Software Design Documentation

Group Name: Pikachu
Members: Jun Qing Lim, Nicholas Loong Kah Fai

Overview

This documentation includes the classes of the extended system, the roles and responsibilities of the new classes, and the interaction between classes in the existing and extended system. It covers the extended system of class diagrams, sequence diagrams and the rationale behind the design. Furthermore, the documentation covers the design principles that the classes adhere to while fulfilling its functionality. This documentation is best understood by reading it and looking at the class and sequence diagrams side-by-side.

Extended Components

Each extended component describes the extended classes and how they interact with the existing classes.

1. Doors and Keys
2. New Enemy: Goon
3. New Enemy: Ninja
4. Non-Player Character: Q
5. Miniboss (Enemy): DoctorMaybe
6. Rocket Building

Doors and Keys

1. **LockedDoor**
   LockedDoor will be an extension from *Ground*, because the *LockedDoor* is considered as a form of terrain type. It will override the *canActorEnter* method to prevent actor from entering the door and actor can only enter the door if there is a key in its inventory because the *LockedDoor* is a form of an obstacle to the player unless the player owns a key. This means that the *LockedDoor* will also override the *allowableActions* method and accesses the items in the inventory of the actor next to the door to check if the actor has the key. If a key is found in its inventory, *allowableActions* will add a new instance of the *UnlockDoorAction* to the list of returned actions. Hence, when the player is next to the door, the player is able to have a choice of unlocking the door.

   In addition to unlocking the door, this *LockedDoor* also overrides the *blockThrownObjects* method and returns true to block thrown objects as it would not be logical for an enemy to throw objects through a locked door. For example, if an enemy (say *Ninja*) wants to throw the stun powder in the direction of the *LockedDoor*, then the enemy will not be able to throw, because there is an obstacle (*LockedDoor*) that blocks the object from being thrown.

   LockedDoor fulfills the *Open-Closed* principle because it extends from the *Ground* and overrides the methods of the *Ground* class, and does not actually modify the actual implementation of *Ground*. This creates greater flexibility for future usages. In addition to that, *LockedDoor* also fulfills the *Single-Responsibility* principle because it operates within the scope of what a door could do, such as it “allows a person to unlock the door with a key” but it “does not actually unlocks the door for that person”.

2. **UnlockDoorAction**

   UnlockDoorAction will be an extension from *Action* abstract class. It stores the attributes of a String of direction (to detect the direction of the door to unlock), this is because in the case where if there are two doors facing each other, the player would have a choice to select the door to unlock (whether if it's the west door or east door), this creates a greater flexibility for the player. It also stores the *Location* of the door to allow the location of the door to be replaced by a type of terrain - *Floor*. Thus, when the player unlocks the door, the door disappears and is replaced with a *Floor* type. In addition to that, it also has *Key* as its attribute, because when the player executes this action, the key must be removed from the player's inventory. So, the key is stored in the class so that the *execute* method can call the *removeItemFromInventory* method to remove the key from the player's inventory.

   Each instance of *UnlockDoorAction* can only have exactly one *Location* and one *Key* stored as its attributes, so the multiplicity of UnlockDoorAction to Location and Key is one.

   UnlockDoorAction fulfills the *Single-Responsiblity* principle, because it only unlocks the door and does not do anything else. The action of unlocking the door happens within the scope of "removing the door and taking the key away from the user", and it does not actually "checks if the user has the key".

3. **Key**

   Key will be an inheritance from *Item*, because *Key* does everything that an *Item* does. Inheritance is used so that we are able utilize the existing methods in the *Item* class and prevent duplicated codes. *Key* is only dropped by enemies, therefore, when a *Key* is instantiated, its attribute of *allowableActions* is cleared and added with only *DropItemAction* to prevent the *Key* from allowing the Actor the holds the key to perform other actions with the *Key*. Therefore, the enemies are only able to drop the *Key*.

4. **Enemy**

   Enemy abstract class extends from *Actor*. As all enemies (*Grunt*, *Goon*, *Ninja* and *DoctorMaybe*) are able to drop key(s) when it is knocked out (or died), Enemy abstract class represents the parent class of all enemies. This means that the default implementation for every enemy is that *Key* is added into the enemy's inventory automatically (refer to *Enemy* abstract class below for details). As each enemy must have at least one key in its inventory, it implies that each enemy can have more than one key. So, the multiplicity of *Enemy* to *Key* is one to many.

New Enemy: Goon

1. **Enemy**

   Enemy is an abstract class extends from the *Actor* class. *Enemy* extends from the *Actor* class because an *Enemy* is a form of an *Actor* and it is an abstract class because we do not want an instance of the *Enemy* class to be instantiated. *Enemy* is also not an interface because the damage of all the enemies are interrelated to one another therefore the damage is required to be stored in the *Enemy* class as its attribute. This fulfills the *Don't Repeat Yourself (DRY)* principle and *Avoid Excessive Use of Literals* as the damage would not be repeated and are not hard coded thus making maintenance of the code easier.

   The *Enemy* class is extended by several classes which are *Grunt, Goon, Ninja* and *DoctorMaybe*. These classes all share a similar attribute in the game which is that they are all different forms of enemies available in the game. The *Enemy* class stores a base damage that is shared by all enemies in the game and the base damage is based off the enemy *Grunt* as the damage of the other enemies are different variations of the *Grunt*'s damage. In other words, *Grunt*'s damage would be solely based on the base damage while other enemies would varies accordingly. For instance, *Goon*'s damage will be double of the base damage (because base damage is also shared by *Grunt*). In short, doing so adheres to *Reduce Dependencies (RED)* principle as the different enemies are no longer dependent on the existence of the *Grunt* class.

   Enemy also stores a list of *ActionFactory* so that each enemy can have its own behaviour (if needed). This means that in the case where if *Goon* has a specific new behaviour (for instance, teleporting), then *Goon* is by default able to utilize the existing list and just adding it. This makes the code to be much flexible as it does not need to modify a lot if new implementations are added. The access level modifier for this list of *ActionFactory* will be set as *private* to create a tighter control over the access over the list, so a separate *addBehaviour* method with a modifier of *protected* is created to allow only sub-classes that inherits from this *Enemy* class to be able to add the behaviours into the list. Due to the fact that each enemy might have different behaviours or no behaviours at all, so the multiplicity of Enemy to ActionFactory is zero to many.

2. **Goon**

Goon is an extension of the *Enemy* abstract class because *Goon* is a form of enemy in the game. *Goon* does double the damage of *Grunt's* in the game and is able to shout an insult to the player. Therefore, *Goon* will be utilizing the base damage from Enemy class and overrides the *getIntrinsicWeapon* method and returns a new instance of the *IntrinsicWeapon* that has doubles the base damage.

Goon adds a *FollowBehaviour* to the list of *ActionFactory* (that was made in the *Enemy* class) as the *Goon* follows the player as the player moves around. The assumption made in our design is that Goon can only follow the player and there is only one player in the game. Hence, the multiplicity shown for Goon to *FollowBehaviour* is one. However, in the case where if there are more players in the game and if *Goon* could follow more than one player, then the multiplicty will be changed, but the overall implementation will not be changed (because behavious are stored in the form of a list).

Next, the *Goon* would implement a *TalkAction* because the *Goon* has the ability to shout an insult to the player during the game. Hence, it overrides the *playTurn* method that has a 10% chance of calling the *TalkAction* with an insult message. *TalkAction* is used because the action of shouting an insult does not have any extra effects on the player other than the *Goon* itself speaking to the player (refer to *TalkAction* class for details).

In the case where if Goon is unable to follow the player and unable to pass the 10% chance of returning the *TalkAction*, then Goon should moves randomly. That implies that Goon will execute the *playTurn* method of the parent class through the "super" keyword. This is because the default implementation of *playTurn* is moving randomly, so instead of re-creating the implementation where it fails to insult player, we could just utilize the implementation of the parent class. In other words, it fulfills the *Don't Repeat Yourself (DRY)* principle as code duplication is prevented.

New Enemy: Ninja

1. **Ninja**

   Ninja is an extension from the *Enemy* abstract class. It stores the *ThrowStunBehaviour* in the list of *ActionFactory* because a *Ninja* is able to stun the player if the player is within range. *Ninja* also stores the *Actor* as its attribute because *Ninja* is able to stun player only so it has to differentiate the player from other actors. In addition to that, storing *Actor* as its attribute allows the *Ninja* to be able to access the player's location from time to time. As each *Ninja* stuns exactly one player at one time, so the multiplicity of *Ninja* to *Actor* is one. The assumption made here is that each *Ninja* has the capability of stunning only one player, so in the case in the future where if more players are needed to be stunned, so extra ninjas will be spawned in the game.

   Ninja also overrides the *playTurn* method from the *Actor* class. During *Ninja's* turn, it has to check if it has thrown the *StunPowderBomb* to determine its action for that particular turn. For instance, after the *Ninja* has stunned the player, it has to check whether it has thrown the stun in its previous turn to avoid throwing the stun on the player again. If it has thrown, then it must move one space away from player. Due to the scenario where *Ninja* is located at the top of the map and stunned the player, therefore *Ninja* is unable to move "exactly one space away from player" (because it reaches the top of the map). Therefore, in our design, we have refactored it to the extent where *Ninja* moves one step in a random direction (so that even if it reaches the corner of the map, it is still able to make a move).

   On the contrary, if *Ninja* has not thrown the stun, that means that *Ninja* is in hide (or unable to see the player), so *Ninja* will only execute the *SkipTurnAction* to remain at the same place.

2. **ThrowStunBehaviour**

   ThrowStunBehaviour extends from *Action* class and implements the *ActionFactory* interface. It stores the attribute of the *Actor* that represents the target to be stunned. This means that the constructor of *ThrowStunBehaviour* would need to have an argument of *Actor* to be instantiated. Each behaviour of throwing a stun can aim only at exactly one player, where the same stun cannot be thrown to two different players at one time. So, the multiplicity of *ThrowStunBehaviour* to *Actor* is one.

   As *ThrowStunBehaviour* implements the *ActionFactory* interface, this means that it overrides the methods (which is the *getAction* method) in the *ActionFactory* and implements its own functionality, which in our case, to get the action of throwing the stun, it has to check whether the target is within 5 squares of the caller. In particular, the *getAction* method is going to use the *Range* class and the *Location* for both *Ninja* and *Player* to check whether the player and ninja are on the same row or same column (with a distance of five steps). In addition to that, this *getAction* method also takes in account whether there are any obstacles that is going to prevent the *Ninja* from throwing stun.

   In our design, assuming that the ninja is able to throw the stun, then *Ninja* will execute the *ThrowStunBehaviour* methods that are overridden from the *Action* class (because *ThrowStunBehaviour* implements *ActionFactory* interface and extends the *Action* class, so it has the ability to override the methods in *Action* class). The overridden *execute* method has a 50% chance of success. If succeeded, it first checks if the player is stunned. If it is not stunned, it will add an item of *StunPowderBomb* onto the player's location (this *StunPowderBomb* will have a stun effect that prevents the player from any actions for two turns, which will be checked by the *GamePlayer* class, refer to *GamePlayer* for more details).

3. **StunPowderBomb**

   StunPowderBomb is an extension from the *Item*. This item does not perform any actions, in other words, it performs like a "furniture" or static item, where the item cannot be picked up. The purpose of this item is to act as a "stun bomb", if the player detects this bomb in its location, then it cannot do anything other than skip its turn (details under *GamePlayer*).

4. GamePlayer

The current implementation of *Player* does not allow for any modification (as it is in the engine class). An extension of the *Player* class - *GamePlayer* is created so as to allow the overriding of *playTurn* method. *GamePlayer* class stores an integer attribute of stunCount to keep track of the number of stuns, which can be illustrated as the following.

When the player is executing its turn, it has to check whether it is being stunned. If it's stunned, then he is unable to perform any actions except the *SkipTurnAction*. This implies that the checking of the stun count must be done in the *GamePlayer*'s *playTurn* method itself, because the *playTurn* method returns the action to be executed during that turn. Therefore, this overridden *playTurn* method is going to first check if the *StunPowderBomb* exists on its location. If it exists, then it returns only *SkipTurnAction* and increases the *stunCount* by 1 (to prevent player from moving while other actors execute their turn). Subsequently, after the next few turns, if the *stunCount* has reached 2, then the *StunPowderBomb* will lose its effect and it will be removed from the player's location and player would as a result able to continue executing its move (because the *StunPowderBomb* is no longer there).

On the contrary, if the *StunPowderBomb* does not exist, the *playTurn* method is going to utilize its parent class of *playTurn* through the "super" keyword, because the player is not stunned and thus implies that the player should executes its default implementation.

This extension of *Player* means that in the *Application* class, *Player* is no longer instantiated but *GamePlayer* being instantiated instead. Doing so ensures that the entire game is not being affected after added a new *GamePlayer* (for overriding the *playTurn* method to check the stun effect).

Non-Player Character: Q

1. **Q**

   Q will be a class that inherits from *Actor*. As Q needs to give the rocket body to the player, *Q* stores a *RocketBody* in its inventory. This means that by default, when *Q* is constructed, a new instance of the *RocketBody* will be added into the *Q*'s inventory list automatically. Due to the reason that *Q* disappears from the map after passing the *RocketBody* to player, so it makes sense to say that *Q* has only one *RocketBody* in its inventory. Thus, the multiplicity of *Q* to *RocketBody* is one.

   Q will also be using *TalkAction* to talk to the player when the player first talks to *Q*. Therefore, *Q* will be overriding the *getAllowableActions* method and checks whether the player has *RocketPlans* in its inventory. If it does, it allows the player to either talk to *Q*, which *Q* will ask the player to hand over the *RocketPlans* or to pass the *RocketPlans* to *Q*. Based on this scenario, the *getAllowableActions* method will then return a list of actions that includes the *TalkAction* (with the message above) and *GiveItemAction* (that pass the *RocketPlans* to *Q*'s inventory). On the contrary, if the player does not have the *RocketPlans*, then the *getAllowableActions* method will return a list of actions in addition to a *TalkAction* with a "need rocket plans" message.

   Assume that player has given the *RocketPlans* to *Q*, *Q* has to give the *RocketBody* in return. This action of handing the *RocketBody* is considered as *Q*'s action to ensure that only *Q* class is responsible for the properties of *Q*. Therefore, *playTurn* method is overridden to check if *RocketPlans* exists in *Q*'s inventory (because after player has executed *GiveItemAction*, the item - *RocketPlans* will be added into the *Q*'s inventory). And if *RocketPlans* exists, it is going to iterate through the *Q*'s inventory and pass the *RocketBody* to player through the *GiveItemAction* class. If this action is executed, *Q* will then disappear from the map (in the next turn) by calling *removeActor* method from the *GameMap* class to remove itself from the map. On the other hand, if it does not detect any *RocketPlans* in its inventory, this means that player has not passed any item and *Q* continues to wander around the map, therefore *Q* will execute the *MoveActorAction* and moves itself in random direction.

2. **GiveItemAction**

   GiveItemAction will be extending from the *Action* abstract class. It stores the *Item* and *Actor* (known as target) as its attributes, this allows the *GiveItemAction* to override the the *execute* method and adds that item into the target's inventory. At the same time, when the item is added into the target's inventory, the item is also removed from the actor's inventory (which is the caller's inventory). Such design creates greater flexibility because if the player wants to give the item to *Q,* the player can call this *GiveItemAction* and the item to be passed to *Q* will be automatically removed from the player's inventory and added into *Q*'s inventory, this concept is also applicable when *Q* is passing the *RocketBody* to *Player*. Therefore, the function of giving an item is not only binded to this specific scenario and can be reused for other scenarios where other actors are required to give an item to another actor.

   GiveItemAction is solely responsible for giving one item to one actor at one time, where the item to be passed to the actor will be removed from the caller's inventory. Therefore, the multiplicity of *GiveItemAction* to *Item* and *Actor* is exactly one.

3. **TalkAction**

   TalkAction will be extending from the *Action* abstract class. *TalkAction* will store a String of message and an *Actor* to talk to. Each *TalkAction* is responsible for talking to a single person, so the multiplicity for *TalkAction* to *Actor* is one. When executed, it will print out the message with the name of the actor. For instance, when *Player* talks to *Q*, *Q* will print out a message like this, "Q: I can give you something that will help, come back to me when you've the plans.". This *TalkAction* also creates a greater flexibility through the usage of *Actor* as part of its attribute, in other words, other actors are also able to call this class. For instance, when *Goon* insults, *Goon* will call this class and pass its insult message so as to print them out accordingly thus allowing this class to be reused for different scenarios.

4. **RocketBody**

   RocketBody will be extending from *Item*. Only *Q* has the ability to create *RocketBody* because the *RocketBody* is only available through *Q* hence this *RocketBody* will be immediately transferred to the actor's inventory when traded with *RocketPlans*. However, the *RocketBody* class is not dependent on the existence of *Q* to adhere to the design principle *Reduce Dependencies (RED)* and to allow greater flexibility to the design where the *RocketBody* is available through another actor. Putting it differently, when the *RocketBody* is placed in the actor's inventory, the *RocketBody* is only able to drop itself, thus when the *RocketBody* is constructed, its attribute of *allowableActions* will be cleared and added with a new initialization of *DropItemAction*. With this design, the player has the ability to drop the *RocketBody* from its inventory without interfering with the *RocketBody*'s class properties.

5. **RocketPlans**

   RocketPlans will be extending from *Item*. As the *RocketPlans* is placed in a room with a locked door, our design assumes that by default, during the creation of the map, the *RocketPlans* is already in the locked room and the player is able to pick it up, so *RocketPlans* must have the *PickUpItemAction* in its *allowableActions* attribute. However, as by default, all constructed items have *PickUpItemAction* (because when an *Item* is constructed, it has *allowableActions* of *PickUpItemAction*). Therefore, the extended *RocketPlans* would only need to use the existing *Item* class through the "super" keyword. Such design adheres to *Don't Repeat Yourself (DRY)* principle as it makes use of the properties of the parent class through the "super".

Miniboss (Enemy): DoctorMaybe

1. **DoctorMaybe**
   DoctorMaybe is an extension from *Enemy* abstract class. It has half the damage of *Grunt*'s. Based on our assumption that *Grunt* uses the base damage in the *Enemy* abstract class, so *DoctorMaybe* will override the *getIntrinsicWeapon* method and instantiate a new *IntrinsticWeapon* with the base damage being halved.

   DoctorMaybe drops *RocketEngine* when defeated thus it will store *RocketEngine* in its inventory when it is constructed to ensure that *RocketEngine* is not dependent on *DoctorMaybe* and is merely just an item added to it. The *DoctorMaybe* constructed in our assumption is that it can has only exactly one *RocketEngine* to be dropped. So, the multiplicity of *DoctorMaybe* to *RocketEngine* is one.

   DoctorMaybe does not move at all and it is placed in a locked room. The assumption made in our design is that, by default, in the *Application* class, *DoctorMaybe* is being instantiated and placed in a room with a *LockedDoor* which can only be unlocked with a key. To prevent *DoctorMaybe* from moving, *playTurn* method is overridden to remove the *MoveActorAction* from list of actions. At the same time, *DoctorMaybe* will also perform *AttackAction* if it detects the actor next to it is a *Player*.

2. **RocketEngine**
   RocketEngine is an extension from *Item*. *RocketEngine* can only be dropped by *DoctorMaybe* when it is knocked out. So, when *RocketEngine* is constructed, its attribute of *allowableActions* will be cleared and added with an instance of *DropItemAction* for dropping itself to ensure that the *RocketEngine* is responsible for its own properties. Doing so will allow the *DoctorMaybe* to be able to drop this item off from its inventory when it is defeated while adhering to the *Single-Responsibility* principle.

Rocket Building

1. **Rocket**
   Rocket is an extension from *Item*. It will only be called / instantiated in the *BuildRocketAction* (as it builds rocket). In our design, rocket is not able to move, so when rocket is constructed, it will clear the attribute of *allowableActions* such that this item does not have any *PickUpItemAction* or *DropItemAction* (as rocket is considered as a static *Item*, which does not move).

2. **BuildRocketAction**
   BuildRocketAction is an extension from *Action* abstract class. To build a rocket, it needs to have a rocket body, a rocket engine as well as the location for building it. Hence, in accordance to this logic, *BuildRocketAction* is going to store *RocketBody*, *RocketEngine* and *Location* (known as the location of the *RocketPad*) as its attributes. So, it implies that the multiplicity of *BuildRocketAction* to *RocketBody*, *RocketEngine* and *Location* is exactly one. Furthermore, the *BuildRocketAction* constructor will have three arguments (which are those three attributes stated above). This creates a tighter control over when this action is able to be instantiated (refer to *RocketPad* on instantiation requirements).

   In addition to that, the *execute* method will also be overridden to remove the *RocketBody* and *RocketEngine* from the location of the *RocketPad* (because this class can only be instantiated when the player has dropped *RocketBody* and *RocketEngine* onto the *RocketPad*). It then adds a new instance of the *Rocket* class onto the *RocketPad*.

   BuildRocketAction fulfills the *Single-Responsibility* principle because it only knows how to build a rocket when it is given the rocket body, rocket engine and the location to build.

3. **RocketPad**
   RocketPad is an extension from *Ground*. The assumption made in our design is that *RocketPad* exists on the map of the game as a location at the start of the game. This *RocketPad* will be overriding the *allowableActions* method to allow the user to call the *BuildRocketAction*. In particular, this *allowableActions* method checks if *RocketBody* and *RocketEngine* exists on its own location. If they do exist, then *allowableActions* method returns a list of possible actions in addition to the *BuildRocketAction*. Doing so, it gives the player a choice to select whether to build the rocket or to move to a different direction (after dropping the engine and body). In the meantime, as *RocketPad* is inherited from *Ground*, so player would not be able to build the rocket while standing on the *RocketPad*, so the player would have to be one step away from the *RocketPad* to have the option of building the rocket from the menu.