GROUP 2
The Duy Nguyen – 1100548
Ramon Javier L. Felipe VI – 1233281
Jonathan Chen Jie Kong – 1263651

SWEN30006 – Project 1 Report

Analysis of Current Design

First and foremost, the `Game` class lacks cohesion as it performs too many operations, even if not cohesive to its set of responsibilities. It also demonstrates high coupling due to the dependency stemming from the fact that `Game` and other classes are inseparable and closely utilising each other's methods. As such, independent implementation of new features could be difficult to achieve, and would only further increase coupling to the bloated class, and reduce cohesion as more behaviours of different objects need to be handled. `Game` class also has the issue of running the game within its constructor, as the primary responsibility of a constructor is to create an instance of a class.

Secondly, the current design of `Game` holds each monster as individual instances, which exemplifies high coupling. It does not have a logical encapsulation for monsters, posing issues when the game is extended to include different variations of monsters since it could lead to a bloated `Game` class. Furthermore, the `Monster` class currently performs operations for both `Troll` and `TX5` which impedes its extensibility and demonstrates low cohesion. With the current state of `Monster` class, methods that differ for each specific monster type will also become bloated, and must handle all differences of each monster (most evidently the `moveApproach` method).

More broadly, `PacActor` and `Monster`, which we will now refer to as live actors, also share considerable similarities. The current design, however, does not take this into consideration. If additional similar behaviors are to be implemented for live actors, the current code would have to add such behaviours to both classes that could have otherwise been generalised, or shared. This denotes low code reusability, generalisation, and the lack of ability to handle alternative subtypes. In other words, it is not sufficiently polymorphic and lacks protected variations.

Finally, the current design does not properly differentiate the various items, as it is using the background colour to determine which item is at a specific location. This extends to obstacles as well, being detected only via the colours. For this reason, the extensibility and flexibility of the logic are hampered, and that if there are ever items with the same background colour, additional logic must be implemented and cannot be refactored. The diagram also indicates that items are stored as integers in their Location list, making the list less useful to determine an item's distinctive behaviours. As a result, the game lacks a conceptual encapsulation for items, making the logic less cohesive, an issue shared with `Monster`.

All in all, the design lacks protected variation since alteration to a single logical unit will affect other logic quite profoundly, precisely because there is an apparent absence of well-separated units of logic. It is undesirable for maintainability, extensibility, and, thus, scalability.

Proposed New Design of the Simple Version

We improved the cohesion and coupling of Game by creating a pure fabrication called ObjectManager. It should be noted that ObjectManager will not be handling anything that is specific to behaviours of a game object, but rather only acts upon the Creator principle to instantiate the objects, as well as holding them and, most importantly, keeping track of their locations on the grid. Object-specific behaviours will be delegated to the objects accordingly. The Game class also follows the Creator pattern, because it not only closely uses ObjectManager but is also responsible for its sole initialisation. Moreover, we extracted the logic of running the game out of the constructor and converted it to a method run to singularise its role of object initialisation. This reduces the overall complexity of Game to just calling ObjectManager for state updates and, initially when running the game, adding actors onto its grid.

Therefore, ObjectManager is created to bridge the interactions between different game objects while still allowing them to have focused responsibilities and also reducing the responsibilities of Game. This ensures that ObjectManager only takes on a highly cohesive set of responsibilities and not bloating itself, but still having the required knowledge to carry out its tasks. As such, it is in-line with Pure Fabrication and Information Expert principle. For these reasons, ObjectManager is able to improve upon cohesion. More specifically, despite acting as a single point of contact for most classes and redirecting information as requested, it remains fairly independent from the logic of any other classes, and that responsibilities of Game class are more emphasised as a result. To ensure low coupling, various classes in the original design are encapsulated into superclasses to avoid dense relationships from ObjectManager.

In our design, we have modified Monster into an abstract superclass and created several other important superclasses, including Item and LiveActor (for Monster and PacActor). These superclasses utilise Polymorphism to facilitate encapsulating their subclasses' behaviours, as well as support adding new subclasses with relative ease. Encapsulation of these behaviours is rather significant since it increases code reusability as a result of shared functionalities among its subclasses, and decreases coupling since it can represent all of its subclasses in a single relationship. Importantly, a lot of tasks are delegated to these classes as much as possible to make the responsibilities more focused, hence high in cohesion. We also make use of template method (notably with moveApproach called in act method of LiveActor) so that a single unit of logic is all that needs to be implemented by any moveable actors in the game. Once again, this significantly reduces repeated codes. Though perhaps most importantly, our design has accordingly achieved greater protected variation.

Proposed Design of Extended Version

Evidently, items share a number of similarities within the game, though with important distinctions. We have therefore made a logical encapsulation of items by converting Item class to an abstract superclass with Pill, Gold and Ice being subclasses. This follows the principle of polymorphism, as the superclass aims to capture the items' similarities while still enabling each item to differ in their specific effects, if any, on the game. Furthermore, items are managed by the ObjectManager by being stored within the same hashmap, where the key is its location and the value is the item itself. This has a number of benefits:

1. Using location as key allows fast access to check if a location has a specific item or not.
2. Knowing what the item associated with a location is makes the map inherently more meaningful than the list in the original design, since each entry of the map actually contains the behaviours of the item.
3. Even as items increase in varieties, the hashmap can still remain as it is since it uses encapsulated Item superclass. As a result, the coupling remains consistent and high protected variation is also achieved.

To elaborate further on the choice of data structure, the hashmap makes use of items' inability to move, hence the use of location of items as keys. To extend further with this, an abstract superclass called `InanimateActor` is created to denote restrictions of `Item`'s behaviours, and increase the extensibility with the use of higher polymorphic abstraction. Currently, our design has yet to implement a concrete class for walls (the obstacles in which live actors cannot bypass) because walls are, so far, uniform and without any behaviour of its own other than its static locational importance to prevent actors from passing through it. This is why we only have a hashmap where key is the walls' locations, and values are arbitrary in `ObjectManager`. Nevertheless, we explicitly make `InanimateActor` extensible to walls, and if ever needed, it should not be a difficult task for A24 to create such a class.

For the same reason, `Monster` class was also converted into an abstract superclass. And similar to `Item`, it also effectively reduces coupling as `ObjectManager` class can make use of a list of monsters rather than individual variables of them. The decision of making `Monster` a superclass also makes it easier for other methods and classes to interact with them, namely the `Item` class, which, for the same reasons above, reduces coupling. This is because it provides a common interface for all items to interact with the monster, enabling methods to be applied uniformly. The `LiveActor` superclass presented in the simple design follows the same philosophy. And for any future extensions, a number of abstractions were also created with the same intention in mind. This includes:

- `Movable` interface which specifies the requisite behaviours for a game's actor to be considered a movable object.
- `GameActor` which entails any actors within the game grid, animate or not, as it is important to understand that, ultimately, each object within the game can be extended to become an actor itself. The reason for this is that, most notably, they all have a specific location they are currently in within the grid. Especially if dynamic, it is perhaps of good design to let the objects be an actor.

Overall, the polymorphic superclasses and interfaces are designed with adherence to several GRASP concepts, as mentioned above, to significantly improve maintainability, extensibility, and scalability. For instance, if new items or monsters were introduced, they could simply inherit their respective superclass and define their own implementations, while other shared logic will be kept unchanged. Ultimately, this has achieved significantly lower coupling, higher cohesion and protected variation.

Software Models

Software models are submitted as a separate document in the following:
DomainModel.pdf, ***StaticDesignModel.pdf***, and ***DynamicDesignModel.pdf***.