TDT4240 – Assignment 2

Step 1: Implementation of a program

I chose my Pong implementation from assignment 1.

Step 2: Implement the Singleton pattern

I created a GameScore class that stores and handles game score logic. By having a private constructor, a private static INSTANCE variable and a public getInstance() method, I ensured that the program can only have one instance of the GameScore class.

Step 3: Implementation of pattern

I chose the State pattern. More about the implementation details in Step 4.

Step 4: Theory

3.a)

Observer: Design pattern

State: Design pattern

Template method: Design pattern

Model View Controller: Architectural pattern

Abstract Factory: Design pattern

Pipe and filter: While Wikipedia and MSDN call it a design pattern, the book «Software architecture in practice» places MVC and Pipe-and-filter side by side.

Design Patterns, such as the observer pattern, are well known solutions to technical problems that we see often in software construction. Design patterns are tools at lower levels than architectural patterns. An architectural pattern, such as MVC, is more fundamental, about how you structure your software application. It's about global properties and mechanisms of a system.

«Architectural patterns are similar to software design patterns but have a broader scope»

- Wikipedia

3.b) I created an abstract GameState class. This class has a reference to the Game instance as well as an abstract Update and Draw function. Every class that extends GameState must implement Update and Draw, or else the program won't compile. Additionally, I implemented CountDownState and InGameState which extend GameState. All the draw and update-related code has been refactored from the game instance to CountDownState and InGameState. The game instance has a variable that always references the current state. In the update function of the game instance, simply gameState.update (with parameters) is called. The same concept goes for the draw function. This way, I could remove the conditional statements and draw and update-related code from the game class.

3.c) In my previous implementation of pong, I had conditional statements and draw and update code inside the game class. If there are many game states, then that way to do it obviously results in large conditionals and spaghetti code that has a negative impact on modifiability. By moving update and draw-related code from the game class to one separate class for each game state, one is able to keep the amount of code in the game class to a minimum. If the game should grow and get many states, this design pattern is good. It lets the game state simply call draw and update on the current state. However, for a simple game such as pong, this design pattern does result in more code. One might say that using this design pattern for such a simple game is over engineering. The two game states both need to access variables inside the game instance, and to do that while still using encapsulation, we need lots of getters and setters. Those are not needed if the update and draw code is written directly inside the game class.