX(map: GameMap): int
  - getRandomY(map: GameMap): int
  - containsObejcts(actor Actor, map:
  GameMap): boolean
```

- A **RandomLocationGenerator** class that acts as a helper for getting random
  locations on a game map.
- A **rand** attribute used as a random number generator.
- A **getRandomLocation** method used to get a random location on a game map.
- A **getRandomX** method used to get a random x coordinate of a location.
- A **getRandomY** method used to get a random y coordinate of a location.
- A **containsObjects** method used to check whether the location contains objects or is
  impassable.
- Designed it this way because:
    - Use the Single Responsibility Principle as this class's only concern is to find a
      random location on the game map that is empty and passable.
    - Reduce the need for duplication of code as we can just reuse the class
      implementations.
    - However, this might lead to dependencies, which may undermine
      maintainability.

```
                Brewery

- potionFactory: PotionFactory

+ Brewery(newPotionFactory: PotionFactory)
+ allowableActions(actor: Actor, location:
Location, direction: String): Actions
```

- A **Brewery** class created that extends a **Ground** class to model after a shop that sells potions. (Disclaimer: the owner of this brewery is very generous and decides to give free potions)
- A **potionFactory** attribute that allows this Brewery to brew potions.
- A **Brewery** constructor that creates a Brewery object.
- An override **allowableActions** method that allows the Brewery to give out potions to actors for free.
- Designed it this way because:
    - Extends existing classes to avoid downcastings, which causes dependencies.
    - Encapsulates the implementations of it within a class.
    - Use the Single Responsibility Principle as this class's only concern is to provide a place to get potions for actors in the game map.

```
              UsePotionAction

- potion: Item

+ UsePotionAction(potion: Item)
+ execute(actor: Actor, map: GameMap):
String
+ menuDescription(actor: Actor): String
- actorDead(actor: Actor, map: GameMap):
String
```

- A **UsePotionAction** class that extends the **Action** class.
- A **potion** attribute that specifies the potion used.
- A **UsePotionAction** constructor to create a UsePotionAction object.
- An implemented **execute** method that allows the actor with this potion to use it.
- An implemented **menuDescription** method that returns a description of using the potion.
- An **actorDead** method to determine whether an actor is still conscious after using the potion.
- Designed it this way because:

- Extends existing classes to avoid downcastings, which causes dependencies.
- Use the Single Responsibility Principle as this class's only concern is to execute the action of using a potion.
- Encapsulates the action of using a potion into one container class.

PotionFactory
- rand: Random - potionConstructors: ArrayList<Constructor<?>>
+ PotionFactory(potion[]: Potion) + getPotion(): Potion

- A **PotionFactory** class acts as a factory for creating different types of potions.
- A **rand** attribute used as a random number generator.
- A **potionConstructors** attribute to store the type of potions this factory brews.
- A **PotionFactory** constructor that creates a PotionFactory object.
- A **getPotion** method that gets a randomly brewed potion.
- Designed it this way because:
    - Provide a centralized mechanism for creating potions, reducing possibilities for dependencies in the forms of constructing new objects.
    - Uses the Dependency Injection Principle in its constructor which allows the specification of the potions this factory can brew.
    - Use the Single Responsibility Principle as this class's only concern is to provide brewed potions.
    - However, there is a dependency link with the Potion class.

Potion
+ Potion(potionName: String, displayChar: char) + getAllowableActions(): List<Action> + usePotion(actor: Actor): String

- A **Potion** abstract class that extends an **Item** class and defines what a potion is.
- A **Potion** constructor that creates a Potion object.
- An override **getAllowableActions** method that allows an actor to get this potion since potions are not portable, i.e. they break once dropped on the ground.
- An abstract **usePotion** method that when implemented, executes the effects of using it, which may benefit or cause harm to the user.
- Designed it this way because:

- Provides a layer of abstraction, which is useful in avoiding downcastings through polymorphic code.
- Use the Liskov Substitution Principle as we can substitute the superclass with subclasses. In this case, Potion is defined as abstract.
- Extends existing classes to avoid downcastings, which causes dependencies.

PosionPotion
+ POISON: int
+ PoisonPotion() + usePotion(actor: Actor): String + toString(): String

- A **PoisonPotion** class that extends a **Potion** class that models a potion that poisons.
- A **POISON** static attribute that determines the damage this poison does.
- A **PoisonPotion** constructor that creates a PoisonPotion object.
- An implemented **usePotion** method that causes the actor to get poisoned.
- An overridden **toString** method that returns a name of the poison potion.
- Designed it this way because:
    - Extends the Potion class to make use of the abstraction used in Potion class and make use of inheritance.
    - Use the Single Responsibility Principle as this class's only concern is to model after a PoisonPotion.
    - Encapsulates its implementations and behaviours within a container class.

HealingPotion
+ HEAL: int
+ HealingPotion() + usePotion(actor: Actor): String + toString(): String

- A **HealingPotion** class that extends a **Potion** class that models a potion that heals.
- A **HEAL** static attribute that determines the damage this poison does.
- A **HeaingPotion** constructor that creates a PoisonPotion object.
- An implemented **usePotion** method that heals the actor using it.
- An overridden **toString** method that returns a name of the healing potion.
- Designed it this way because:
    - Extends the Potion class to make use of the abstraction used in Potion class and make use of inheritance.
    - Use the Single Responsibility Principle as this class's only concern is to model after a HealingPotion.

- Encapsulates its implementations and behaviours within a container class.

```
                GetPotionAction

- potion: Item

+ GetPotionAction(potion: Item)
+ execute(actor: Actor, map: GameMap):
String
+ menuDescription(actor: Actor): String
```

- A **GetPotionAction** class that extends the **Action** class.
- A **potion** attribute that specifies the potion to get.
- A **GetPotionAction** constructor to create a GetPotionAction object.
- An implemented **execute** method that allows the actor to get the potion.
- An implemented **menuDescription** method that returns a description of getting the potion.
- Designed it this way because:
  - Extends existing classes to avoid downcastings, which causes dependencies.
  - Use the Single Responsibility Principle as this class's only concern is to execute the action of getting a potion.
  - Encapsulates the action of getting a potion into one container class.

How are these classes used to implement the features described in the Assignment:

From the Going To Town question,
1) You must implement a town level. Place a vehicle somewhere on your existing map. The vehicle should provide the player with the option to move to a town map.
   a) Create a new class Vehicle that extends an Item abstract class. Create a new addAction method in it that adds a specified Action into its allowableActions variable.
   b) In the Application class, create a new instance of Vehicle and add a new MoveActorAction that moves to a different Location on a different GameMap.
   c) Designed it this way because:
      i) Extended this class with Item to reduce downcastings through polymorphic code and implement own behaviours.
      ii) Make use of the existing class (e.g. MoveActorAction) to reduce duplicate codes.

From the Mambo Marie question,
1) Mambo Marie is a Voodoo priestess and the source of the local zombie epidemic. If she is not currently on the map, she has a 5% chance per turn of appearing. She starts at the edge of the map and wanders randomly. Every 10 turns, she will stop

and spend a turn chanting. This will cause five new zombies to appear in random locations on the map.

a) Create a new MamboMarie class that extends the ZombieActor class.
b) MamboMarie has a playTurns attribute to keep track of the number of turns played.
c) Create an inner private class inside the MamboMarie class called ChantBehaviour, which implements a Behaviour interface.
d) Create a CreateZombieAction class that extends the Action class.
e) ChantBehaviour checks whether MamboMarie has played 10 turns. If so, it returns a CreateZombieAction which will create 5 new Zombies and place them on different Locations on the same GameMap as MamboMarie.
f) In the MamboMarie class, add a behaviour attribute that stores an array of behaviours, which are WanderBehaviour and ChantBehaviour in order.
g) Access these behaviours by implementing the abstract playTurn method.
h) Designed it this way because:
   i) ChantBehaviour implemented as an inner class to encapsulate the chant behaviour of MamboMarie, which results in the creation of new zombies.
   ii) CreateZombieAction encapsulates its complexities and behaviours within a class. It also applies the Single Responsibility Principle as it has the purpose of just creating new zombies, which also allows for easy maintainability and reusability.
   iii) However, the behaviours attribute may have some level of connascence of execution (but since it's already implemented beforehand, no need worry)

2) If she is not killed, she will vanish after 30 turns. Mambo Marie will keep coming back until she is killed.

a) Create a new VanishAbleActor interface.
b) Create a new VanishActorAction class that extends the Action class.
c) Create a new VanishGround class that extends the Dirt class.
d) Group these 3 classes into a vanishcapabilities package.
e) MamboMarie implements a VanishAbleActor to allow MamboMarie to vanish after 30 turns.
f) Inside the overridden playTurn method, check whether the playTurns attribute is more than 30. If so, create a new VanishActorAction instance which will remove MamboMarie from the current GameMap and place it in a new VanishGround instance. Set the ground of the Location MamboMarie was last seen using this VanishGround instance.
g) Inside the VanishGround class, override the tick method to check whether MamboMarie wants to remain vanished by reusing the VanishActorAction, which overrides the getNextAction method to check whether the VanishAbleActor wants to remain vanished.
h) If not, she will reappear at a random Location of the GameMap it was in before it vanished.
i) Designed it this way because:

3) For the Bonus Mark question, we decided to implement a Brewery, which is a shop that brews potions and gives these potions freely to any actor within close proximity to it. We made changes to the initial design suggested, which was to only brew and give out potions based on their rarity to instead just brew and give out a random potion.