SWEN30006 Software Modelling and Design
Project 1 Report (exactly 4 pages excluding diagram)
Group 3 Monday 17:15

Section 1. Analysis of the current design and limitations

- There are multiple interactions between Game and many other primary classes (Monster, Actor, etc…) - **low coupling**, however, it is unavoidable because Game holds the control to every class to run the game
- Items such as gold, ice cubes, and pills are now inherited by the Actor class, which is not reasonable. Actor class contains other responsibilities which items do not possess, turn() for instance. Notice that this does not facilitate further extension when taking into account that they do have an effect on other Actors.
- Monster class does not represent a high level of polymorphism as it can only be implemented by 2 types of monsters: Troll and Tx5. In specific the walkApproach() functions only deals with 2 types of monsters; however, in the future, if upcoming types of monsters have different behavior from these two, re-implementation and refactoring can be challenging, making it hard for extension.
- There is no **Controller** for handling input and setting up from the .properties files, which makes the game very difficult to handle input from different types of files later on, .txt or .csv file for instance.
- Moreover, Game tries to cover many methods which it should not have (**Information Expert**). Methods to check for End Game conditions, checking collision between monsters and the pacActor, are implemented by Game constructor (wrong responsibility + place), while the true responsibility should belong to pacActor with a correct implementation of hitEnemies(), which raises concerns regarding **Information Expert.**
- Although Game is the **Creator** for its constructor, the actual run() and loadItems() functions are located in the wrong place, which should have been split into 2 separate methods instead of being located in the constructor, leading to inconsistent separation of responsibilities. If we extend and all the methods are still kept in the constructor, new implementation might be challenging as finding location of the new method in implementation is difficult. Therefore, Game is highly incohesive, in which it has to handle unrelated responsibilities
- Therefore, gold, pills, and ice cubes should extend a superclass called **Item** from which they could inherit with mutual methods (effectsOnPacActor, effectsOnMonsters) and attributes (sprite image file path, etc), which facilitates other attributes in future.

Section 2. Proposed refactoring and new design of the simple version

- ***Actor*** is extended by only 2 main types of character in this game: PacActor and Monsters, which are movable objects.
- Reassign responsibility for the actor to check for collision with monsters instead of the Game class, increasing cohesion.

Monsters

- Monster is a subclass of Actor with emphasis on act() and walkApproach() methods, and all types of monsters (Troll and TX-5) are subclasses of Monster (instead of using enum, since subclasses will give unique actions by each monster type). Hence also removed all MonsterType related attributes (getType) and methods from Monster class. This will facilitate future monster types extension, because new monsters can inherit Monster abstract class and override the act() and walkApproach() method.

- Troll and Tx5 have the same pattern of walking approach: randomWalk, which is the main walk approach for Troll. Therefore, it is kept at the Monster abstract class level, and Tx5 can utilize this randomWalk approach when it cannot move to its desired direction.
- Moreover, for the extension, also note that Alien and Wizard can also move in 8 directions; therefore, an interface MoveFullDirection with defined constants is also implemented to assist both design (future monsters can walk in 8 directions) and implementation purposes. An optional interface is MoveThroughWall Interface, which allows monsters to go through the wall (Wizard, and potential future monster types).
- This will **decrease coupling (**with Game class and Item class, etc**) and increase cohesion through polymorphism.**

Items

- We will create an independent abstract class called **Item**, which does not depend on the Actor class. Gold, pills, and ice cubes extend this abstract class from which they could inherit with ***mutual methods*** (effectsOnPacActor(), effectsOnMonsters()) and ***attributes*** (visibility sprite image file path, etc for visualization purposes). For the future, these items can override these methods to inflict different effects on characters (both monsters and pacActor), utilizing methods in abstract class.
- Example, the current version only requires pills to increase the score for PacActor; however, effectOnPacActor() is also ready for future changes.
- Hence, we made an adjustment in Game class, for Goldpieces ArrayList and IceCubes ArrayList, we alter using Object as GoldPieces object and IceCubes Object instead of Actor Object.
- This will **decrease coupling** (with Game class, Monster class, etc**) and increase cohesion through polymorphism.**

Game

- **Controller:** A controller is used to handle the input from the .properties files, and the game is run based on the controller's playGame method. This controller is mainly responsible for handling the given data (in .properties file) and triggering the Game instance to start and play the game. This is useful in the future if there are any changes

in the data format provided, which **reduces coupling (**with Game**) and increases cohesion** via Pure fabrication (not originally in domain model)**.**

- **A controller** for counting is also added, which increases the cohesion and reduces coupling for both Game and **Counter**, which is initially not in the domain model (*Pure fabrication*) . This facilitates future code as there might be more desired features to count.
- A **controller** for setting up Pill and Item locations: **ItemSetter** is also added to separate this unrelated responsibility from Game via Pure Fabrication. This increases cohesion for both classes and reduces coupling.
- A **controller** for víualization purposes: **VizController** is also added to separate this unrelated responsibility from Game via Pure Fabrication. This involves methods such as drawGrid,..., which increases cohesion for both classes and reduces coupling.
- A list of monsters and items to perform their own responsibilities. This will increase the cohesion, and reduce coupling in both Game and Monsters classes, in which they are more focused towards their responsibilities:

```
for (Monster monster: monsterList){
  monster.setStopMoving(true);
}
```

- In the original version, all Game behavior (running, ending) is grouped inside Game constructor, the constructor might be bloated if we extend further behaviors. Hence we choose to split the initial constructor into various methods including: playgame(), checkEndGame(), printEndGame(). This helps assign better responsibilities for each method, which facilitates reproducibility and high cohesion.

Section 3. Proposed design of extended version

Game

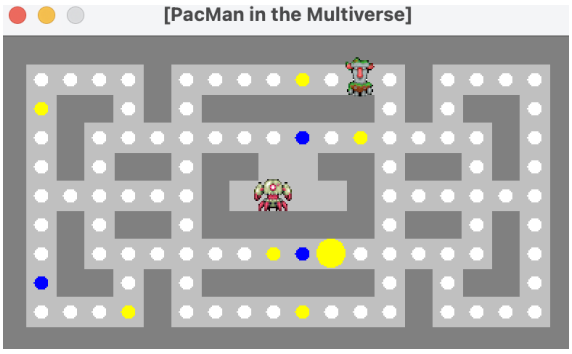
- To keep track of the configurable version, an attribute called state is added to Game class.
- A controller is implemented based on the description above.

Monsters

- Monster abstract class is inherited by 3 new monsters Orion, Wizard and Alien. In this version, randomWalk() and randomWalkFurious() are still kept in the Monster abstract class to avoid the state at which these 3 monsters' locations remain unchanged.
- Monsters will override act() and walkApproach() based on their own behavior, and depending on the state ("normal" or "furious"), it will determine its own walk approach.

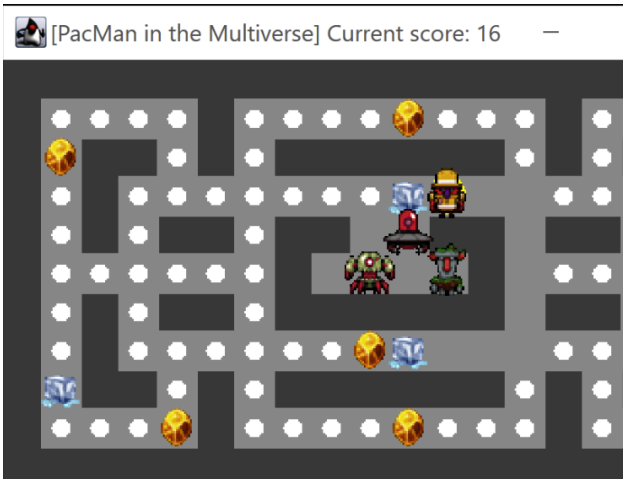
Assumptions

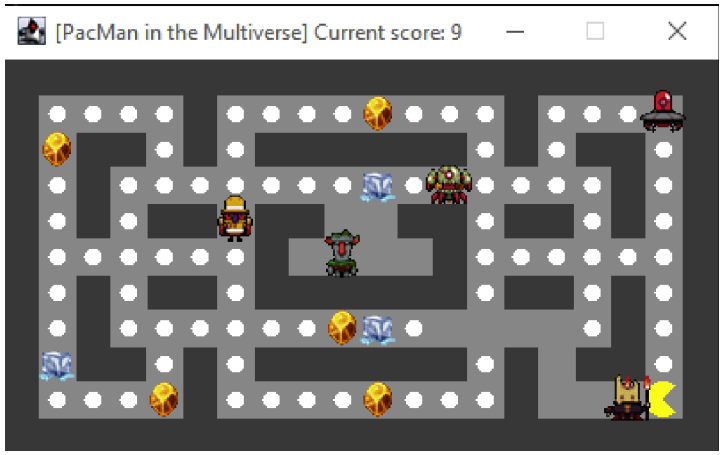
- **Orion:** can move in all 8 directions instead of 4 directions to avoid the state at which it got stuck in a loop finding the gold piece, which makes it unable to go to all the gold locations. In the example below, Orion (red) will always choose to go down in the loop to get to Gold (Yellow)



The gold locations will be sorted based on its availability, and a visitedGoldList is also implemented to ensure Orion will visit all gold locations and not revisit visited ones.

- **Alien and Wizard:**
- Based on the description, Alien will search for the nearest neighbor to PacMan, while Wizard will go towards a random direction until it can. Therefore, both of these monsters will implement an interface called MoveFullDirection which allows them and future monsters (extension) to iterate through 8 directions to determine their next destination.
- It is a **rare case** that either Alien or Wizard cannot move; however, in the case at which they cannot move, we may allow for no change in location, or implement the randomWalk in abstract Monster class.





In the example above, Alien (furious state) cannot move in any directions because they are fully blocked, while in the second picture, the Alien is also stuck in the corner due to constraints in the history path. Therefore, checking for history path and remaining unchanged in location are 2 additional points for corner case for extension version.

Item

- In the second section, Item is supposed to be an independent class; however, during our implementation process, we have no choice but to make it inherit from Actor class.
- As explained in Section 1, Actor class is limited in extracting and splitting, so we make Items extend Actor, to import the functionality of Actor class. We tried to implement Items on its own, by copying and pasting some functionality of Actor class to read and import images to Gamegrid, but there are limitations as Item is in the src package (not j.gamegrid), also methods in Actor class are protected, hence we chose to extends Actor.

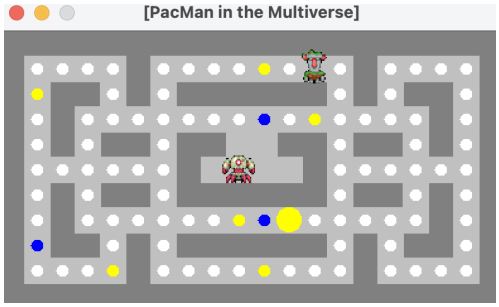


Figure 3.3: Example of implementing Item directly without inheriting Actor class, causing mis-visualization due to requirements about not changing provided package-based code.

- Moreover, the main method responsible for visualization is addActor(Actor actor) method from Gamegrid() which only accepts Actor instances. We also tried to re-implement this method, yet there are so many variables to keep track of, forcing us to inherit the Actor class.

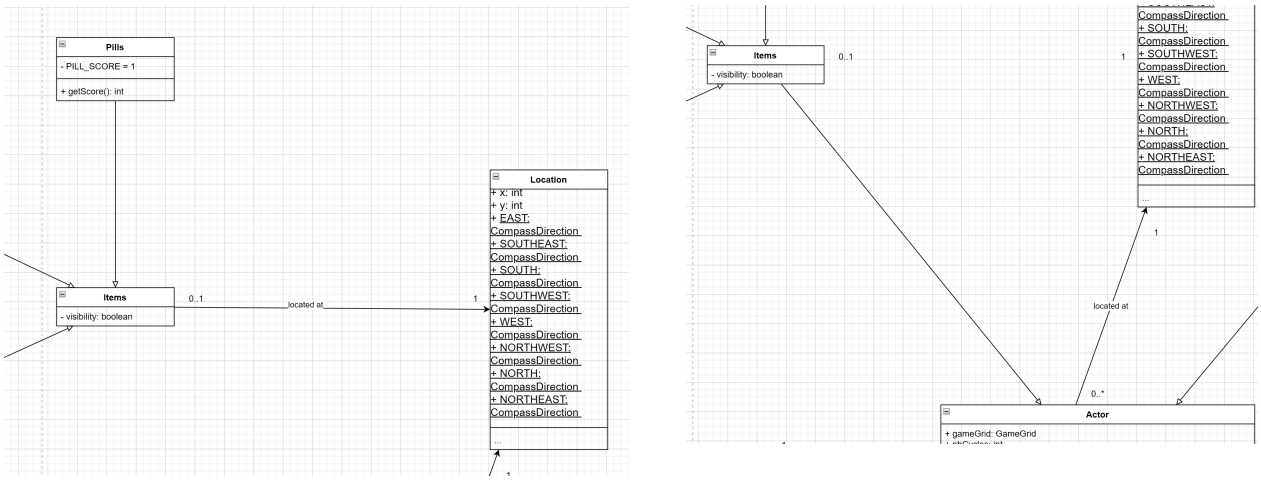


Figure 3.4: Changes in terms of design class diagram

- Gold, Pills, and Ice cubes will override the effects on both monsters and pacActor, based on its behavior. Moreover, additional features: PILL_SCORE or GOLD_SCORE, for example will also be added to each class: Pills and Gold, respectively, depending on its properties.

Model assumptions

1. In the Static Design Model, multiplicity of Game and Monster, we assume that when extending the game, there will be multiple monsters in one monster type (e.g. 2 Aliens), hence we put 2..* instead of 2..5
2. Assume one location can be visited by many characters
3. Assume all items will be allocated in different location, cannot be on the same location hence the multiplicity will be 1 and 0..1 between Location and Item
4. Assume that there is only 1 pacActor in the map.
5. Not included in Domain Class Model, we considered create an interface eg. “Moveable Against Wall” for Troll and TX-5, we consider not to put it in Monster class because other extended monsters such as Orion, Wizard and Alien will inherit it.
6. No dependency between Location and Game because Game does not use Location directly, it can be accessed through Actor/Character (optional)