Design Rational

FIT2099 Assignment 1 - Live and Let Design: Team DirtyBytes

Doors and Keys

Door is an extended class of Ground class in the edu.monash.fit2099.engine package. A door can be passed through if the player has the Key to it. Key is a subclass of the class Item in the package edu.monash.fit2099.engine which is created by the DropItemAction of the Enemy abstract class. DropItemAction Is an extension of the Action class in the engine package, which implements the ActionFactory interface – once an enemy is defeated, a key is dropped by them and the player could pick it up and use it to pass through a door.

- Constructors
    - Door's constructor takes a character as its argument which represents its display character on the user interface. This will then be passed to its parent class's constructor.
    - Key takes a String and a character as arguments for its constructor. The String represents the name of the Key and the character represents the display character of the Key on the user interface – these will then be passed to the constructor of its parent class.
- Methods
    - The Door has a few methods, such as getDisplayChar() – returns the display character, allowableActions() – returns an empty Action list for the class, moveActorAction() etc. which inherit from the parent class.
    - The class Key too inherits a few methods from its parent class such as newInventoryItem() – allows the Key to be dropped, picked up and be placed in the inventory.

- The '**Don't repeat yourself**' principle is adhered here as we extend both of these classes from classes in the engine, and also for an instance, we have the Enemy abstract class which prevents repetition of code for actions like dropping a key when an enemy is defeated, since any enemy could drop a key when defeated.

Goon

Goon extends from the abstract class Enemy, which inherits the Actor class from the edu.monash.fit2099.engine package. It has a FollowBehaviour class which implements the ActionFactory interface, allows Goon to follow the player all around the map. Two action classes – AttackAction and InsultAction which inherit from the Action class, are used to allow Goon to attack and insult the player.

- Constructor
    - Goon's constructor takes a String as its argument which represents its name and a character which represents its display character on the user interface. These will then be passed as arguments for its parent class's constructor along with the default priority and hitPoints.
- Methods
    - Goon has a few methods such as getDisplayChar() – returns the display character, getInventory() – returns a shallow copy of the player's inventory, addItemToInventory(), removeItemFromInventory() – add and remove items from the inventory respectively, isConscious() – check if the actor is conscious, etc., which inherit from its parent class.
- The principle '**Don't repeat yourself'** can be seen here as the Goon class inherits from the Enemy class, while Enemy is an extension of the Actor class from the edu.monash.fit2099.engine package, resulting code to be not repeated, reusable and consistent in creating instances that has the same set of properties while having the freedom to extend the system.

Ninja

Ninja also extends from the abstract class Enemy, which inherits the Actor class from the edu.monash.fit2099.engine package. It has a ThrowPowderBehaviour class and a MoveAwayBehaviour class which implements the ActionFactory interface, allows Goon to throw a bag of stun powder at the player and move one space away from player respectively.

- Constructor
  - Ninja's constructor also takes a String as its argument which represents its name and a character which represents its display character on the user interface. These will then be passed as arguments for its parent class's constructor along with the default priority and hitPoints.
- Methods
  - Ninja too has a few methods such as getDisplayChar() – returns the display character, getInventory() – returns a shallow copy of the player's inventory, addItemToInventory(), removeItemFromInventory() – add and remove items from the inventory respectively, isConscious() – check if the actor is conscious, etc., which inherit from its parent class.

The '**Don't repeat yourself'** principle can be seen here too as the Ninja class inherits from the Enemy class, while Enemy is an extension of the Actor class from the edu.monash.fit2099.engine package, resulting code to be not repeated, reusable and consistent in creating instances that has the same set of properties while having the freedom to extend the system.

Q

Q is a subclass of NPC abstract class which inherits from Actor class. It has a WanderBehaviour class which implements ActionFactory interfaces, allows Q to wander around the map at random. Two Action classes – TalkAction and GivePlanAction, which inherit from Action class were created to allow Q to perform Talk and Give plans to player.

- Constructor
    - The Q constructor takes a String as its argument which represent its name. The name will then be passed as the argument of its parent class's constructor along with the default displayCharacter, priority and hitPoints.

    - NPC has a list of ActionFactory object as its attributes and an addBehaviour method. As Q is a subclass of NPC class, so we can use the addBehaviour method which inherited from NPC class to add WanderBehaviour to ActionFactory. We will do this right after calling the parent's constructor.

- Method
    - As Q is consider as a subclass of Actor class, so it will definitely have an inherited method called playTurn(…). We will override this method to decide Q's action by calling the getAction(..) method from all the ActionFactory subclasses.

- The '**Don't repeat yourself**' principle can be seen here as we do not recreate the constructor, default attributes and methods for Q. Instead, we are reusing the constructor and methods from its parent class. In some cases, it allows us to override the parent's method for its own use.

Miniboss: Doctor Maybe

Doctor Maybe is a subclass of Miniboss abstract class, while Miniboss is a subclass of Enemy abstract class. Doctor Maybe will be placed inside of a locked room and does not move at all. It will attack player when both of them are neighbours. The reason of creating a Miniboss abstract class is to make the system more extensible if there is a need to apply some extra attributes or methods to it. Such as, every Miniboss would have different behavior / actions.

- Constructor
  - The Doctor Maybe constructor takes a String as its argument which represent its name. It would be passed as one of the parameters of its parent's constructor.

- Method
  - playTurn(..) method from the parent's class need to be overridden in order to performs Doctor Maybe actions. We can achieve this by referring to the argument of playTurn(..), an instance of Actions.
    - If instance of AttackAction class is in Actions object, playTurn(...) will return the AttackAction's instance.
    - Otherwise, Doctor Maybe will skip the turn (return instance of SkipTurnAction).

  - We could override getAllowableActions(..) method which was inherited from parent's class so that only instance of Player will be given the "Permission" to attack Miniboss object. (By returning instance of AttackAction class) **or just override this in Enemy class (preferred)** .

- The principle '**Don't repeat yourself'** can be seen here as the Actor class has been inherited to create Enemy class. Miniboss class inherits from Enemy class. Doctor Maybe class inherits from Miniboss class. This ensure that code is reusable, not repeated and consistent in creating an object that has the same property while having the freedom to extend the system.

Building a rocket

Building a rocket is the player's goal. A rocket can be built only when player has placed rocket body and rocket engine on the rocket pad. Rocket body and rocket engine would be an instance of item, while the rocket pad is a subclass of Ground.

We implemented RocketPad as a subclass of Ground because RocketPad has all the properties of Ground but with some extra features.

We will mostly focus on override the allowableActions(…) method from the parent's class, so when one of the arguments of allowableActions(..), instance of Location class, has rocket body and rocket engine on top of it at the same time, allowableActions(..) will return an instance of Actions. In this case, one of the element of Actions instance would be BuildRocketAction, which performs some process of rocket building.

- The principle '**Don't repeat yourself**' can be seen here as Ground class has been inherited to create RocketPad class, Action class has been inherited to create BuildRocketAction class. This ensures that code is reusable, not repeated and consistent in creating an object that has the same property.