Pacman into the TorusVerse Redesign

Felicia Suryawinata 1297668
Liam Anthian 1269187
Trevor Wei Hong Poon 981078

University of Melbourne
Thursday Workshop 13:00, Team 05
SWEN30006
26 May 2023

Problems with the previous code

The original code supplied had numerous issues that would limit the extensibility of the solution to different requirements / further extensions. A large number of these issues were resolved in this project's prior, and so will only be summarised briefly below:

In the original design, many classes - such as game, the representation of items, monsters, etc. - had very high coupling design wise and resultantly low cohesion in the solution space. Key examples include the fact that all item instances existed as just an instance of the JGameGrid Actor class, with overlapping, repeated, and convoluted behaviour. This design was difficult to extend to new requirements like the autoplayer developed here, and was rectified by splitting the items up into different classes that extended the base Actor class. The same solution was applied to the Troll and Tx5, seeing the Tx5 instance as just an extension of behaviour on the base Troll. This is an example of the composition GoF design pattern and will be discussed further later.

As before, the Game class also had the bulk of the code in it, very highly coupled with all classes (coupled both ways) to manage the whole system. In the project 1 solution, methods and code were delegated out of Game and into the most suitable location as per the Information Expert GRASP principle. This approach has been reconsidered and tackled to a greater extent in this project.

A new problem from the additional code provided was that two detached applications that needed to run together (map editor and pacman) were supplied, completely unlinked. Although resultantly coupling between the systems was very low, these elements needed to run together and couldn't do so in their current form. Some form of GoF facade or adapters would need to be implemented to act as the middle man between these systems without having them fully merge and ruin cohesion and coupling goals.

Two main issues that arise from this fact:

- Pacman / PacActor would need to be able to play/test numerous levels ("Game"s, referred to as levels from this point forth), and logically there can only ever be one pacman instance at a time. Additionally, keeping extensibility in mind, the pacman between levels should maintain values and information, but in the current form of creating multiple different pacman instances this would be highly difficult and involve unnecessarily complicated parameter parsing.

- The maps created, loaded and saved in the editor would need to be read into the testing environment and actually played as levels, however the current code always read and implemented the same level from PacManGameGrid, and changed minute details from an overly complex properties file. This ".properties" management then bled throughout the code with many arrays for read-in values parsed back and forth, especially prevalent in the Game class. Furthermore, as the given solution currently stood, the levels saved from the editor had no way of being read into the Game class to be played, and thus some reader class would need to be introduced in between as a link.

Modified design and solution

As is often the way when applying design patterns to a solution, there were numerous directions to take the project with different outcomes for understandability and efficiency. The below are some of the main decisions made, and are by no means an extensive list of potential improvements.

As hinted at in the "Problems…" section above, the design improvements from the previous project were mostly carried over in full to this project. The first project implemented the composition design pattern through the troll-tx5 relationship observed, with a tx5 basically being a smarter troll, and while the pure abstraction of the super Monster class made monster generation and management a more streamlined process for Game there was still room for a little extra improvement. Through implementing a Monster Factory, different predefined monsters can easily be generated from wherever (just Game in this case) without knowing the basis of how these monsters differ in construction and behaviour. Resultantly, Game now only needs to know how to handle an abstract Monster and call the factory to generate this instance. The Factory has been designed as a singleton, as behaviour between monster factories is identical and thus numerous instances aren't necessary or useful.

Following from this, an Item Factory has been introduced to manage the creation of the different items separated in project 1. Originally treating each non abstract item instance (gold, pill, ice) as an extension of similar behaviour stored in Item, redundant and duplicate methods were reduced. This time, through handling construction with enums (ItemType) in the Item Factory, each extending class is viewed more as a strategy – leaving all specifics of generation to their super Item and factory – overloading the aforementioned shared behaviour. Just like with Monster, Game now only needs to know the general methods shared between items to handle them. This is highly extensible, as new item types can easily be introduced to the factory, the overloading strategy implemented, and read in via (the later mentioned) GameMap and they're suddenly good to go.

One way to handle the overly bloated and low cohesion Game class was to group similarly behaving game elements together via an interface:
Addressing how actors know about other kinds of actor types and the behaviour they exert towards each other, an ActorCollidable interface and accompanying ActorType enum has been introduced. Each actor deemed "collidable" then has specific behaviour whenever a collision is detected between them and each actor type. For example, when Pacman collides with some item (implements collidable) pacman calls it's collidable behaviour, determined in the specific strategy applied to the item (based on type), letting it know that a Player has interacted with the Item. Whereas when a monster collides with an item, it lets it know that a Monster has interacted with it. This way, different collidables can respond differently to unique collisions, but also similar (e.g. portals interacting with pacman and monsters the same way). This is an example of a bit of a hybrid between strategy and adapter design patterns.

- It should be noted that the above increases the coupling between different classes a significant amount. To deal with this in the future, the adapter GoF design pattern can be fully fleshed out to move this interweaved interaction into a separate easily swappable location.

Another great help on the cohesion of Game was the reduction / effective removal of the Properties class dependencies throughout the solution. Through fleshing out level loading separately and handling pacman auto movement in a new smarter way, all the property storing and interacting arrays could be removed, greatly improving readability.

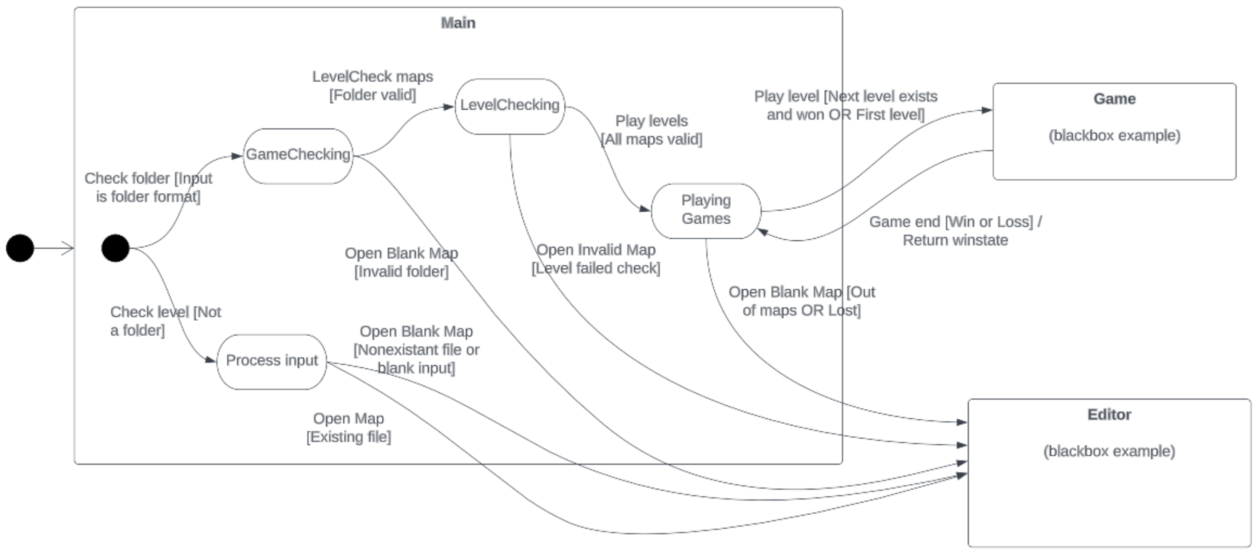
A final change in the viewing of Game was seeing it almost as a facade of the actual system present. While the code itself wasn't particularly altered (other than the above interface change), Game itself does very little for the game elements. Having moved the gamestate (Win, Lose, Active) changing to the collidable interface and movement checks inside the changing game elements (pacman, monster), all Game needs to do now is construct and place the game elements, let them run on their own, and check every set interval if the game is complete (pausing and closing then). While it additionally holds methods utilised between classes to talk to each other and change state (e.g. updateActor), it itself just facilitates this communication and is a much simpler root for interacting with the system then it initially was. This approach was teased as an improvement in project 1 but not implemented or seen the way that it has been this time.

New features

A main consideration that motivated how maps were read from the editor to the testing environment (playing) was the association present between these two halves in the first place. While information passed out from the editor has to be able to be read into the testing, this relationship does not go both ways – just from Controller to Game. Thus, the existing XML level output of the editor portion was deemed acceptable for use for Game, given it's read in and validated somewhere first.

A GameMap class was procured to read in this XML into a format desirable and interpretable by the existing game. While this design itself is fairly straightforward, a few useful things were done. Firstly, the implementation of all the type enums was done here, and thus resultantly for the rest of the code any later time a collidable interface or a constructing factory needs to differentiate between types it knows it'll only get objects of the right type. Additionally, level checking (mentioned later) was done here to guarantee that any GameMap instances called later could be trusted valid, allowing reliable implementation simplifying assumptions (especially handy with portals and teleportation).

This GameMap class ended up being one of the key bridges between the two systems, easily taking output from the editor and returning an object usable for level set up in Game instantiation. Furthermore, it was especially useful in the Main class, the actual bridge between editor and testing, called by the Driver on program start. The Main class would receive one of three inputs, seen in the state diagram supplied, and then handle editing and testing appropriately. In this way, the testing and editing environments never have to explicitly know about each other and remain uncoupled, and thus Main acts as a facade of both, handling just Controller, GameMap, Game, and a little bit of PacActor to pull everything together.


Main
LevelCheck maps [Folder valid]
LevelChecking
Play levels [All maps valid]
Play level [Next level exists and won OR First level]
Game
(blackbox example)
GameChecking
Check folder [Input is folder format]
Playing Games
Game end [Win or Loss] / Return winstate
Open Blank Map [Invalid folder]
Open Invalid Map [Level failed check]
Open Blank Map [Out of maps OR Lost]
Check level [Not a folder]
Process input
Open Blank Map [Nonexistant file or blank input]
Open Map [Existing file]
Editor
(blackbox example)

Note in the above diagram, GameMap is accessed from the Level Checking state, and PacActorSingleton from the Playing Games state.

Following the idea from earlier that pacman should really be one instance across all the games, the singleton design pattern was an easy fix. Although mostly underused in implementation as GameMap already checks that there's only ever one pacman, and all attributes are currently reset between Game instances, this pattern is in accordance with the concepts of the PacActor suggested in the design specification. Additionally, it makes extension of scope and how *the* pacActor interacts with games that are running quite simple, with a few easy changes in the pacActor getInstance() method.

Portals

A brand new game element added to the system, portal design was deliberated over for a period of time before implementation. Using the fact that portals only ever come in pairs or not at all, and the knowledge that level checking (seen below) will always precede actual game element construction (non valid portal pairs will never be parsed for construction), portals were paired together by type (colour) in the class PortalPair. PortalPair acts as the access point for the Game and game elements to interact with portals, and they follow the implementation of the earlier ActorCollidable interface, meaning that they would do well to have adapters introduced in the future. A PortalPair instance starts empty and can be (and should be) filled with two Portal instances, constructed from a given location. This way, according to the information expert principle, the class with the most knowledge on how to handle portals and what they actually are is PortalPair and thus everything can access portals via PortalPairs.

Checking

The GameChecking class determines what behaviour to follow given an input argument sent through at startup. It checks if the input is a folder, a file or as an empty argument. When not passed a folder, if an existing map file is passed through (whether valid or not), the editor will open up and load this file. If the map is invalid via levelChecking these details will be printed to a log file and the map opened in the editor. Otherwise, if the file is not found or input is an empty argument, it will open the Map editor on a blank page without a current level.

Given a folder is sent through, it checks that the contents of the folder are valid map files. A log file is produced if no valid (by name, not content) maps are found or there are multiple maps having the same number, and a blank editor is opened. Otherwise, given success, it checks the validity of each level via levelChecking. Should all levels pass this checking they are run sequentially as separate GameMaps passed to separate Game instances, all sharing the one PacActorSingleton. If this is not the case, the first numerically ordered map to fail is opened in the editor.

Given the folder is loaded successfully, the levels all pass checking, the games are run until pacman is defeated by some means or clears all the levels. The application returns to the map editor (a Controller) with an empty map regardless of this outcome.

LevelChecking, occurring inside the GameMap level reading, begins with reading the corresponding XML file and examining whether the map possesses the required attributes for playing the game that can be won or lost. If a level fails any check, it will be recorded into a log text file with that corresponding map name, and the validity flag of the GameMap instance will be set to false. This isValid method then found in GameMap is used inside the Main class when checking the validity of each level read in successfully via GameChecking.

Autoplayer

The autoplayer calculates the closest loot item to Pacman when it spawns, utilising a breadth first search (requiring $PacMan.isAuto = true$ in the read properties file to be active). It then follows this path until reaching the loot, then calculating another path once again. It repeats this process until all loot items have been consumed. This autoplayer design could be modified to handle ice cubes and monsters a couple of ways.

Instead of following the entirety of a path to a decided item, the autoplayer could recalculate this path each update, making only 1 step at a time. Then, applying different weights to certain game elements could have pacman choose different paths depending on surrounding context - e.g. preferring slightly further away gold over close pills as the gold is worth more for score.

In the event that the next step/set of steps towards the nearest item location is taken up by a monster or if the monster's next move will take it to that location, Pacman will opt to move in a differing direction in order to maintain a set buffer between itself and the monsters (e.g. 2 tiles).

Assuming an ice cube freezes the monsters for a certain duration, the buffer between Pacman and the monster could be reduced to 0, treating the monsters as some other non-passable game entity such as a wall instead. This way, Pacman is able to get close to the monster and collect the items around it. If the freeze duration is close to ending, the buffer will be restored prematurely to avoid collision and urge pacman to restore a safe distance.

These aforementioned weights could then be dynamic, scaled by the surrounding circumstances. For example, the autoplayer could give greater weights to ice cubes when monsters are within Pacman's buffer zone. By mapping out paths further than just the closest loot and then scoring them based on their weight, more intelligent and monster adverse behaviour can be built. For example, let's say that a nearby monster scales the context value of ice to 4, and pills keep their score value of 1. Considered the following two paths:

- A pill 6 tiles away has a preference value of 5, obtained by getting its distance to pacman, 6, subtracted by its score value 1.
- An ice block 8 tiles away has a preference value of 4, obtained from subtracting 8 by its context weight 4.

Treating 0 as the optimal preference, in this context, the further away ice is more valuable to the autoplayer due to the close danger. To extend this further, ice weight could be multiplied by the number of nearby monsters as well, prioritising escaping monsters over collecting loot.

Design wise this would require a new calculator of sorts to determine the value of each tile along a path. Applying a strategy method to the pacActorSingleton class would be a good way to go about this, swapping between manual control, default autoplay and this more complex autoplay based on game start parameters.

Additional Notes

The task below has been omitted from design due to paradoxical and mixed design signals from the spec and staff. See the below Ed Discussion posts.

2.2.2 5. If Test Mode is started with a specific level/map, that level is played until PacMan is destroyed by a monster or that level is completed. The application will return to Edit Mode on that map.

- https://edstem.org/au/courses/11685/discussion/1392511
- https://edstem.org/au/courses/11685/discussion/1398384

Main.java in Map Editor

- If Game crashes from end calling method .endgame(), able to change to .hide() inside Game.java