SWEN30006 Project 1
Rohit Sandeep Ambakkat 1200197
Thaya Chevaphatrakul 1167144
Tran Thanh Han Ha 1472202

Analysis of Current Design

Problems have been identified for the current design of the simple version of PacMan in the Multiverse, in which suggestions are made based on the GRASP principles. A code refactoring process is required to ensure the application's extensibility for new features. Hence, the analysis will start from the given design partial class diagram to determine all current issues in order to improve internal code more efficiently. Notably, the extended features are kept separately using a configurable version later in the code.

First, there are certain risks from adding new elements to the application that might lead to an overwhelming number of methods and functionality in the game class in the codebase. As such, it is advisable to reduce the coupling slightly but, more importantly, increase the overall cohesion of the design. However, there is a tradeoff between coupling and cohesion due to high dependencies between the classes in a game's design. Therefore the implementation needs to be adjusted moderately. The issues identified are as follows:

1. Low cohesion
   <<Game>> class is hugely bloated and performs many responsibilities, which may make it unfocused. In addition, the current design is not applying the information expert pattern, which may help it to achieve better cohesion. Methods such as putGold(), putPill(), and drawGrid() should ideally be placed in the **<<GameGrid>>** class or another specialised class instead to reduce the bloating of the game grid as the <<**PacManGameGrid>>** also contains the necessary information to perform these responsibilities.
2. High coupling
   Much of the current design revolves around **<<Game>>** class. This is not ideal for the extension of new features, as the majority of the classes are not independent of the game. Additionally, changes to a new class may involve changing the game and vice versa, requiring much maintenance.
3. No polymorphism
   Lack of appropriate polymorphism leads to poorly protected variation in **<<Monster>>** class. Using only an enum to record various types of monsters as well as multiple if statements in the walking approach could be a better design in which we would need additional switch conditions for them. Lack of polymorphism, such as inheritance or overridden methods, is ineffective. This is because, when we want to develop new monsters rather than simply extending our code, we would have to make significant changes to different methods, such as the walkApproach(), which is not ideal as we would like modules to be open for extension and closed to change.
4. Creator pattern
   Currently, the class **<<Game>>** is adhering to the creator pattern as it has the initial information regarding the actors and the game grid to initialise them. As both elements

are contained within the game, it is appropriate that their creation falls upon the game class. However, as discussed, the game class could be more balanced and cohesive. To improve this, we can delegate the creation of the grid elements/items to the **<<PacManGameGrid>>** class, as it also has the information required to create these elements, which is done by passing from **<<Game>>**. This process allows us to adhere to the creator pattern and even use the information expert pattern, as the Game grid class is responsible for knowing about the location of elements within it.

5. Poor design regarding the pills and items
   The current design does not employ any GRASP patterns regarding the pills and items. In the design, no class or method contains information regarding their properties and functionality; instead, the data is scattered. An example of the high level of maintenance required in the current design can be seen if we wanted to update the colour of an item such as gold; it would need us to change it in multiple locations, such as in drawGrid() and eatPills(). If it were to employ the pure fabrication pattern and the information expert pattern, the design for the pills and items would be much more easily extensible, cohesive, and reusable as we would be able to assign all the responsibilities to a singular class. Additionally, if we were to add a new item, it would require minimal maintenance.

Proposed Design of Simple Version

Our proposed design aims for higher cohesion and strives to keep classes focused, giving a side effect of low coupling. According to the problems determined above, some suggestions to improve the quality of the simple version include:

1. **<<Monster>>** class becomes an abstract class allowing us to use polymorphism to ensure protected variation by creating multiple specific subclasses. With the creation of a monster superclass, we can comfortably make changes or extend the behaviour of walkApproach() of existing monster types without causing any possible impacts to another class and store all the information regarding an individual monster type in its own unique class. It is also ideal as we can still contain the similar behaviour of all the monsters and, instead, focus on the behaviour that differs between them within their specific subclass, providing us with higher cohesion. The use of polymorphism in this manner is also preferable to handle new variations of the monster.
2. Creating a **<<Sprite>>** superclass helps contain all the typical behaviour between the game's sprites and reduces the coupling in the original design. Rather than having multiple classes with similar behaviour extending both Actor and linked to Game, we now have one primary class.
3. In order to promote higher cohesion and to ensure that the game class does not become heavily bloated, especially when considering possible extensions, specific responsibilities are moved to **<<PacManGameGrid>>. <<Game>>** responsibilities are changed to simply load the game and its actors and to run the game and manage the log. We moved the responsibility of rendering the grid and setting up the location of the grid elements to **<<PacManGameGrid>>,** which, as stated earlier, would allow us to adhere to the information expert pattern. **<<Game>>** class still provides indirection

between the game grid and sprites which is ideal as there is less coupling between these classes with common interactions.

4. The **<<GridManager>>** class is created using the pure fabrication pattern to manage all the responsibility of removing and putting items into the grid. This is an ideal choice as it promotes high cohesion by placing a similar set of responsibilities into a singular class and promotes extension. This is because when a new item is developed, the responsibility of putting it on the grid can be contained with the others.
5. **<<ItemType>>** enum class is created via pure fabrication to store information regarding each item/pill and their expected behaviour as it produces higher cohesion. An enum class is an ideal choice to store the information as they work well when there are limited options in both types and behaviour, which is the case with this game. It makes use of the information expert pattern, as all responsibilities regarding dealing with a specific set of objects are held within a singular class. With the original design, the information is scattered and needs to be well encapsulated so it does not promote reusability. Using the **<<ItemType>>** enum, we can store important information, such as item colour, and access it from the enum through a getter function. Essentially this allows for lower maintenance if we were to make changes as we would have to change the property of the value in the enum, and the changes would propagate to all functions using this information. We chose not to add it as a class as the items/pills have no additional attributes other than their colour and behaviour when consumed. Additionally, it allowed us to maintain the initial implementation of the eatPill() with the use of colours in a neater fashion. As each item has its unique behaviour when consumed, we used an interface named consumable with a singular function called consume. The function is then overridden for each item within the enum, and it is ideal as it promotes reuse rather than having the behaviour stored separately in the eatPill method. If a change or extension is made to the behaviour of when an item is consumed, it is much easier to alter and has less impact as the functionality is well contained. Interfaces combined with the enum are ideal as they promote lower coupling. The same function can be comfortably used with different items and would ensure they provide a different effect.

Proposed Design of Extended Version

Extended features are added with the following assumptions regarding the game logic.

1. Orion can walk through other gold pieces on his way to a specific gold piece. There is no definite path through which Orion must travel between gold pieces.
2. Ice cubes do not allow PacMan to pass through the monsters in its freeze mode, and he can still collide with them.
3. The time is not aggregated when the monsters are frozen and PacMan eats another pill.

Implementation of the design for extended features:

1. New monsters such as Orion, Alien, and Wizard are added as new subclasses of the monster superclass due to the use of polymorphism. The use of inheritance allows any new type of monster to immediately gain access to all the functionality that a monster should have. As stated previously, we ensure protected variation as adding new monsters does not require us to change any other classes, and we extend our current

design. When adding new monsters, we are only required to implement their walk approach and any additional functions or attributes, such as an ArrayList<> of visited gold locations for Orion and their filename for their image, which can be assigned to the constructor of the specific class.

2. The additional behaviour of the gold and ice can be recorded in the consume method as part of the interface they implement. This allows us to easily alter the behaviour or the ability of a particular item without causing any impact on the other classes as we have to extend the current consume function within **<<ItemType>>**, creating protected variation. The effect of this behaviour and the consequences can then be passed on to the actors via the Game class, which manages the game's running through indirection. Within the **<<Monster>> class**, we also implement a boolean that records whether a monster is frozen or furious to determine the associated effects when PacMan eats an item pill.
3. The use of a **<<Monster>>** superclass allows us to implement the behaviour when a monster becomes furious through the use of an interface called **<<furiousBehaviour>>** with the method furious(), which has two different variations depending on the type used to record the direction of the monsters movement. This is an ideal design for the extension as the behaviour of all the monsters when they become furious is similar. Using an interface within the abstract superclass allows all types of monsters that inherit the superclass to access that behaviour. As such, through the use of an interface, we are able to promote high cohesion as the responsibility is assigned via the use of the information expert pattern due to the monster superclass possessing the boolean. The boolean is used to determine whether the monster is furious. Additionally, the subclasses inherit the information from the **<<Monster>>** superclass. The use of polymorphism with the two variations of the method is also ideal, as it allows the comfortable use of the function.

Additional Design Choices

- Changed implementation of stopMoving to allow implementation of freezing monsters and stopping TX5 from moving in the same function
- Using an array of monsters via the monster superclass to easily access each monster without having to access them individually allows PacMan to check collisions using a simple loop instead of hard coding each monster. The loop is also used to set the values of all monsters instead of setting the values one by one, reducing the line of code used and making the system more modular, as we only have to add a new monster into the array. All conditions are checked using the array.
- Orion uses the TX5 walk approach with different targets, e.g., gold instead of PacMan.