SWEN30006 Report Project 1 Pacman

Workshop [Tue17:15] Team 5
1341243 Supakorn Sumongkol
1214247 Xierui Sun
1214289 Shanaia Chen

In order to create a better system that corresponds to the GRASP principles and patterns, we modified the Pacman in the Multiverse design created by Arcade 24™ (A24). We created a domain class diagram in order to capture a better understanding of the current implementation and from there, we developed static design and dynamic design models to provide a basis for our new design. These models can be found at the end of this document.

ItemFactory and MonsterFactory

In the original design, the Game class has quite a large number of responsibilities that range from running the game to creating new instances of monsters and items and then setting them up in the game. While this adheres to the Creator principle since the class contains both monsters and items, the code exhibits high coupling and low cohesion, which makes it less readable and more difficult to extend to new features such as new monster or item types. To improve this, we delegated the tasks of creating items and monsters to an ItemFactory and a MonsterFactory, respectively.

Monster Superclass and Monster Classes: Troll, TX5, Orion, Alien, Wizard

The initial Monster class does not account for possible extensions of monster types in the game. As can be seen in the walkApproach method, it made use of if-else statements to determine the walk approach of each monster. This would be a problem if there are several monsters with various walk approaches, since it could lead to high coupling and low cohesion. Additionally, this could even lead to repeated code if there are similar behaviours between the walk approaches of a few monsters.

For our design, we created a Monster superclass and several classes for the given monster types: Troll, TX5, Orion, Alien, and Wizard. As can be seen in the models, the only difference between the monsters are their walk approaches. Therefore, we have the superclass contain all the common operations and then create the polymorphic operation (walkApproach method) for each class accordingly. If there will be new features for some monsters in the future such as special skills, we could simply add a new method in the classes of those specific monsters. Additionally, since Troll, TX5, and Orion could walk randomly (based on some conditions), we decided to include the randomWalk method to the superclass. With this, we have demonstrated the polymorphism principle and code reuse.

Item Superclass and Item Classes: Gold and Ice

Prior to any modifications, the design places all item operations in the Game class. This makes it have high coupling and low cohesion. Not only is this due to the Game class having other responsibilities as mentioned previously, but it is also because both the gold and ice items have similar operations which leads to repeated code. One such instance is the putIce and the putGold methods. This would become even more complicated if future extensions include new item types, especially if these have different features. Please note that the pill will be discussed in the next section.

To modify this, we created an Item superclass and separate classes for the given item types: Gold and Ice. As with the previous section, the superclass contains all the common operations, and the individual classes contain polymorphic operations. In this case, the Gold class contains methods related to the monster Orion's behaviour as can be seen in the domain class and static design diagrams. Additionally, new items (with different features if required) could be added easily by creating new classes and methods. This makes use of the polymorphism principle and code reuse.

Pill Class

We created a Pill class and separated it from the Gold and Ice items. This is because the code before modification puts them all in one class and so has high coupling and low cohesion. Unlike the situation where Pacman eats gold or ice, eating pills does not influence the behaviour of the monster in this case. Also, pill is not an actor like the other items which can be seen in the given code. By separating the class, we can achieve low coupling and high cohesion for both the Item classes and the Pill class. Thus, it simplifies the code and makes it more readable. Additionally, it makes it much easier to add extensions to items and also add special effects directly to the pills separately.

Game Class

As mentioned above, the original design of the Game class involves storing each monster and item as distinct objects which results in high coupling and low cohesion. Based on the Protected Variation principle of GRASP, we modified the methods in the Game class to create a stable interface (means of access) in order to handle its variation points. By encapsulating the behaviour of Monsters and Items within their respective classes, the design reduces duplication of code and is better suited for extension of new item and monster types.

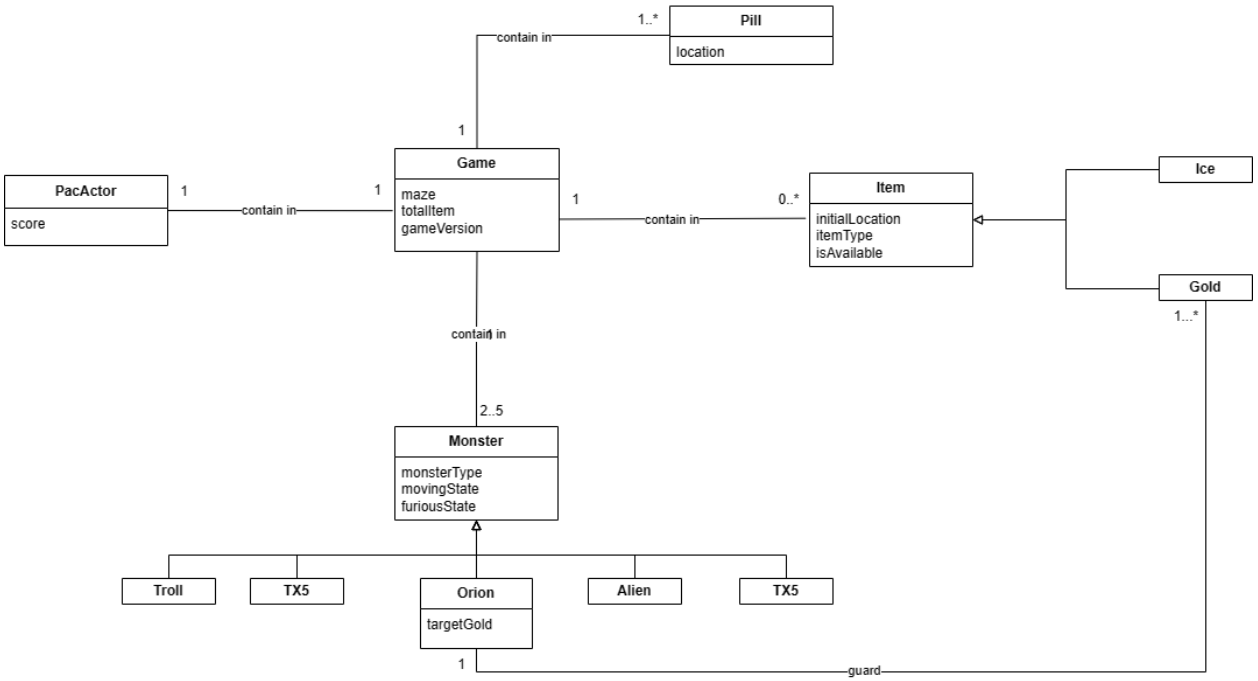
Additional Design Justifications

1. Game Class still preserves GoldList as a distinct object from ItemList, this is because Orion (which is one of the monsters) needs to keep track of these object locations. According to our design, this behaviour only belongs to Orion Class. Thus, we need to find a way to provide a list of all gold to Orion without passing it through the Monster Class.

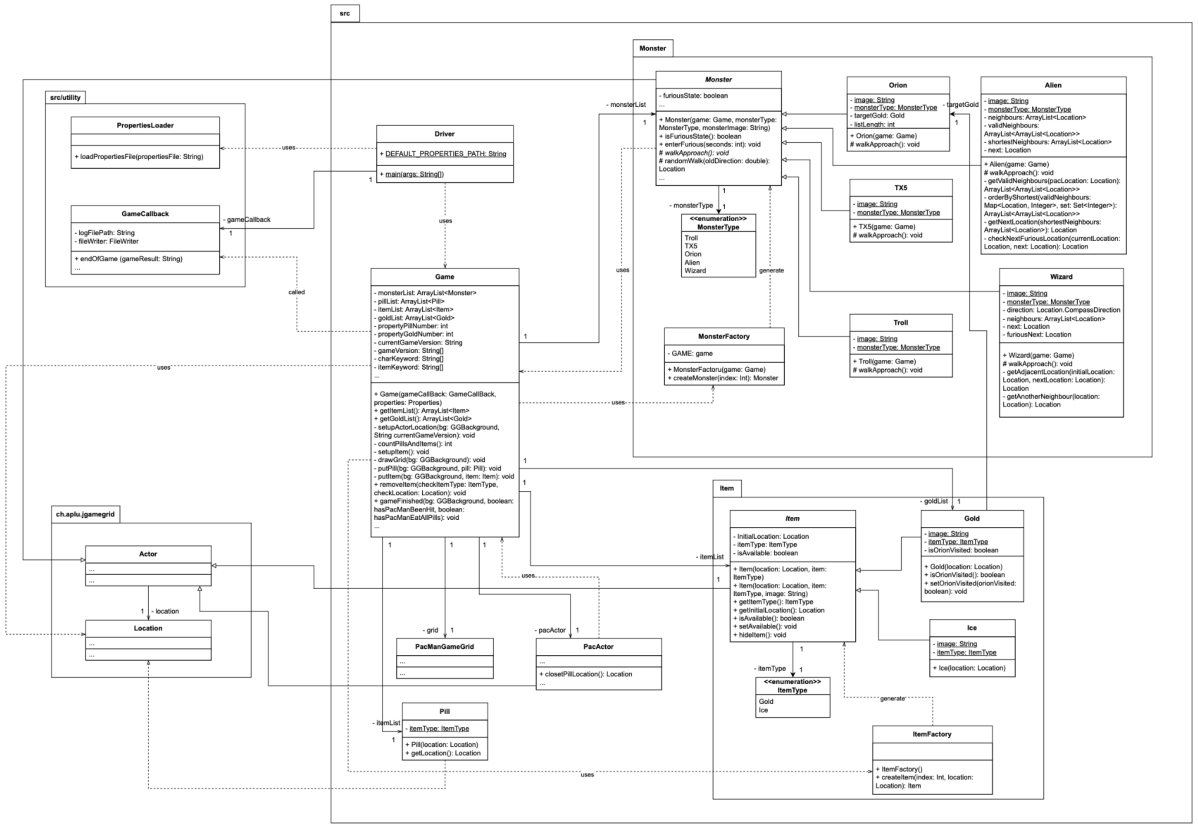
2. Based on the design class diagram, both PacActor and Monster can be grouped together under the Character Class since there are some duplicate attributes and functions between them. However, these duplicate functions in PacActor only existed for the testing purpose. Thus, we decided to keep PacActor as a dedicated Class for our design.

Diagrams:

Domain class diagram



Static design model



Dynamic design model