Exercise 2 Theory

Task 4

.a)

The difference between an architecture pattern and a design pattern is mainly that a design pattern can be represented with a class diagram, and thus represents the structure of the application on the lowest level, class level. An architecture pattern, however, represents a collection of classes or modules and how the major parts of the system work together. Thus, architecture patterns are a tool for designing the system on a high-level, while design patterns are a tool to design class by class.

This makes the observer pattern, state pattern, template method pattern, model-view controller, abstract factory pattern and pipe and filter are design patterns as they can be explained with a class diagram. Only the Entity-Component system is an architectural pattern as it handles components on a larger scale than the others.

.b)

I maily chose to implement the model-view controller pattern. This pattern is about separating the state and data for something into a model class, having the representation of such state in a view class, and having the control and communication between the model and view class through a controller class. The controller class is the class that accepts the inputs. I had already structured my game into sprites and game states, and thus I wanted to split the sprite up into a model, view and a controller class. In this case I thus wanted the controller class to be the interface between the sprite and the game state. There are three models: HelicopterModel, BirdModel, BallModel, and these are all variations of the base model GameSpriteModel. There are three corresponding views: HelicopterView, BirdView, BallView, and these are all variations of the base model GameSpriteView. Then the view and model are combined in the corresponsing controllers: HelicopterController, BirdController, BallController, which are all variations of the base controller GameSpriteController.

The model classes handle the states of the applications, that is the position, velocity and acceleration of the sprite. It also handles what to do with those parameters when different types of collision occurs. The view class handles the representation of the sprite to the screen. This is done through the methods render() and updateAnimationFrame().
The controller takes in the corresponding model and view and creates an interface to the game state using the methods in the model and view. The controller also handles input and change and calls the corresponding methods in the view and model classes.

I also follow the state pattern by having different game states: a menu state and a play state. The different states are handled with the GameStateManager. Through different occurrences or inputs in the different states, the state changes. For example will a click on the play button in the menu state transfer to the play state, and if you (the helicopter) hits the floor in the play state, you lose and are transferred to the menu state.

I am also using the template method by calling different methods in update function of the controller. By overriding the methods used in the update method of the superclass, you can add functionality to these steps in the update method. This makes it easier to handle for example input for each sprite since you do not have to rewrite the whole update function, but only the handleInput() method.

.c)

Advantages of using **model-view controller**:

- Separating each sprite into a model, view and controller makes it possible to develop on each separate part at the same time with different developers.
- This separation makes it easier to work with the system as the project grows bigger. In my project, the model, view and controller files for each sprite only has a few lines of unique code. As the project grows larger, having independent files for each will make the project easier to understand how the code works.

Disadvantages of **model-view controller:**

- For such a small game as this one, separating the code like this adds more “boilerplate” code and is probably not worth it.
- Adds a bit of latency for user interactions as some function calls are passed down from the controller to the model or view. This creates some overhead.

Advantages of using the **state pattern:**

- Makes it easier to add new menus/pages, game modes, or difficulties.

Disadvantages of using the state **pattern:**

- Adds overhead as another data structure must handle what state the application is currently in and pass the call down to each state.
- Adds a lot of “boilerplate code” relative to the rest of system when the project is small.

Advantages using the **template pattern:**

- Makes it easy to implement a tiny change in functionality for a class while still following the basic overarching algorithm related to that class.

Disadvantages using the **template pattern:**

- Depending on the implementation, you may lose some flexibility in what tiny changes you are able to make.