Assignment 3: Design Patterns

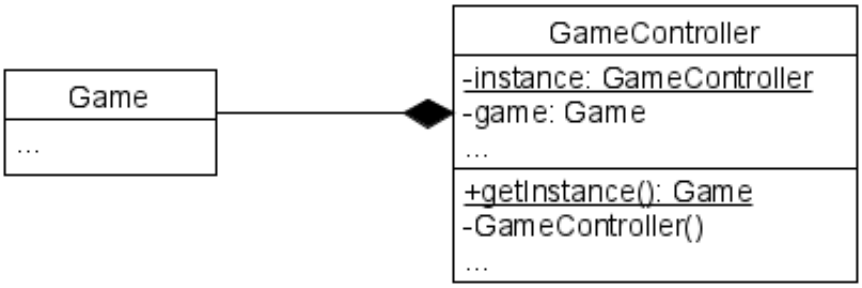
SEM Group 25

9th January 2020

1 Exercise 1

1.1 The Singleton Pattern

1. We used the singleton pattern for our `GameController`. There only needs to be one instance of this class because it manages a `Game`, which is used for displaying graphics and keyboard input. These are things that we would rather only have happen once, otherwise it would cause undesired behaviour. The pattern itself is quite simple: we just have the `GameController` contructor private so no other class can instantiate it. Then there is a private static field, this is the instance of the controller and a public static method that retrieves this instance. This method ensures that only one instance of the class is ever created.

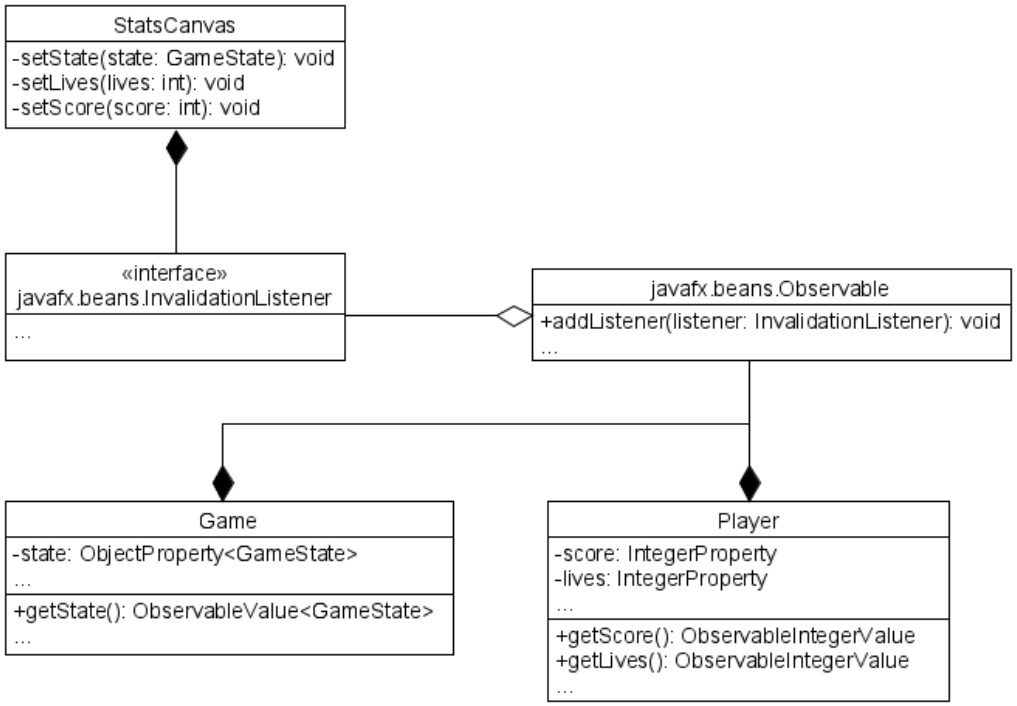


2.

3. For implementation, see the `pacman.logic.game.GameController` class.

1.2 The Observer Pattern

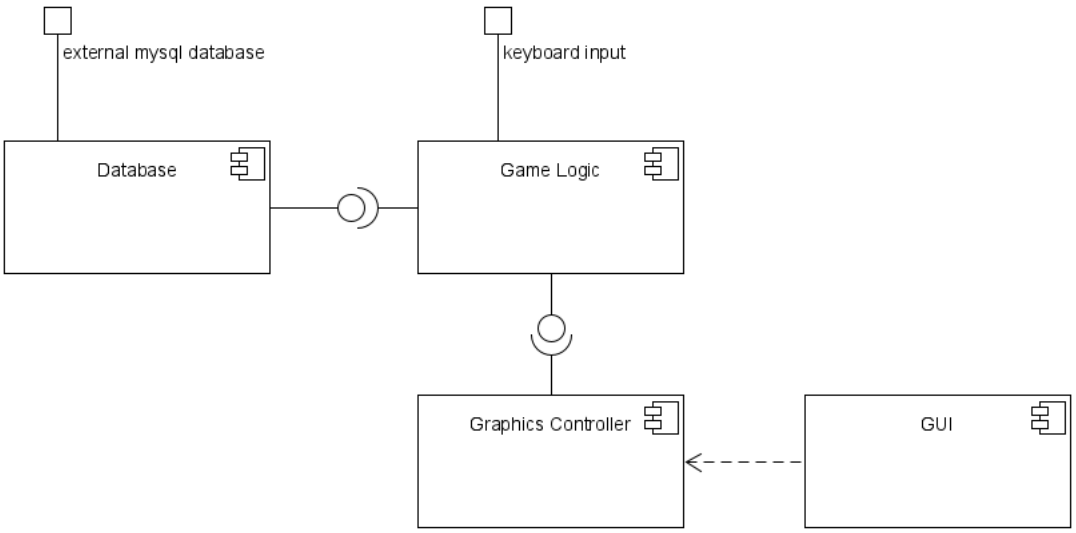
1. We used the observer pattern for some classes in our `graphics` package. For reasons explained in the Software Architecture section we want to have our graphics as loosely coupled from our game logic as possible. Therefore when something needs to be drawn on the screen, for example the player's score, we have an `ObservableValue`, which we can attach listeners to that will call some function whenever the value updates. In the example case this would be the `setScore` function in the `StatsCanvas` class. We use the javafx builtin observable classes: `ObservableIntegerValue`, `IntegerProperty`, `ObjectProperty` and `ObservableValue`, which all inherit from the `Observable` interface.



2.

3. For implementation, see the `pacman.logic.game.Game`, `pacman.logic.Player` and `pacman.graphics.StatsCanvas` classes.

2 Exercise 2



2.1 General architecture

The general architecture is model-view-controller. In our case the model is stored in the database long term and short term in the game logic. The user directly influences the game logic by using the keyboard and the mouse. What is displayed on the screen and what should be controlled by keyboard and mouse is determined by the graphics controller. This is because we use javafx with fxml. For every scene we want to create, we create an fxml file using an external tool. Then we write a controller class, which is the first class that gets control. Then everything is drawn to the GUI and logic is handled by the game logic. Things like loading levels, moving entities, determining win/loss is all game logic. The game logic also uses the interfaces provided by the database to write to and read from the database. The game logic and the graphics logic together form the controller and the GUI is the view.

2.2 Database

We use an external mysql database and jdbc to connect to that database. We have several classes that handle connections to the database and all have a certain purpose, e.g. updating leaderboard, logging in / registering. This allows us to access the database simply by calling one or two methods.

2.3 Game Logic

This is the largest and most complicated component. Its most important purpose is managing a game instance. It updates the entities, moves PacMan when the user presses keys and determines when a level is won or lost. It also provides an interface to the graphics controller to allow the graphics controller to see the game state and player's score and amount of lives. We achieve this using the observable pattern as described in section 1.2. Having the graphics logic as separate from the game logic as possible allows us to focus on the implementation and not worry about the look of the game. We can change basically the entire graphics controller without it influencing how the game works. Another benefit is that we can without too much trouble add quite complicated features. We would change the game logic and as long as we do not touch the interface, the graphics could remain unchanged.

2.4 Graphics Controller

The graphics controller's main job is initializing everything in its proper state as soon as it gets control after javafx initialization. It makes sure the proper setup functions in game logic are called and the GUI is ready to be drawn. It makes use of the observables in the game logic to update labels and graphics in the GUI. Also it loads all the fxml files when the game is started.

2.5 GUI

The GUI is mostly handled by javafx, but we can determine how everything looks by providing fxml files. The GUI makes sure that everything is drawn correctly. It can draw the graphics accordingly if the window is resized too. We mainly use javafx canvases to draw 2D graphics, we can draw any shape in any colour. We also have several sprite classes, that determine how every entity should be drawn. Javafx also allows us to use the built in buttons and labels and give them a custom style using css, which the GUI draws in the provided style.

2.6 Motivation

The main reason we chose this architecture is the ease of use javafx provides. However javafx almost forces you to use the model-view-controller architecture when used with fxml, because of its controller classes. The benefits we gain from using this architecture mentioned before, i.e. having independent game logic and graphics, which gives flexibility in programming, ultimately made us decide on this architecture.

3 Exercise 3

For the last sprint we decided to focus on polishing the game so it really looks finished. We decided that we would add some small final features, but we are not working on anything big. Removing bugs is also one of our priorities.