TDT4240 – Software Architecture
Assignment 2

Dionysios Rigatos
dionysir@stud.ntnu.no

Part 1 – Pong Game

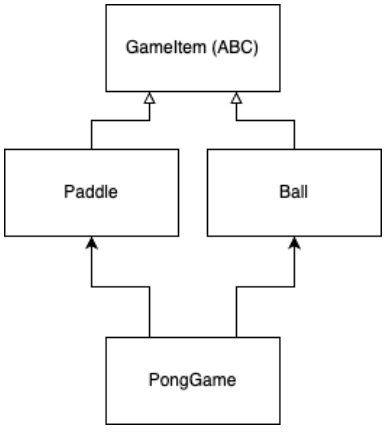
The choice of program for this assignment is the **Pong Game** implemented in the first assignment. It was implemented based on the traditional rules outlined in the assignment01 description.

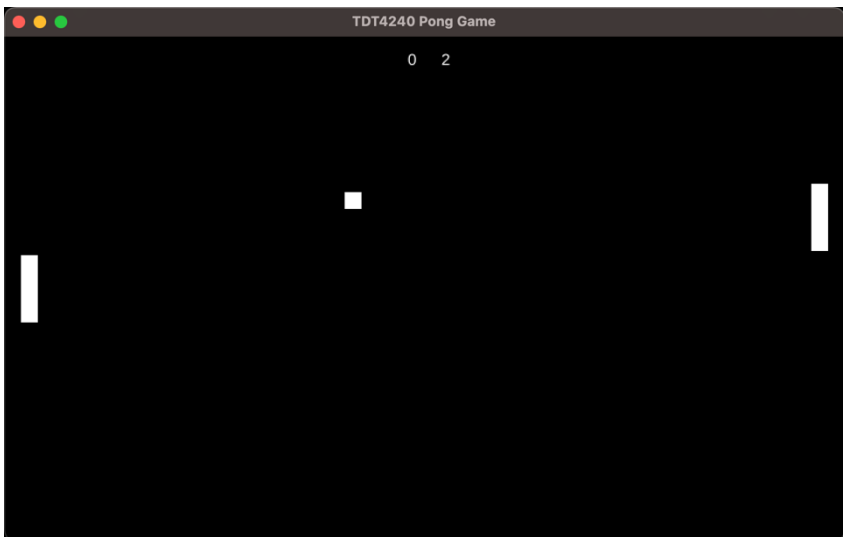
LibGDX was used and the program was tested both in **desktop** and **android** mode.

A simple structuring of classes is done. The main **PongGame Class** serves as the **ApplicationAdapter** for our program – handling everything from user input to sprite management and game flow.

GameItem implementations, such as **Ball** and **Paddle** exist so as to encapsulate behavior and state of these game items.

The main game logic is handled in the **PongGame** render class.



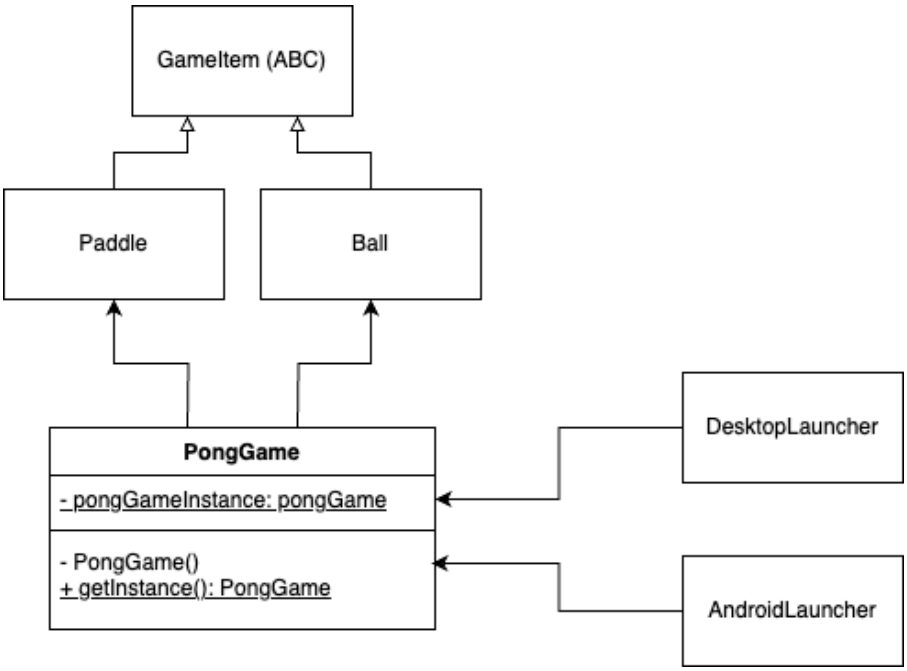


Part 2 – Singleton Pong Game

The implementation of the **Singleton** pattern is done on the **PongGame class.** This makes total sense, as we do not wish to have multiple active PongGame instances – rather states of the game (as we'll see later!).

Singleton's addition was typical – a private static **PongGame** instance variable was added so as to hold our single instance. The **PongGame** constructor was made private, and it is now called conditionally by the **getInstance**() public method, which returns an existing game instance, or creates a new one if none exists.

One major difference here is that the Launchers that hold the PongGame are now calling the **getInstance**() method instead of creating a new PongGame. A simple, non-exhaustive diagram will illustrate the additions.



Part 3 – Singleton Pong Game w/ State

The last pattern choice is the **State pattern** – a quite obvious choice for a game! This pattern was chosen so as to improve the management of **PongGame** states – specifically "Running" and "Finished". The "if" statements and multiple lines of code in the render method of the PongGame have been replaced by one line which calls for the state's render method – resulting in much cleaner and well-seperated code.

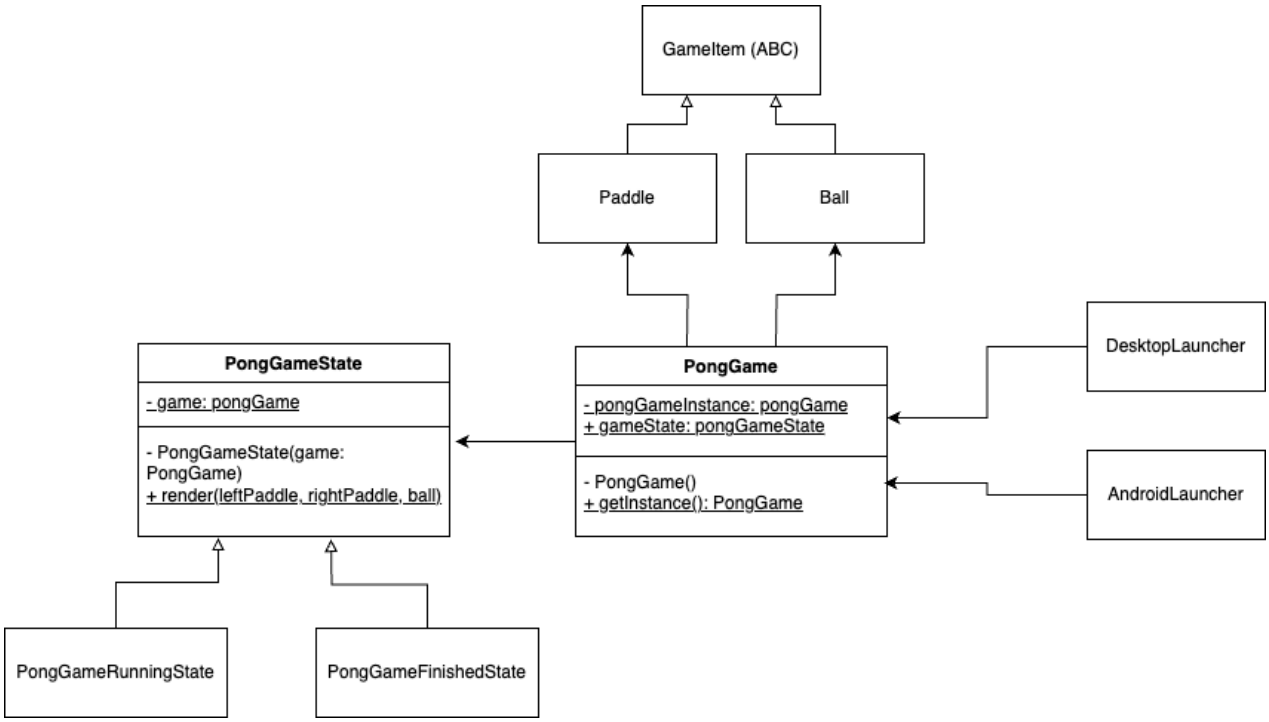
An **abstract PongGameState** class was added to serve as the general representation of our states – holding our single **PongGame** instance as well as an **abstract render**() method which takes all the to-be-rendered game objects in as arguments.

The game itself behaves and plays exactly the same across all implementations. However, the code is much cleaner and open to modifiability now that patterns were introduced.

The state implementations are:

- **PongGameRunningState**, which simply prints all of the objects as is and allows the user to move their paddle. As soon as the maximum score (21) is reached, it switches over to the **FinishedState**.
- **PongGameFinishedState**, which prints the final score, the winner as well as a prompt to restart the game (by switching back to the **RunningState**). The game assets are frozen as the game has concluded and there's no need for movement.

A simple, non-exhaustive diagram to improve understanding.



Part 4 – Theoretical questions

4a)

Architectural Patterns: Model View Controller, Entity Component System
Design Patterns: Observer, State, Template Method, Abstract Factory, Pipe and Filter, Model View Controller (? Disputed)

The main difference is that Architectural patterns are usually a general blueprint for a whole system of components, while Design patterns are implementation-oriented. Design patterns **are** concerned about component implementation, while architectural patterns are more concerned with component structure.

In simple terms, architecture patterns are concerned with the general structure of the system while design patterns are concerned with fine-grained implementations within that structure.

4b) Already answered in (3).

4c)

Using a **State** in a game seems almost like a natural choice. Games often have a lot of states, and switch between them quite often. By adding a **State** to the PongGame I was able to separate the execution from the finale. In a future addition, another State could act as a Pause, a Main Menu screen and so on. The foundations for multiple future state implementations are there with just a basic PongGameState **abstract** class which can be overloaded to our desire.

Part 5 – Comment

This assignment was carried out individually.