MONASH University

FIT2099: Object Oriented Design and Implementation

Assignment 2:

Design Rationale

Hoang Phan
Josephine Leye

Introduction

This design rationale will outline the new classes that our team has decided are necessary to fulfill the project requirements and why, as well as how these classes will be implemented into the existing system, and as a result, how it will interact with the existing system to achieve the required functionalities.

New classes

In this section we will outline the new and important classes that will need to be implemented as part of our new system. Some of them will be extended from already existing classes in the game engine package.

BuilRocketAction: Extends the existing Action class. This will check if the current location where the action is called is the RocketPad, contains the RocketEngine and RocketBody. If all these specifications are met, the method will add a Rocket item to the location, and remove the player, thus ending the game.

GivePlanAction: Extends the existing Action class. This will check if the Player has a RocketPlan item, this will be removed and replaced with a RocketBody, and Q will be removed from the GameMap. If not, a message will be printed to collect the RocketPlan.

Goon and Ninja: Similar to the already existing Grunt class, both the Goon and Ninja class will extend the provided 'Actor' class already in the system and as such will possess the same methods and interactions as the existing Grunt class, the difference being which behaviours will be added from the ActionFactory.

InsultBehaviour: Extends the already existing Action class and implements the existing ActionFactories interface. This action will at random attack or miss the player, on attack the actor has a 10% chance of shouting an insult on top of the attack, it also checks if the target is conscious and replaces it with a "SleepingActor".

Key: A Key is an extension of the Item class, and can be added as a new Inventory Item to any Actor. It can be dropped or picked up by any extension of the Actor class via the DropItemAction and PickUpItemAction. An Actor must have this key in order to pass a Door to enter a Room on the GameMap.

LockedDoor: Extended from the existing Ground class. This will act as an obstacle preventing the Player from entering a Room by returning False on the Ground canActorEnter method. It can only be unlocked if the Player has UnlockDoorAction as an allowable Action. (See UnlockDoorAction).

MiniBoss: Similar to the Goon and Ninja, MiniBoss is an extension of the existing Actor class. He will not have any additional behaviours added, and will not move around the GameMap. He will have in his inventory the Item RocketEngine to start with, which can be dropped upon defeat by the appropriate actions.

Q: A unique NPC that the player can interact with. It will be an extension of the existing Actor class. It will have access to GivePlanAction and TalkToQAction when a player is next to it in order to replace a Players RocketPlan with a RocketBody.

RocketEngine: A RocketEngine is an extension of the Item class and can be picked up by any Actor via theDropItemAction and PickUpItemAction. An Actor must have this RocketEngine in order to use the BuildRocketAction to build a Rocket item and end the game.

RocketPad: A special location that the player must find in order to build the rocket. Extended from the existing Ground class. Will have BuildRocketAction as allowable Actions for its location.

StunnablePlayer: Extends the already existing Player class. The same as a player but supports the stun effects of Ninjas. This class holds methods to check the stun count of a player and skip the turn if it is greater than zero.

StunAndMoveBackBehaviour: Extends the already existing Action class and implements the existing ActionFactory interface. This represents the behaviour of a Ninja. It allows the Ninja to remain in one place until a Player is in range. Once the Player is in range, there is 50% chance of hitting the target, if this occurs, the target skips the next two turns (is stunned). The action is blocked by walls and doors.

TalkToQAction: Extends the already existing Action class. Like the GivePlanAction, this will check the Players inventory for the RocketPlan, if it is there, a message will print to hand the plans over, if not message will print that you need the plans.

UnlockDoorAction: Extends the existing Action class. This will check if Player has a key via the Actor getInventory method. It will replace the Door at the location with Floor which can be traversed by the Player to enter the Room, otherwise it will print a message to find a Key.

WanderBehaviour: Implements the existing ActionFactory interface. This will find all possible Exits for an Actor at a Location and will at random choose one as its destination. If there are no allowable exits the Actor will skip its turn via the SkipTurnAction.

By dividing the system into separate classes such as the ones above, we will be able to divide the work amongst the team, so that classes with large dependencies are coded by the same team member.

Interactions between classes

The file FIT2099_Communication_Diagram_Assignment2 in our git repository contains 5 different communication diagrams, each outlining a different complex interaction within the new system we are looking to implement. These diagrams are described in the section below.

Diagram 1: Unlocking a Room

This diagram walks through the interaction a Player must go through in order to gain access to a Room. Once the player is next to a LockedDoor, the getAllowableActions method will return UnlockDoorAction. This will check if the Player in range has a Key item through the getInventory method from the Actor class. If the Player is in possession of a key, the UnlockDoorAction will replace the current LockedDoor Location on the GameMap with a Floor, which can be traversed by the Player in order to enter the Room. The action will also remove the Key from the Actor via the Actor method removeItemFromInventory.

Diagram 2: Defeating an enemy

This diagram walks through the course of action after the defeat of an enemy. Firstly, upon encountering an enemy (Ninja, Goon, Grunt, MiniBoss), the Player will execute the AttackAction action, which will cause damage to the enemy. Once this action is performed, the Actor class should have a boolean method to check whether it is conscious, isConscious(), if this returns false, the enemy will drop a Key using DropItemAction(). From this, the Player will pick up the Key using PickUpItemAction().

Diagram 3: Creating the Rocket

This diagram walks through the steps that must be taken in order to create the Rocket. First once the Player is next to the RocketPad, getAllowableActions will return the BuildRocketAction on the Players PlayTurn. The BuildRocketAction will retrieve the RocketBody and RocketEngine from the RocketPad Location and will then create a Rocket Item and add it to the GameMap using addItem method. The action will also remove the Player from the GameMap using the removeActor method, thus ending the game.

Diagram 4: Interaction with Q

This diagram walks through the interaction between the Player and Q in order to retrieve the RocketBody in exchange for the RocketPlan. Once a Player is next to Q, the allowable actions will be TalkToQAction or GivePlanAction. If the Player selects TalkToQAction this will use getInventory method from Actor to check for the RocketPlan and react accordingly. If the GivePlanAction is selected, once again getInventory will check for RocketPlans and can then remove the RocketPlans with removeItemFromInventory and replace it with the RocketBody using the addItemToInventory method. Q will then be removed from the GameMap via removeActor method.

Diagram 5: Ninja's interaction with player

This diagram walks through the course of action between the Player and the Ninja class. Firstly, on every turn the Ninja will check if the Player is in range , if yes the getAllowableActions will return StunAndMoveBackBehaviour when the Ninja PlayTurn is executed. The StunAndMoveBackAction has a 50% chance of hitting the Player, if successful, it will increase the StunnablePlayer stun count, using the increaseStunCounter method, to 2 which will result in the SkipTurnAction being returned for the Players PlayTurn for the next 2 turns.