SWEN30006 - Software Modelling and Design
Project Assignment 1: PacMan in the Multiverse

Report
Fri-13:00-Team 05

Background and Problem Statement

The current implementation of the PacMan game has poor design choices and therefore is not very conducive and suitable for extensions. Therefore, one of the major tasks is to refactor the codebase, to not only meet the software engineering design principles but also, to make it is easier to add more functionality to the game in the future (such as adding the extension of playing pacman in the multiverse), without the need for a major rewrite of the whole program. Building on this, the next task is to add the proposed extension of playing PacMan in the multiverse. Below is the analysis of the current design (P1), proposed new design of the current codebase (P2) and also the approach used for the extension of PacMan in the Multiverse (P3). Some important assumptions in implementing the extended codebase are also highlighted in the conclusion of this report.

(P1) An analysis of the current design:

Problems with Monster class:
In the current design, the monster class tests for the type of monster (Troll or TX5) and uses conditional logic in methods such as walkApproach() to perform alternatives based on the type of monster. Following the principle of polymorphism, this behaviour of a class is not an optimal way to assign responsibility to the Monster class. This results in low cohesion of the Monster class given it contains responsibilities for both the Troll and TX5 monster. Thus, we also identify the monster class as bloated.

If we were to add more monsters in the extended pac-man model, the Monster class and how it is structured, will struggle to cope with these additions. Specifically, in the method walkApproach(), the if-else statement is currently only set up to handle two different types of monsters. If we wanted to add more monsters, this would require more conditional statements to perform more alternatives based on different monsters, bloating the class. This realisation does not follow the principle of future variation. *This is further covered in (a) of P2.*

Problems with MonsterType class:
The MonsterType enum class is a rather strange class simply to return the image name based on the type of monster. This seems like an attempt to increase cohesion of the Monster class, when in reality it is unnecessary, and only just greatly increases coupling between Monster and MonsterType. If we were to add more monsters into the current design, this would mean that the programmer would have to update the conditional statements in both the Monster and MonsterType classes, which is unnecessary if good

design practices are followed. This suggests the current model is not future proof and does not follow protected variation.

Moreover, taking into account the principle of polymorphism,it is not recommended that a class have conditional logic to perform alternatives based on the type of monster in the first place. So, MonsterType class in itself is not the optimal way for a class to behave. *A solution to this high coupling and lack of protected variation is elaborated on in (b) of P2.*

Problems with Driver class
Currently the Game class is the only class responsible for calling the methods associated with an object of the gameCallBack class. But the Driver class is the creator of the initial gameCallBack object. This means that we must pass the gameCallBack object through the constructor when we initialise a Game object in the driver class. This is unnecessary and counterintuitive given the Creator principle. *This gives rise to another design flaw that can be amended in (c) of P2.*

Problems with the Game class:
Currently, the game class contains a lot of code that handles many different responsibilities. For example, responsibilities include, reading in and handling the parsing of pills, items, monsters and PacActor from the properties file. It is also responsible for drawing the grid given these objects. It appears that the Game class must perform a lot of set-up behaviour for these Actor objects before they can actually interact with the grid. If we interpret the game class responsibility as handling and interacting with the final and finished objects involved within the game sequence, including actors and grid, this class seems very bloated with responsibilities. In turn, suffering from low cohesion. This set-up-like behaviour for Actor objects should be abstracted away from the Game class and refactored to another meaningful class.

Given the game class directly reads and parses contents of the properties file, this results in high coupling between these two objects, which is a poor design choice given properties within the properties files can vary, and fails protected variation.

Furthermore, in the current design, If we add more monsters to the map, then it means that we have to add extra code to methods such as the setUpActorLocations() method. It would make sense to protect variation so that the game class does not need to deal directly with these additions in reading in properties files and parsing them. On a similar thought, if we want to configure many different game versions, such as simple and multiverse, the current game class would need to handle all this conditional functionality, also suggesting it does not uphold future variation. *These issues will be further covered in (d) of P2.*

(P2) Proposed new design of the simple version:

(a) We can use polymorphism to achieve protection at the variation point, which specifically occurs in the conditional statements within the walkApproach() method of Monster class in the current design. This is because more monsters need to be added to the game. We should create a polymorphic operation for each type for which the behaviour varies. This can be achieved through using a generalisation specialisation class hierarchy to organise these monster types. We can have a parent class named Monster, but using inheritance we can create children's classes of the monster types, like Troll or TX5. Given this new design, we never need to create an object of the Monster class directly, allowing the Monster Class to

become abstract. This parent class creates a common interface for operations that are shared between the types of monsters to be inherited from the parent class and reduces duplicate code. Then for each child, we can implement the different walkApproach(). This makes it easier to add in new types of monster in the updated version of the game by simply adding new classes - protecting variation.

(b) Given we have now implemented a class hierarchy for Monster, with the types of monster being the children, it makes it rather unnecessary to have an enum 'class' MonsterType to simply give us the image name conditional upon the monster type. This behaviour can simply be reassigned to each of the children classes, allowing us to remove this enum class entirely.

(c) Given our GRASP principles, the Game class should be the creator of the gameCallBack object because it is the only one to use the methods gameCallBack. Therefore, we can remove this instantiation of gameCallBack within Driver (reducing coupling to GameCallBack) and refactor it to the Game class. In turn, this simplifies the constructor of the Game class, given we are now only passing a PropertiesLoader object through from Driver.

(d) To achieve high cohesion in the Game class, so it is more focused on its core responsibility of running all the final elements of the game, we can delegate some of its tasks, namely reading and parsing the properties files to another more suitable class. We can use the PropertiesLoader class to do this, following the principle of indirection, and because it is the information expert of reading and parsing the properties file. Methods such as loadPillAndItemsLocations() and setUpActorLocations() can be transferred to this class. This makes the Game class more focused, easy to understand and manageable. This also means with new additions of properties in the extended version of the game, we don't complicate the Game class code. Instead, the handler of this behaviour, the PropertiesLoader class can be amended.

(P3) Proposed design of extended version

With the proposed refactoring of the simple version in P2, this protected against future variation and made the implementation of the extended multiverse game much simpler. Therefore the only real major change to the design of the model in P3 was the need to split the facilitation of running the simple vs the multiverse game types.

Given we now have two different ways to run a game, simple and multiverse, the Game class will require conditional logic to perform varying alternatives based on the different types of game. This includes alternative behaviour such as additional monsters or how the monsters react to PacMan eating items. This will result in the Game class being bloated with many different conditional tasks, lowering cohesion. Following the principle of polymorphism, a class should not test for the type of a Game object and behave like this, suggesting that the Game class in itself is not assigned the correct responsibilities. In turn, we can use a generalisation specialisation class hierarchy to organise the game concepts, whereby we can have children classes named MultiverseGame and SimpleGame, and common interfaces between the two can be inherited from the abstract parent class 'Game'.

Given the conditional behaviour of the two types of game is located within the constructor method of the Game class in the old design, we separated this behaviour from the constructor and made a separate method run(), to handle the running of the game logic. This method then becomes a polymorphic

operation for SimpleGame and MultiverseGame classes, given this is the behaviour that varies between game types. In saying this, an abstract run() method is applied in the parent class Game, forcing the children classes to implement it.

Therefore in the Driver class, conditional upon the command-line arguments, either a SimpleGame or a MultiverseGame object will be created.

Assumptions:

Below are the assumptions we made during the refactoring of the codebase and implementation of PacMan in the Multiverse.

- Frozen Monsters: When PacMan eats an IceCube, it freezes the monsters for 3 seconds (a timer starts). And whilst they are frozen, if PacMan eats another IceCube, it doesn't reset the 3 second timer or add another 3 seconds - it just freezes the monsters for as long as 3 seconds from when the first one was eaten.
- Orion walk Approach: Following the specifications, Orion picks a gold piece at random to move towards. This won't change until it reaches that gold piece, but if it can't walk in the direction of the gold piece, it will perform a random walk until it can.
- T-X5 in the specifications should not move for the first 5 seconds. Any ice cubes eaten during the first 3-5 seconds will not impact the T-X5, and when the 5 second wait time for T-X5 has finished it can move, although other monsters remain still.
- In testing the extended version of the game, the alien Monster, following PacMan eating a coin and making the monster furious, can potentially experience a location in the grid where it cannot move in any direction for 2 cells. In this rare situation, we enabled the Alien monster to revert back to its one step walkApproach for one step until it is able to move in a direction towards the PacMan for two steps.

Figure 1:

Domain Model

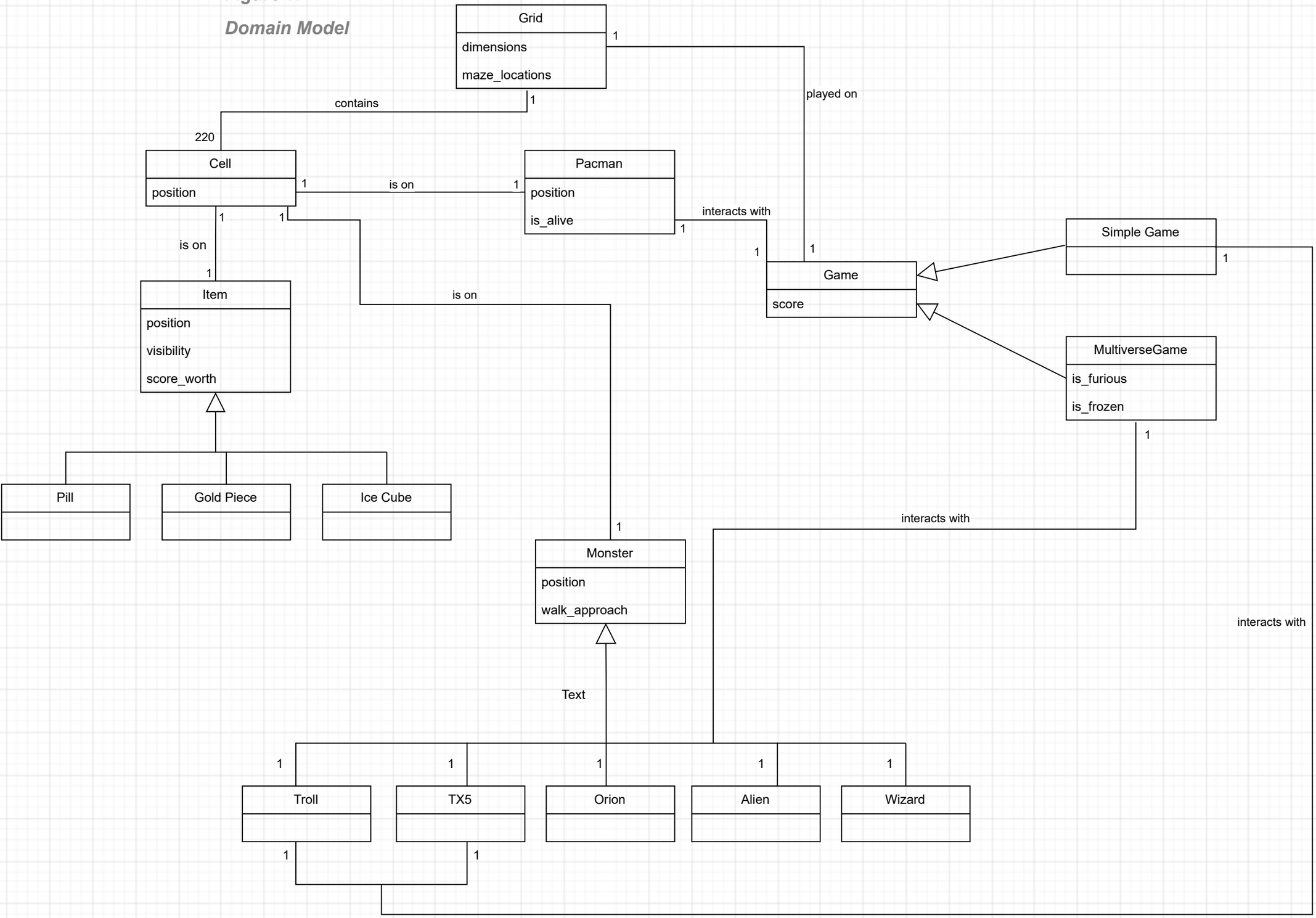

Figure 2: Static Design Model

src

utility

PacManGameGrid
- nbHorzCells : int
- nbVertCells : int
- mazeArray : int[][]

+ PacManGameGrid(nbHorzCells : int, nbVertCells : int)
+ getCell(location : Location) : int
- toInt(c : char) : int

GameCallback
- logFilePath : String
- fileWriter : FileWriter

+ endOfGame (gameResult: String)\
+ actorLocationChhanged (pacmanLocation : Location, monsterLocations : List<Location>)

PropertiesLoader
+ properties : Properties

+ PropertiesLoader(propertiesFile : String)
+ loadPropertiesFile(propertiesFile : String)
+ getActorLocation(actor : String) : Location
+ getPropertyPillLocations() : ArrayList <Location>
...

Played on
has
Loads From

PacActor
- score : int
- visitedList : ArrayList<Location>
- isAuto : Boolean
...

+ PacActor(game : game)
+ keyRepeated(keyCode : int) : void
+ act() : void
- moveInAutoMode() : void
- canMove(location: Location) : Boolean
- eatPill(location: Location) : void
...

Played By

Game
- nbHorzCells: int
- nbVertCells: int
- pillAndItemLocations : ArrayList<Location>
pacActor : PacActor
troll : TrollMonster
tx5 : TX5Monster
...

+ Game(propertiesLoader : PropertiesLoader)
+ *run() : void*
+ getPillAndItemLocations() : ArrayList<Location>
+ removeItem(type : String, location : Location) : void
- drawGrid(bg : GGBackground) : void
...

SimpleGame

+ SimpleGame(propertiesLoader : PropertiesLoader)
+ run() : void

Driver
+ DEFAULT_PROPERTIES_PATH : String

+ main(args : String[])

Run By

MultiverseGame
- orion : OrionMonster
- alien : AlienMonster
- wizard :WizardMonster

+ MultiVerseGame(propertiesLoader : PropertiesLoader)
+ run() : void
- freezeMonsters() : void
- furiousMonsters() : void

Contains

-goldPieces
-iceCubes

ch.aplu.jgamegrid

Location
...

+ act() : void
+ getDirection() : Location
+ setDirection(direction: Double) : void
+ setLocation(location: Location) : void
+ turn(angle: Double) : void

Actor
...

+ getX() : int
+ getY() : int
+ get4CompassDirectionTo(location Location) : CompassDirection
+ setDirection(direction: Double) : void

TrollMonster

+ TX5Monster(game : Game)
# walkApproach(isFurious : boolean) : void
+ toString() : String

TX5Monster

+ TX5Monster(game : Game)
# walkApproach(isFurious : boolean) : void
+ toString() : String

OrionMonster

+ OrionMonster(game : Game)
# walkApproach(isFurious : boolean) : void
+ toString() : String

AlienMonster

+ AlienMonster(game : Game)
# walkApproach(isFurious : boolean) : void
+ toString() : String

WizardMonster

+ WizardMonster(game : Game)
# walkApproach(isFurious : boolean) : void
+ toString() : String

Monster
- visitedList : ArrayList<Location>
- stopMoving : Boolean
- isFurious : Boolean
...

+ Monster(game : Game, imageName : String)
 # walkApproach() : void
moveRandomly(direction : double) : Location
+ fastMoving(seconds : int) : void

Figure 3: Dynamic Design Model