SWEN30006 Project 1: PacMan in the Multiverse

Workshop: Wednesday 4:15pm
By Sanskar Agarwal, Elise Marcun

Part 1: Analysis of Current Design

The current design, as provided, for the simple PacMan game had several areas of concern which would hinder the extension and development of new features when progressing from *"simple"* to *"multiverse"*.

a) Bloated Game Class
The Game class had become overly bloated, complex and difficult to read as a result of essentially being used as a controller for both the Monsters and the Items in the game. This led to high coupling and dependencies with other classes, especially the Monster class. Furthermore, most of the Item-related logic was being handled in the Game class - only adding to its bloated nature.

b) Monster Class and functionality with different types of Monsters
In the original design, the Monster class was being used as a generic object for all types of Monsters, resulting in low extendibility. In the original implementation of the "*simple*" PacMan game, this meant that the Monster class created both the Troll and TX5. These different types of monsters were only distinguished through the MonsterType attribute 'type'. The Troll and TX5's differing walk approaches and behavior was, hence, handled using if/else statements, creating low cohesion in the code. While this was manageable in the "*simple*" version of the PacMan game, the if/else logic would become increasingly difficult to maintain and extend with the additional functionality required in the "multiverse" version. That is, the three additional monster types (Alien, Wizard and Orion) that have more complex behavior. The Domain Class Diagram (Software Model 1)**,** which has 6 lines connecting Game and Actor, demonstrates high coupling between the Game and Actor classes. This high coupling between classes indicates that they are too interdependent which can be problematic as changes to one class can have significant impacts on the other.

Part 2: Proposed new design of the simple version

To address the key areas of concern outlined in Part 1, we proposed a new design of the simple PacMan game, focusing on enabling extendable Monster classes, improving the maintainability of the code, increasing the cohesion within classes, and decreasing coupling between classes. Importantly, the changes we made to refactor the "*simple*" version were designed to form a foundation of a design that would support further extension for the "*simple*" and "*multiverse"* game . This is captured in the Static Design Diagram (Software Model 2), noting that this diagram also includes the features required when extending for the "*multiverse*".

a) Creation of the MonsterHandler
The MonsterHandler class was created to help separate the Monster logic from the Game class, ultimately increasing cohesion. As per the Controller pattern principle, the MonsterHandler acts as an intermediary between the Game, Monster class and the Monster subclasses, reducing coupling between them. The MonsterHandler became responsible for handling the creation and controlling of the behavior of Monsters in the game, and allowing the Game class to delegate this responsibility to it. For example, in the "*simple*" version, the MonsterHandler is relied upon to set-up monster data and set common attributes, such as their seed and slow down factor. Note that the MonsterHandler

was more heavily relied upon in the extended version of the game when the monsters were required to freeze or become furious.

b) Implementation of Monster subclasses
As described in Part 1, the original design and code had low extendability when adding new monster types and did not handle the two “simple” monsters (Toll and TX5) consistently with best practice as it used numerous if/else statements.

To address these issues, we applied the concept polymorphism and introduced Monster subclasses (only Troll and TX5 in the “simple” version). Each Monster subclass could now implement its own specific behavior for the unique ‘monsterWalk’ method. In doing so, we separated the random walk functionality to the parent (Monster) class, as it was common between both monsters. Each monster could call on the same method, reducing code duplication and increasing maintainability. We also decided to make the Monster class abstract since every monster instantiated would represent one of the subclasses. This design also meant the ‘monsterWalk’ method could be abstract in the Monster class, and it would be automatically overridden by each specific monster subclass. This approach made the design and code more extendable, maintainable and readable when adding new monsters in future.

c) Creation of the ItemHandler Class
We used the principle of indirection to decouple responsibilities from the Game class by introducing an ItemHandler class, which is responsible for managing functionality related to the pills, gold and ice items. This includes functionality required for the game, such as counting pills and items, setting up the pill and item locations, and placing and removing items from the grid as required. There was a tradeoff between whether to have methods such as putPill(), putIce() and putGold() in the Game class or the ItemHandler

class, and ultimately decided to implement these methods in the ItemHandler class to increase cohesion, supported by the Creator pattern. Although all the Graphical User Interface (GUI) functionality was originally held in the Game class, it was more logical for the ItemHandler class to create and put these items on the grid. This also adheres to the high cohesion GRASP principle as having these methods in the ItemHandler class rather than Game class keeps the object more focused and understandable.

It is worth noting that this may contradict the Information Expert pattern which suggests that all GUI logic should be contained within the Game class. However, we decided to include these methods in the ItemHandler class to align with the Creator and High Cohesion patterns.

On the other hand, we decided to keep the logic of loading the pill and item locations (loadPillAndItemsLocations() method) in the Game class, as it required the reading of the ‘properties’ files. According to GRASP principles, as the Game class is an Information Expert of the properties data, it is more logical and understandable to keep this method in the Game class. Similar reasoning was also applied to the drawGrid() method in the Game class. Although this method references the itemHandler, which could be seen as increasing coupling, we concluded that, because the Game class is an Information Expert in regards to the GGBackground and data, that it made more sense for this method to remain inside the Game class.

Part 3: Proposed design of the extended version

All of the changes made in Part 2 had the benefit of improving the design in the extended "*multiverse*" version. The final proposed design which was implemented for the extended version with the ability to be configured as "*simple*" or "*multiverse*" is represented in both Software Model 2 (Static Design Diagram) and Software Model 3 (Dynamic Design Diagram). Below, we have detailed how the changes outlined in Part 2 have allowed the new features of the multiverse version to be implemented:

a) Creation of MonsterHandler
In the extended version of the game, monsters are required to freeze when the pacman eats an ice cube, and become furious (move at a faster pace) when pacman eats gold. This is exhibited in the Dynamic Design Diagram (Software Model 3), which also shows the corner cases where the Monster can go directly from the Furious state to the Frozen state when the pacman eats ice within 3 seconds of eating gold, but cannot go directly from the Frozen state to the Furious state (i.e. if a monster eats gold within 3 seconds of eating ice, the monsters will not go from Furious to Frozen).

These additional features would have been very difficult to implement without the MonsterHandler class, proposed in Part 2. The MonsterHandler class enables a central intermediary method that handles the behavior of all monsters when they enter the Furious state and the Frozen state, as depicted by the furiousMode() and freezeMonsters() methods respectively. If the MonsterHandler was not implemented, there would be significant code duplication as the stopMoving() and setSlowDown() methods would have to be called individually for each monster when the pacman eats ice and gold respectively.

b) Implementation of Monster subclasses
The implementation of the Monster subclasses allowed us to introduce the Alien, Wizard and Orion monsters more easily. Without the introduction of subclasses, for each of the new monster types, we would have had to add additional if/else statements to the already complicated code in order to implement their different walk approaches. This would have significantly worsened understandability and maintainability. However, the introduction of the subclasses meant that we could easily add these three new monsters to the game, implement their more complex monsterWalk() methods and add additional methods where required. For example, this design allowed the Alien and Orion to implement the isPossible() method to determine whether their next move was feasible as per their own individual requirements. Additionally, the Monster subclasses allow for greater flexibility in the game version, meaning the user can easily switch between the "*simple*" and "*multiverse*" versions of the game.

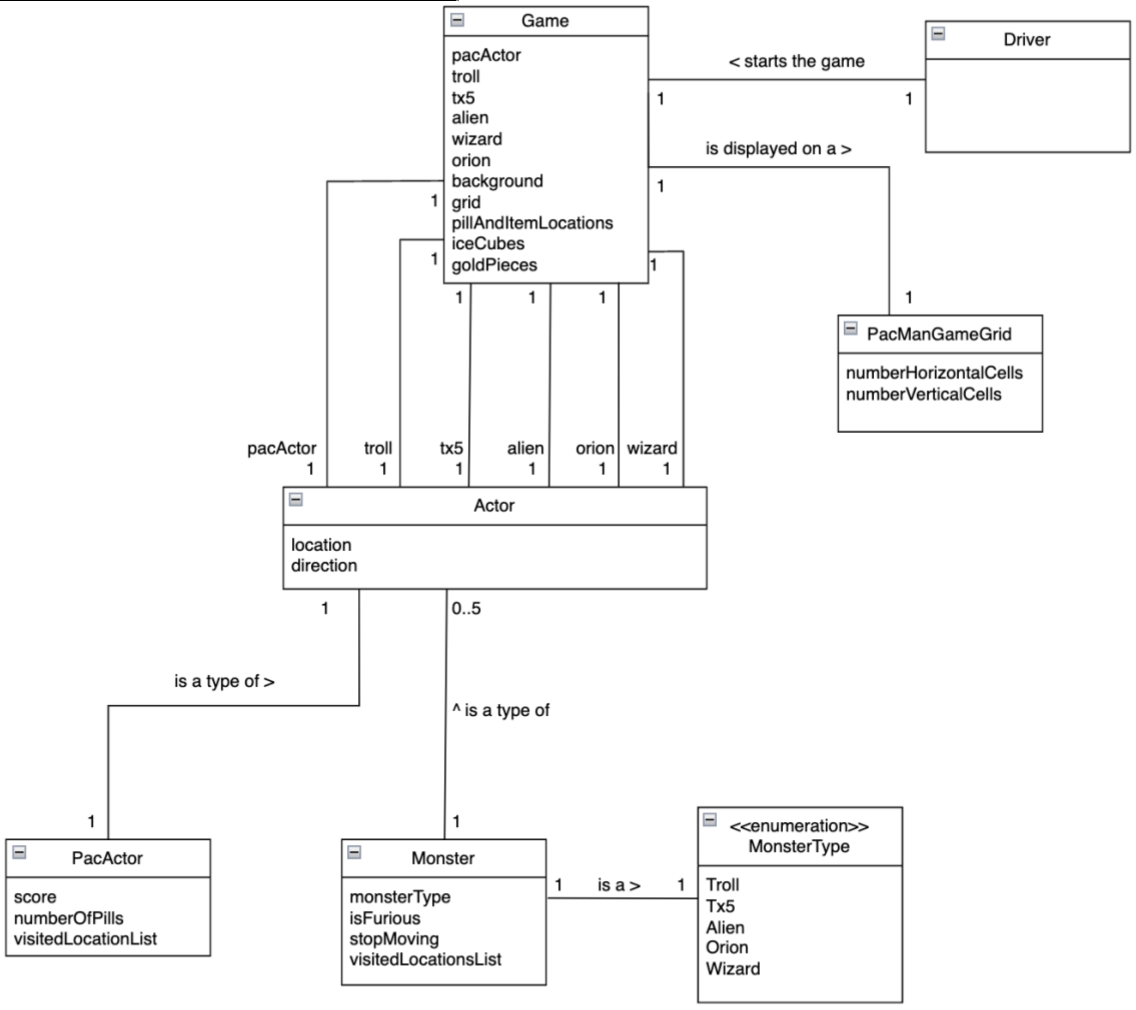
There were also other design choices that we contemplated when refactoring the code. For example, we discussed the benefits of using pure fabrication to create a separate 'Score' class, to keep track of the pacman's score, however, we determined that keeping 'score' as an attribute in the PacActor class would be better practice as implementing such pure fabrication would adversely affect the design by increasing coupling.

Appendix: Software Models

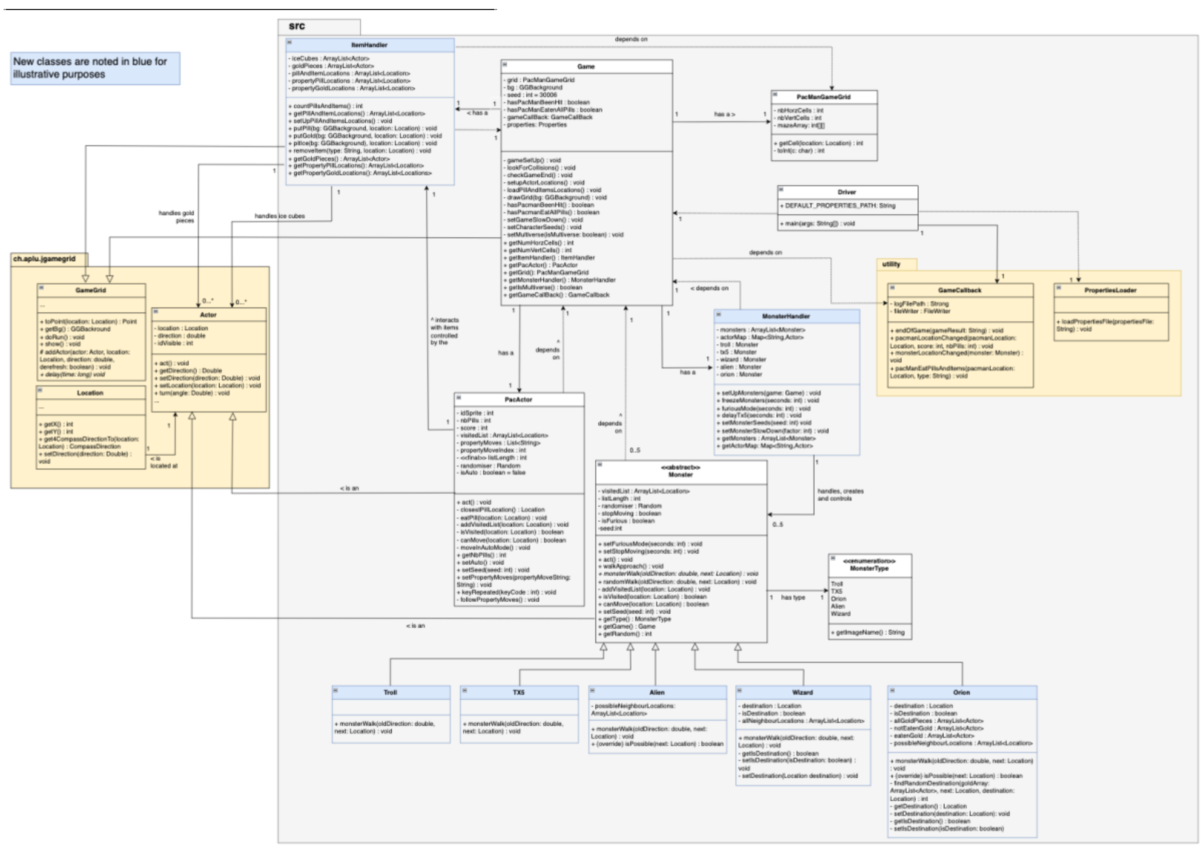
Please use this link to refer to the Software Models in drawio for improved readability:

https://drive.google.com/file/d/1w-JO36SjKu4MLnCQN4lva1PSmbY-TcDr/view?usp=sharing

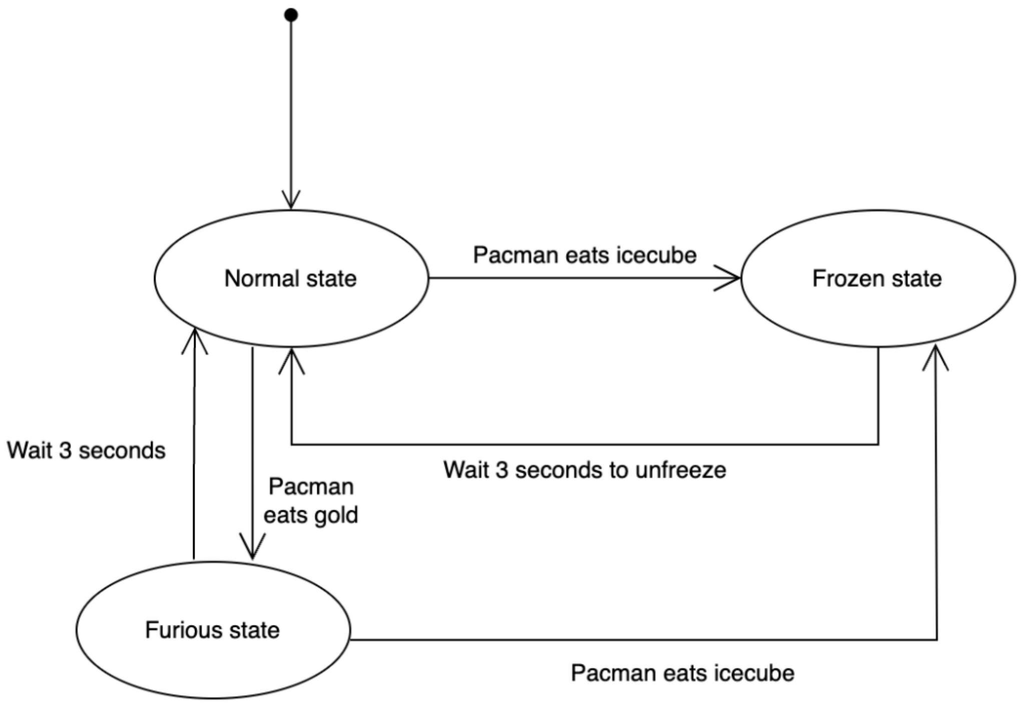
a) Software Model 1: Domain Class Diagram



b) Software Model 2: Static Design Model



c) Software Model 3: Dynamic Design Model


Normal state
Pacman eats icecube
Frozen state
Wait 3 seconds
Wait 3 seconds to unfreeze
Pacman eats gold
Furious state
Pacman eats icecube