Assignment 3 Design Rationale
Requirement 1: The Ship of Theseus

Implementation

Newly created class

Teleporter: A class which extends the Item abstract class and implements the Purchasable interface and has name “Theseus”. The base price is declared to be 100. It overrides the allowableActions function and adds a MoveActorAction to it. Interacts with Location to fetch a random position and with Actor to handle transactions.

Modification for previous created classes

ComputerTerminal class: added a Hashmap<Location, String> to the constructor and all these locations will be created MoveActorActions and added to the actionList to be returned in the allowableActions together with the purchase actions implemented in assignment 2.

Application: created a new Teleporter purchasable item and added to the Purchasable list which will then be passed to the computer terminal. In requirement4, we created two other different game maps and there are three game maps now including the game map used in assignment 1 and 2, Each GameMap instance now potentially includes a ComputerTerminal. Then, create a hashmap of locations and string to indicate the location the player will be moved and the description of this move. Each hashmap is then specifically passed to their own computer terminal and creates moveActorAction in the computer terminal according to each location and string in the hashmap passed to its constructor. There will be three hashmaps and three computer terminals for the three maps having their own one. The locations in the hashmap passed to the computer terminal will exclude itself and having the locations the player will be moved to the two other maps if the player chose to travel. Locations passed to the hashmap is fixed to each map’s spaceship.

Implementation approach

The first goal of this requirement is to create a teleporter item which can be sold by the computer terminal. As it can be purchased from the computer terminal, it should implement the Purchasable interface. And as we have made the computer terminal having dependency injection in the constructor, we do not need to modify the computer terminal in this case to make it able to let players purchase this teleporter. All we need to do is create a new Teleporter in the application and add this newly created purchasable item into the purchasable list which is added to the computer

terminal constructor later. For the feature of this teleporter, once the player purchased it, it should be able to teleport the player to a random location by some steps. First, the player should place it on the ground and a teleport action should be printed for the player to choose if it wants to be randomly teleported, and the teleporter that is placed on the ground should remain at the same location and be able to teleport the player again when the player is back to that location. So, to achieve this, we should use the allowableActions function which is returning a list of allowable actions that can be performed on the item when it is on the ground. As there are different allowable action functions which have different purposes. This function will only execute when the item is placed on the floor. So, when we want the player to be able to be teleported only if it places the teleporter on the ground, we should use this function. As soon as the player has bought the teleporter, and as it is an item, the player should have an option to drop the item. Before dropping the item, the player won't have the teleport option as the item is not on the ground. So once the player chooses to drop the teleporter on the ground, next round the teleporter will execute the allowableActions of giving an option to the player if it wants to be teleported randomly. So basically, after the player purchased the teleporter, the teleporter can be seen as a scrap as the player can always pick and drop the teleporter and bring it along anywhere and drop whenever the player wants to be teleported randomly. So, how will the teleport action be executed? Here, in the allowableActions which we mentioned earlier, we will add a MoveActorAction which is an action already present in the engine package given. We don't even need to create any new action classes to teleport the player, we can just use the location input parameter to get the map and generate a new location and pass it to the MoveActorAction. Which is a very efficient approach to maximum utilise and reuse the created classes. At the same time, to make it able to be purchased, it implements the Purchasable interface and implements the purchase function. At this point, we have completely done creating the Teleporter Theseus! Next feature of this requirement is that the computer terminal should be able to have an option for the player to travel to other maps. Here, we modified the computer terminal constructor to accept a hashmap of <Location, String>. As in requirement 4, we easily implemented the maps by having map patterns as array lists, then created a game map for each of them. We have three game maps of polymorphia, connascence and static factory now. We will fix the location when the player travels to other maps, which should always be in the spaceship, we will create a location for the player to be moved to for each map and indicate where it is to go with a string and add these pairs of locations and strings into the hashmap. Now, as we will have three computer terminals for each map, we can just create locations to be moved(in their own spaceship) for the particular maps. And as the computer terminal accepts this hashmap, in the allowableActions which we previously added purchase action for each purchasable item, we now will loop through the hashmap and create a MoveActorAction for each location. Now, if the player moves near to the computer terminal, the player should be able to purchase items and also two options to travel to the two other maps it is not in! We pass locations instead of the action itself to the computer terminal as we can check some

condition of the location in the computer terminal before creating a action for it but if we pass some specific actions to the computer terminal we might need to do some condition check in the application and also check repeatedly in the application if like in this situation we are creating multiple locations (if we do multiple actions). So, we will pass a location and choose to create / not create actions by checking conditions in the computer terminal itself! Another thing we observed is, in the computer terminal allowable actions which process the hashmap, we are still using the MoveActorAction! We can see how scalable our design is to reuse existing classes and actions! This approach shows how efficiently we can make use of all existing classes as besides creating new Teleporter classes, wwe are reusing all other classes to help achieve our final goal of this requirement!

Concepts and principles applied

1. Don't Repeat Yourself (DRY):

   By leveraging the already existing MoveActorAction, we minimise the code duplication and utilise a well-tested functionality to handle three move actor actions which were being explained in a slightly different way (teleport to random location, travel to different maps, and generally move actor action in the previous assignments). However, it is all the same which is built on top of the execution of moving the actor to a location! So we thought of how to reuse the already existing MoveActionAction which aligns with the DRY principle very well!

2. Single Responsibility Principle (SRP):

   The Teleporter class is solely responsible for handling the teleportation functionality. It encapsulates all the behaviour related to teleportation and purchasing without mixing it with unrelated functionalities.

3. Liskov Substitution Principle (LSP):

   This principle is achieved by the computer terminal by having a list of purchasable items. By implementing the Purchasable interface, the Teleporter class adheres to the contract defined by the interface. This means that any class that uses Purchasable items can rely on the behaviour promised by the interface without worrying about the specific implementation of Teleporter.

4. Interface Segregation Principle (ISP):

   The Teleporter class implements the Purchasable interface. The Purchasable interface defines a clear and specific contract for items that can be purchased. It avoids forcing classes to implement methods they don't need.

5. Dependency Inversion Principle (DIP):

   The ComputerTerminal class depends on abstractions (interfaces and lists of actions) rather than concrete implementations. It receives its dependencies (purchasable items and travel actions) through its constructor.

6. Cohesion and Coupling:

   Each class has a clear, distinct responsibility, enhancing cohesion. The use of interfaces and inheritance reduces tight coupling between components, allowing for easier modifications and additions in the future. For instance, adding another purchasable item or another travel destination involves minimal changes to existing code.

Advantages

1. Flexibility and Scalability:

   New items and actions can be added easily without modifying existing code, allowing the game to grow and evolve over time.

2. Maintainability:

   Clear separation of concerns and adherence to SOLID principles make the codebase easier to understand, maintain, and debug.

3. Reusability:

   By leveraging existing classes and actions (e.g., MoveActorAction), we avoid code duplication and make the most of the already available functionality. This reduces the likelihood of bugs and simplifies future enhancements.

4. Testiblity:

   Dependency injection in the ComputerTerminal class supports better unit testing. We can inject mock objects to test different scenarios without relying on the actual game environment.