SWEN30006
Project 1 Report
21/04/2023

By Matthew Pham, Jason Hoang, Jason Yang.

Part 1: Concerns With Original Design.

Several aspects of the current design and implantation break many GRASP principles and hinder the extension of new features. However the following are the most concerning:

1.1 Items are not well defined:

At the moment, no item class implementation exists, and are defined by various unrelated variables (e.g. background colour, random sprites, characters in maps, integers). This by itself breaks many GRASP principles, such as lack of polymorphism, lack of indirection, lack of protected variation and the lack of a controller

- Lack of an Information Expert and Protected Variation:
  - Currently, in order for PacActor to consume a pill, it must understand the exact definition of a pill in terms of which integer corresponds to each item, which background colour the item uses, as well as if the item has a sprite. Additionally, Game also must know similar information about how items are defined as integers. This means that if the definition of an item changes, the programmer must manually adjust PacManGameGrid, PacActor and Game individually.
  - Additionally, because items are scarcely defined, it is practically impossible to gain knowledge of when an Item is consumed on the side of Monsters, without manually detecting change in the entire game grid, meaning that the extensions relating to items are currently impractical to implement.

- Lack of Polymorphism and High Coupling:
  - There is a lack of an abstraction between an item's consumption and the effect's it has on the state of the game. Once again, this means that PacActor and Game are dependent on the implementation of an item as described in PacManGameGrid, meaning that changes to either class cannot be viewed in isolation. This results in high coupling.

- Lack of a Controller and Indirection:
  - This is a more minor issue, however currently items can be created in any point in the current design and affect PacManGameGrid without the class's knowledge, even though PacManGameGrid currently defines many of the implementation details about items. It would be preferable to have a controller instead for the instantiation and creation of these items on the game grid.

1.2 PacActor and Monster Implementation is Inextensible:

Despite PacActor and Monster following similar execution loops, their functionality is strangely separated. Additionally, Monster's implementation lacks extensibility, both with their response to Items being consumed and individual monster type behaviour.

- Low Cohesion:
  - PacActor shares a lot of functionality with Monsters like the Troll, sharing patterns in general movement (use of randomiser, visitedList, canMove()). Even despite the fact that some of these functions should outright be moved to other classes, PacActor and Monsters share enough functionality to warrant at minimum, some form of close-knit relationship between the two.
  - Monster itself is also managing the functionality of two different types of monsters (Troll and TX5), resulting in difficulty extending/adding the behaviour of both old and new monster types.

- Lack of Polymorphism:
  - Monster has to accommodate for the needs of all Monster types individually, even though all aliens share the same basic functionality and behaviour. Ideally, each monster type should be unaware of the other monster types' implementations.

1.3 Confused Responsibilities of PacManGameGrid and Game:

At the moment, Game is performing too many different functions that should be delegated to PacManGameGrid.

- Lack of Information Expert and Protected Variation:
  - Despite the fact that Game adds items on its GameGrid, PacManGameGrid defines the structure of the GameGrid as well as the definition of items, resulting in information regarding items somehow being in both PacManGameGrid and Game. If PacManGameGrid changed how it defined the GameGrid, much of the Game code would have to change.

- Low Cohesion:
  - Game currently knows about every other class in the current design. Ideally, we want to isolate Game's knowledge to knowing how to manage Actors and the GameGrid, not knowing how they actually function.

Part 2: Proposed Fixes to Simple Game.

Each of these changes correspond to the above mentioned issues to the old game design and provide a foundation to facilitate future extensions.

2.1 ItemHandlers, Items, ItemEventListeners and ItemEventCodes:

Items are now defined classes that rely on an implementation of an ItemHandler. Instead of the creation of items being dependent on PacManGameGrid's definition, the ItemHandler is the creator of Items, and Items are defined by the both private attributes (Color, Radius, Sprite) as well as the ItemEventCodes they emit to their ItemHandlers. ItemHandlers distribute these emitted ItemEventCodes to their subscribers, who must implement the ItemEventListener interface

- Information Expert, Creator (Implements Indirection) and Polymorphism:
    - Now, ItemHandlers are the only objects that know how to create an item, so they are Creators of items. Any object that wants to create an item must query an ItemHandler to do so, meaning ItemHandlers will track the creation of all Items throughout the game's runtime.
    - Items are now responsible for their appearance and provide information on how to render themselves, becoming Information Experts of their implementation.
    - Any object that interacts with an Item does not need to understand the specific implementation of any specific Item object, implementing Polymorphism.

- Protected Variation and Low Coupling:
    - Now, any changes to the definition of an Item no longer results in the rest of the design needing to adjust to it. This applies to both how to render an item as well as how to respond to an item. All an object must do to respond to an Item is become an ItemEventListener, subscribe to the relevant ItemHandlers, and respond to their respective ItemEventCodes.

Extending Functionality:

If a programmer wants to implement a new item, they simply need to create a new Item along with its respective ItemHandler, which has been done for Pills, Ice and Gold (see Diagram 1 below). Each of these items should emit a specific ItemEventCode on consumption, which other ItemEventListeners can respond to without needing to know about the items themselves. Creating an Item is as simple as calling itemHandler.createItem().

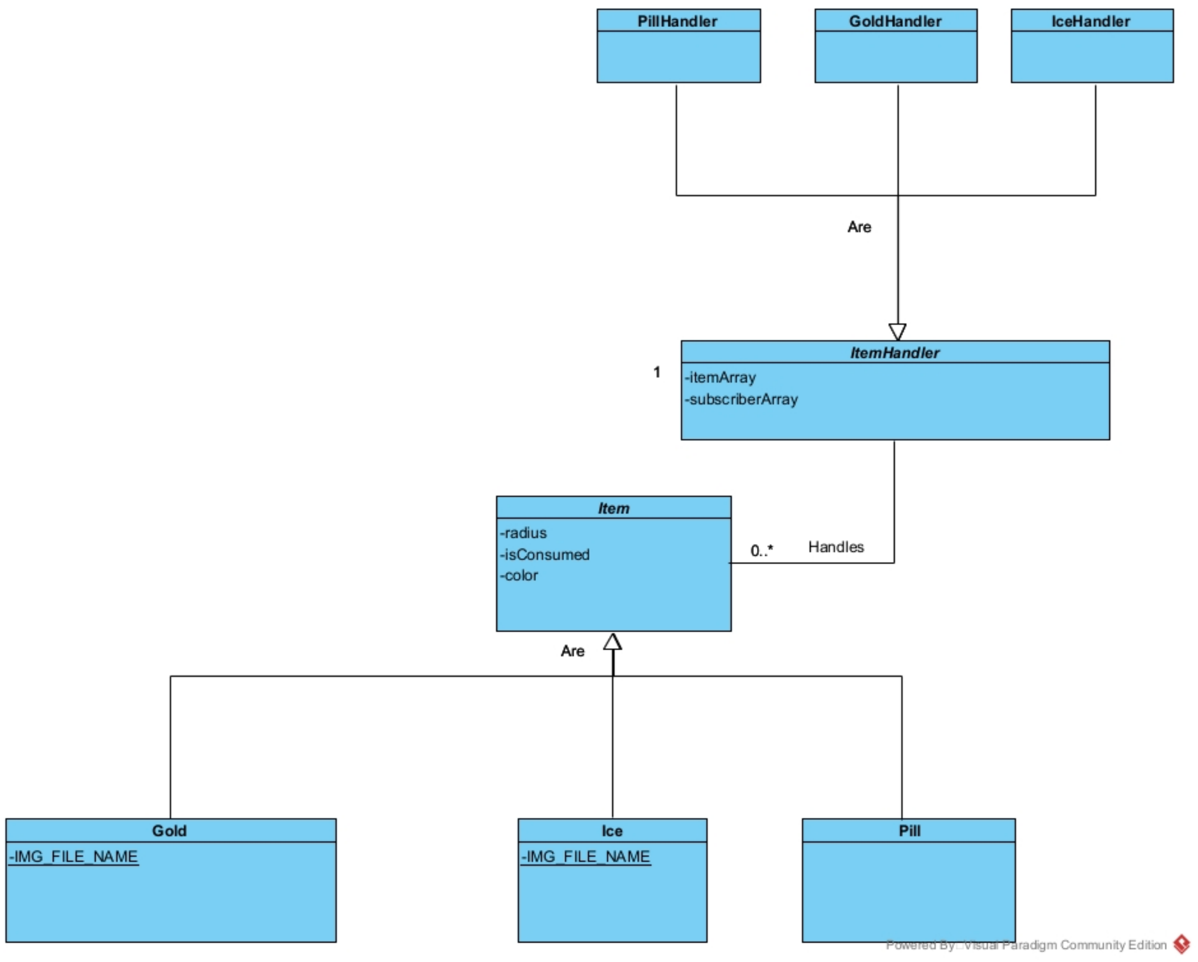

Diagram 1: Domain class diagram for simple refactor of Item and associated classes.

2.2 Abstract Entity and Abstract Monster:

The PacActor and Monsters now both inherit from Entity (extends Actor), which provides useful utility that is now shared by both respective classes, instead of being separated. Entity requires a Game object to be instantiated. On top of this, the Monster class itself must be extended to accommodate other Monster types.

- High Cohesion and Polymorphism:
    - Monster is now only responsible for the general behaviour of its subclasses, and subclasses are now only aware of their own internal implementations of walkApproach() and the state necessary to perform this function.
    - Monster no longer needs to know about how its subclasses have implemented its abstract functions.
    - PacActor can now use the same utilities that Troll and TX5 used, which before were implemented individually.

- Creator:
    - Game now manages entities on a closer level, meaning that all Entities created can be tracked throughout the game's run-time.

Extending Functionality:

Creating new types of Monster simply requires creating a subclass of Monster (see Diagram 2 below) that overrides Monster's walkApproach() method . For Monsters to respond to Items, Monster class only needs to become an ItemEventListener and subscribe to the relevant ItemHandlers. Monster can then deal with the necessary ItemEventCodes, irrespective of the implementations of its subclasses.

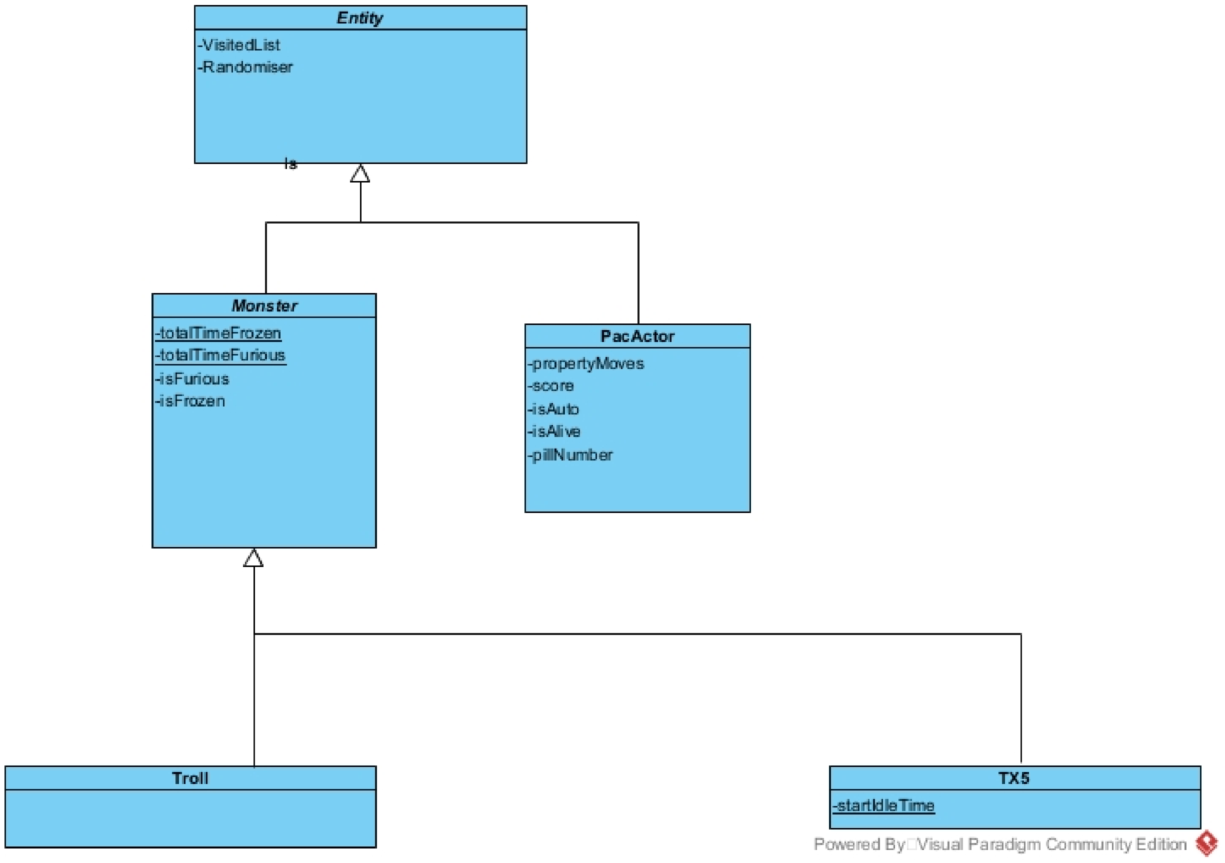

Diagram 2: Domain class diagram for simple refactor of Entity, Monster and PacActor classes.

2.3 Game and PacManGameGrid Refactor:

The responsibilities of Game and PacManGameGrid are no longer confused. Now, Game is responsible for providing Entities and the PacManGameGrid with the necessary information to function, and PacManGameGrid deals with how the GameGrid should be structured and how Items should be placed on the GameGrid (The Game is still responsible for adding Entities to the PacManGameGrid because it manages the two). PacManGameGrid now uses ItemHandlers to create Items, without the need to define the Items themselves.

- High Cohesion and Protected Variation:
    - Game no longer requires knowledge of PacManGameGrid to function, and now only needs to understand what information to parse from Properties into each entity and PacManGameGrid. Additionally, PacManGameGrid now only relies on ItemHandlers to define the implementation of Item and can now focus on placing and rendering those items and defining the GameGrid appearance and structure.
    - Changes to any class are now unlikely to majorly affect other classes.

- Low Coupling:
    - PacManGameGrid and Game require little to no information from each other to function properly. Entities now only need a reference to Game to gain information about the PacManGameGrid.

Extending Functionality:

Any change in how Game operates is no longer likely to impact the functionality of PacManGameGrid or Entity. Since the new design is more tree-like (see Diagram 3 below), this should generally make adding functionality to any class easier and less impactful to the rest of the design.

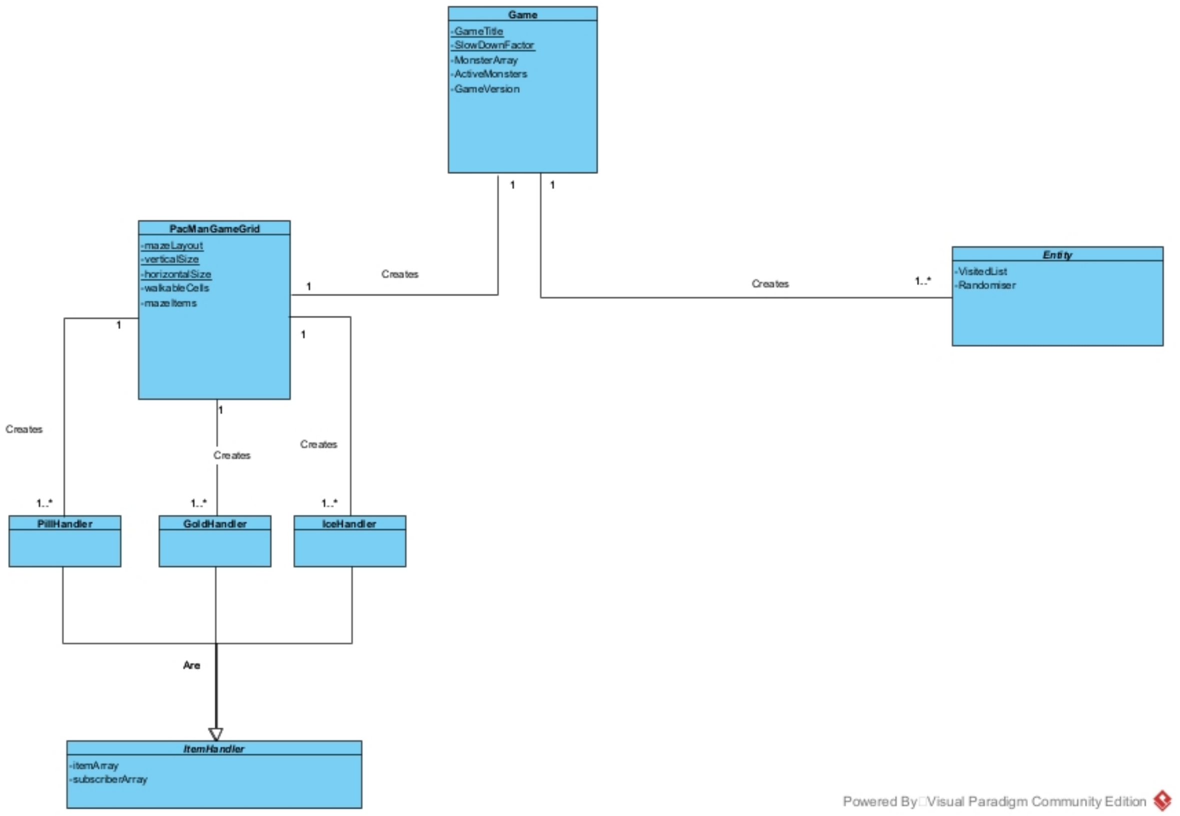

Diagram 3: Domain class diagram for simple refactor of Game and PacManGameGrid classes.

Part 3: Proposed refactor for extended version.

The changes made to accommodate the simple version of the game already accounts for the extensions to be applied to the extended version of the game, so little change is needed to extend functionality of the new design. For the Monster behaviour, Monster simply needs to implement ItemEventListener and Game can subscribe each Monster to the IceHandler and GoldHander through the PacManGameGrid. In the Monster class, Monster can then respond to each ItemEventCode in a non-subclass-specific manner, and the different monster type subclasses are implemented to override the abstract walkApproach() method used by Monster.