Nicholas Woo 1313825
Min-hyong Lee 1270723
Suhail Cassim 933878

SWEN 30006 – SOFTWARE MODELLING & DESIGN

Project 1 – Pacman in the Multiverse

Design Analysis Report

Nicholas Woo 1313825
Min-hyong Lee 1270723
Suhail Cassim 933878

Analysis of Current Design

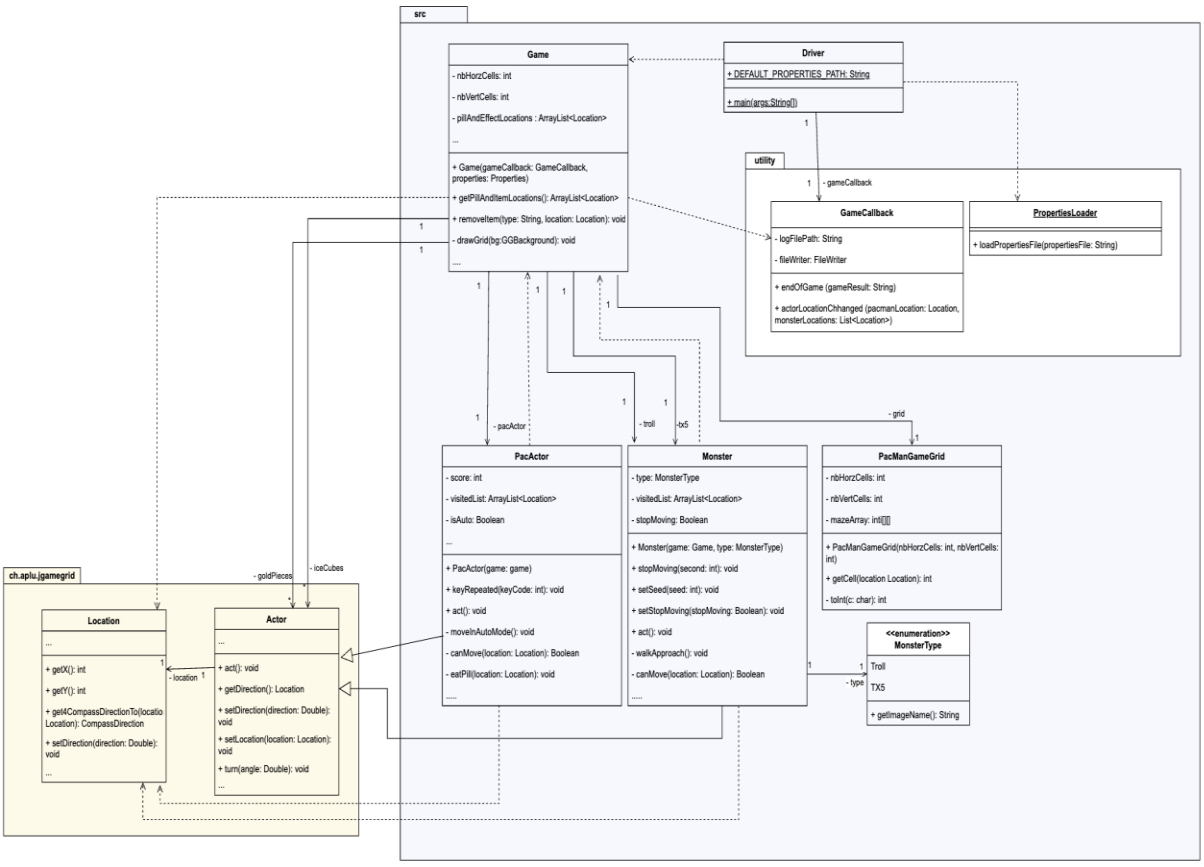

Figure 1. - Partial Class Diagram (Original)

Upon conducting an extensive review of the code and partial class diagram (i.e., Figure 1) in line with GRASP principles, it has become apparent that certain issues within the system require attention and resolution.

At a fundamental level, the original implementation suffers from ‘High Coupling’ and 'Low Cohesion’; due to the strong interdependencies and unrelated responsibilities between the different components of this software system. Therefore, we can argue that the underlying design is extremely fragile and not very extensible, to support current and future versions of the game ‘PacMan in the Multiverse’. To preserve the original functionality of the system, we refactored the existing code bases and implemented new features through modifications in the existing framework.

Nicholas Woo 1313825
Min-hyong Lee 1270723
Suhail Cassim 933878

Analysis of Proposed Design (Simple & Extended)

This section will seek to evaluate the proposed design for our simple and extended game with respect to three main concepts, namely, items, monsters, and game. These are the core changes that we have made to the original code with the aim of making it more extensible and configurable. Future additions to the types of monsters and items will be more convenient and efficient. Our changes have also improved the design of the code base by increasing cohesion and reducing coupling between the classes.

Items

Items within the game are unique objects in themselves, however they do share common functionality, excluding those that extend from the ‘Actor.java’ class in the ‘jgamegrid’ library. Thus, we can apply polymorphic techniques such as the implementation of the parent class ‘Item.java’ to abstract-out this commonality, as seen in Figure 2.

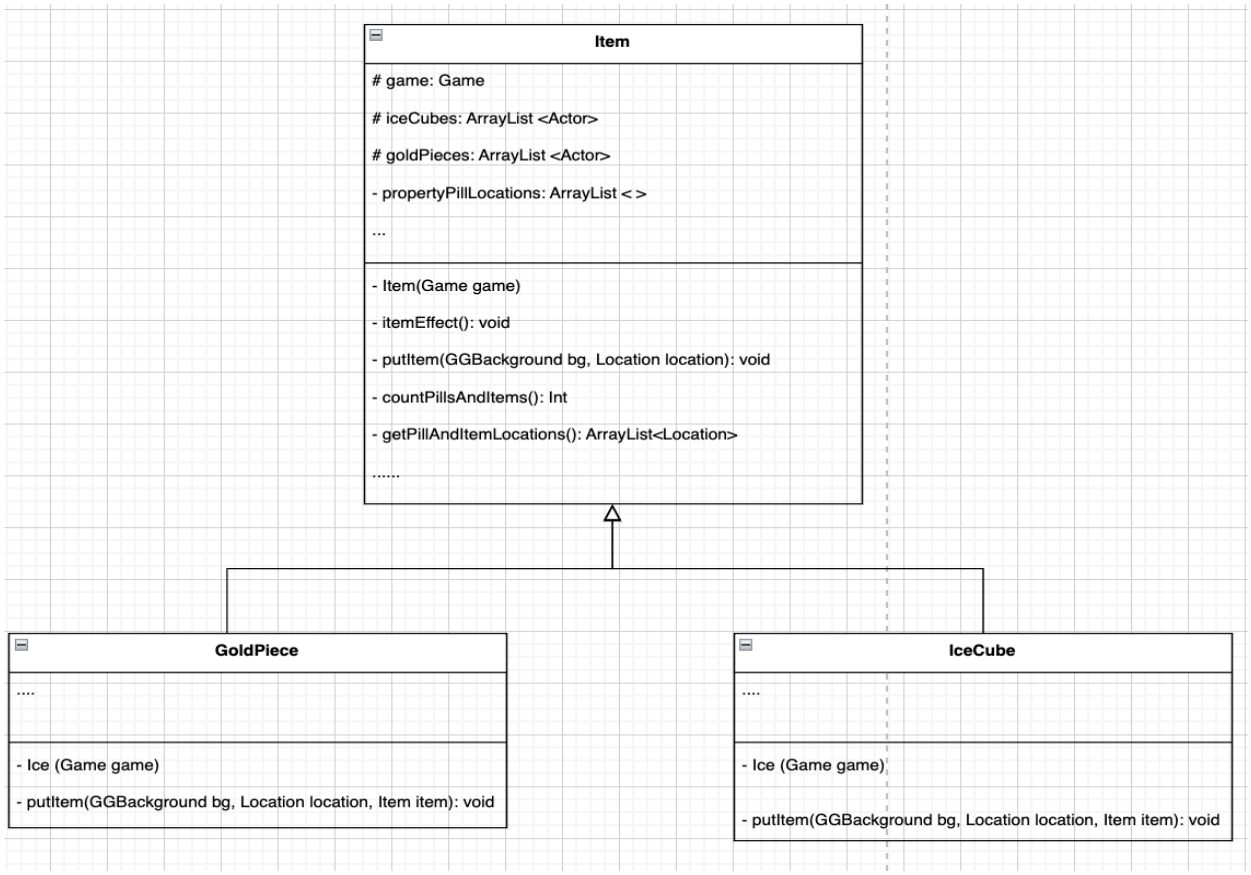

Figure 2. - Partial Static Design Model for Items

At face value, the new inheritance relationship will eliminate the use of any duplicate code across Items, in effect improving its readability and reusability. This is likely to resolve any logistical issues in the code, especially in relation to modifying existing code or facilitating future extensions. Furthermore, this design embeds Item specific functionality in a manner that aids modular and flexible interaction. Therefore, we can argue that this design offers a greater degree of cohesion for item specific components within the system; to produce a more stable design and facilitate extensions/improvements for items within the game. For example, this implementation allowed us to override methods such as ‘putItem’ (in ‘Item.java’) to successfully place the new items on the grid e.g., Gold & Ice. It also reduces the need for switch statements to prevent low cohesion and bloatedness.

Moreover, we apply abstraction techniques to ensure that only ‘Item.java’ and its relevant child classes consist of the responsibilities relating to Items in the game. For example, we can remove methods such as ‘countPillAndItems’ and ‘setUpPillAndItemLocation’ from the ‘Game.java’ class; in effect removing any unrelated Item specific responsibilities. Therefore, we can argue that this hierarchical design induces ‘loose coupling’ capabilities for Item components within the system i.e., this design reduces dependencies between individual Item components and the rest of the system. In effect, this improves the maintainability of Item components, because any logical changes within its domain will affect fewer components in the overall system.

The Item class also stores gold, ice and pill locations and their corresponding array lists which keeps track of their numbers. Aligning with the Indirection GRASP concept, the Game class only has to access the Item class to obtain data on the locations and number of gold pieces and ice cubes instead of the Ice and Gold class separately which reduces coupling in our implementation.

Nicholas Woo 1313825
Min-hyong Lee 1270723
Suhail Cassim 933878

Monsters

Similar to Items, Monster objects are unique in themselves, but do share common functionality. Therefore, we can apply polymorphic techniques such as the implementation of a parent class ‘Monster.java’, as indicated in Figure 3.

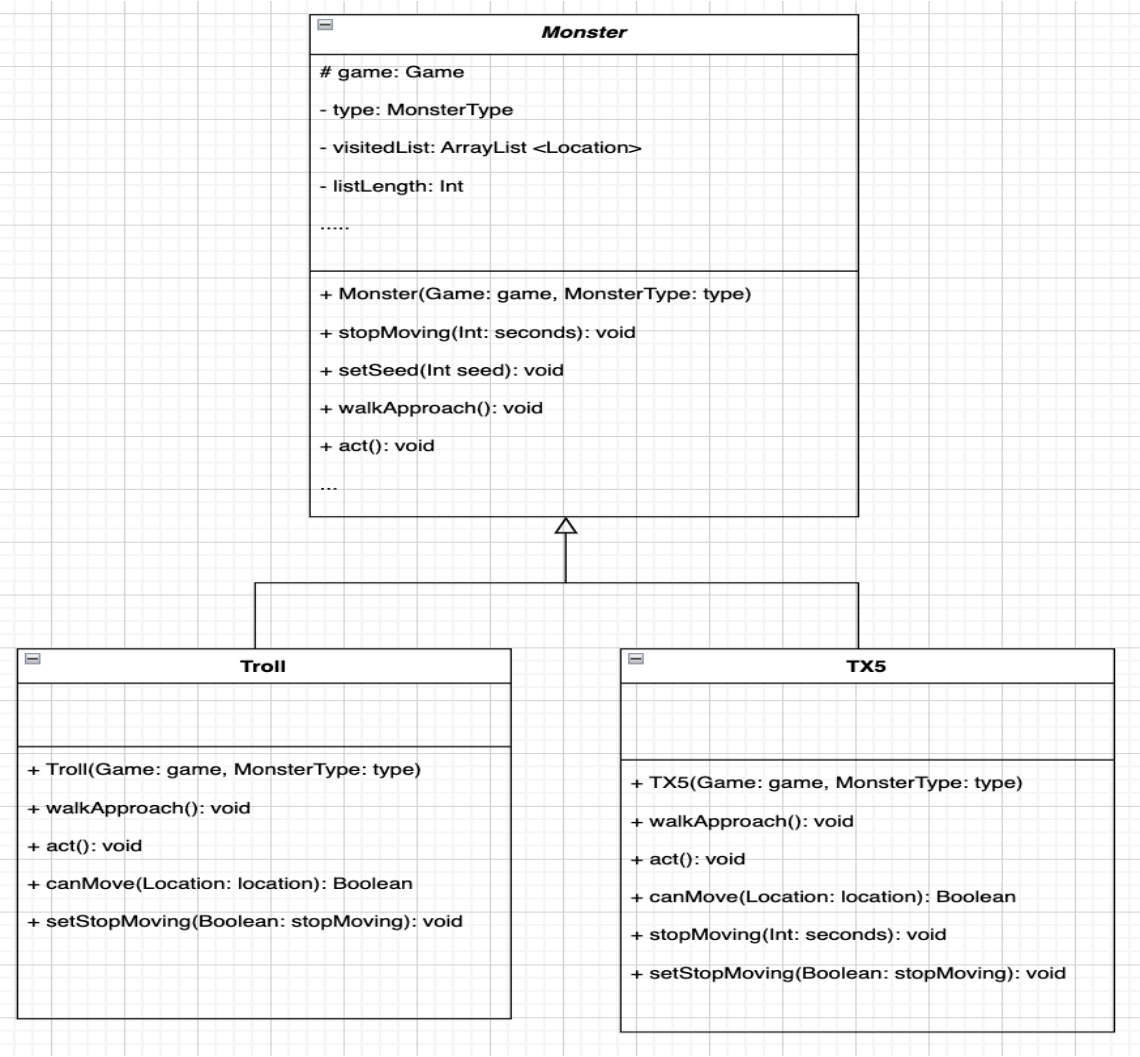

Figure 3. - Partial of Static Design Model for Monsters (Simple Version)

This inheritance relationship removes any duplicate code across monster components. For example, methods such as ‘stopMoving’ or ‘setSeed’ are common across different Monster Types. Therefore, we can embed this common functionality into the parent class ‘Monster.java.’ to better encapsulate the monster components.

This technique prevents any logistical issues, especially in relation to modifying existing code or facilitating future extensions for relevant Monster components. For example, this implementation allowed us to extend to the simple version of the game, to include more monsters, with enhanced capabilities, as seen in Figure 4. Specifically, this principle of polymorphism enabled us to override methods such as ‘canMove’ and ‘walkApproach’ to account for the unique interactions across the different Monster types.

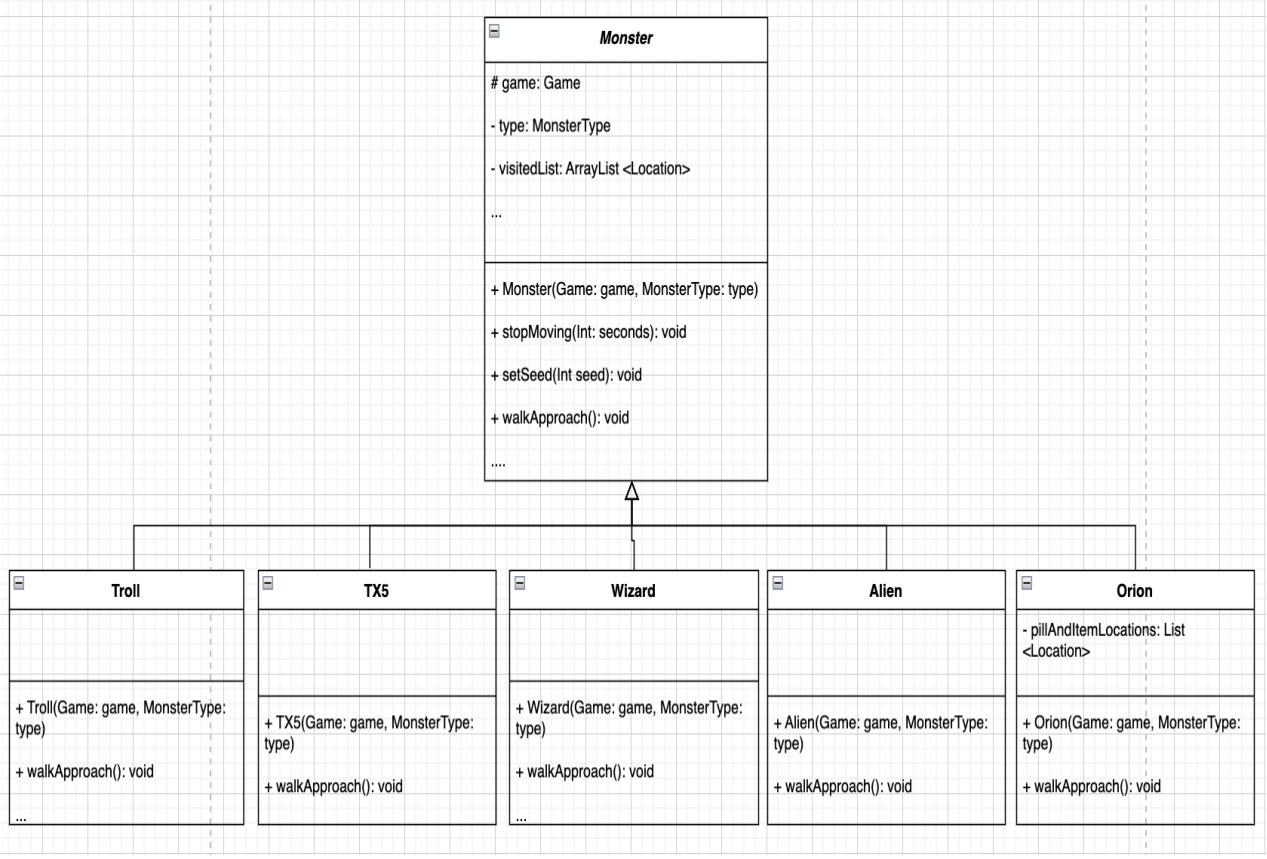

Figure 4. - Partial of Static Design Model for Monsters (Extended Version)

Therefore, we can argue that this hierarchical design facilitates higher cohesion between the system and corresponding Monster components, due to tighter coupling across the individual components within the Monster Domain.

Moreover, this level of encapsulation and abstraction achieves looser coupling between the overall Monster component and the rest of the system i.e., this technique eliminates any irrelevant dependencies between the Monsters and the rest of the system, by restricting its responsibilities only within the Monster Domain. In effect, this design offers greater stability to the underlying system and further facilitates future extensions/improvements.

Regarding the implementations of new features in the Multiverse version, a major change was to take the advantages of polymorphism and method overriding to implement each monsters' new walk approaches. For example, Orion's walk approach locates a random gold piece on the map (whether or not it has been picked up) and uses the same algorithm as the Tx5 as a function in Monsters (walkTowards()) to proceed to its location. Employing code reuse has helped reduce code bloat, saved time, and minimized risk of code development.

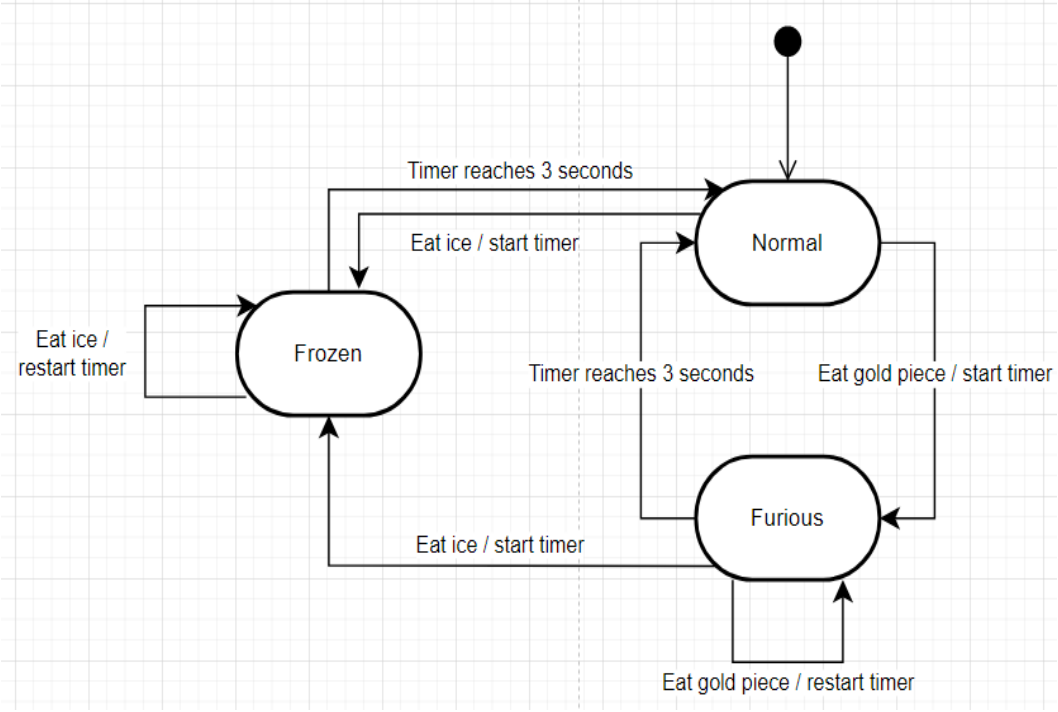

Figure 5. - State machine diagram describing items' effect on monster states

Illustrating the additional capability of items and monster states, PacActor who interacts with the items, will call new methods in Game to implement new mechanics of freezing and angering monsters (freezeMonsters() and angerMonsters()) as it acts as the information expert with respect to the game version and records instances of monsters. The specification describes monsters as able to be angered but not while already frozen; this behaviour suggests that monsters is a state-dependent object. Figure 5 below details the logic and conditions in which the new items have effect on the state of Monsters.

Game

It is evident that the original implementation of the class ‘Game.java’ lacked a clear set of responsibilities governing the underlying goal it was trying to achieve. To support the new implementations of Items, Monsters and the PacActor, we apply the creator pattern across the ‘Game.java’ class. This pattern ensures that only one component in the software system will be responsible for initializing the relevant objects i.e., only ‘Game.java’ is responsible for creating the corresponding monster, item and PacActor object(s). We can justify the use of this pattern by the multiple references to, and records of, instances (Monsters, Items & PacActor) within the Game component. Therefore, we can argue that this principle helps improve cohesion and reduce coupling within the system; because it defines a clear set of responsibilities for the ‘Game.java.’ class, in effect removing any unrelated dependencies originating from it.

Nicholas Woo 1313825
Min-hyong Lee 1270723
Suhail Cassim 933878

Complete Static Design Model