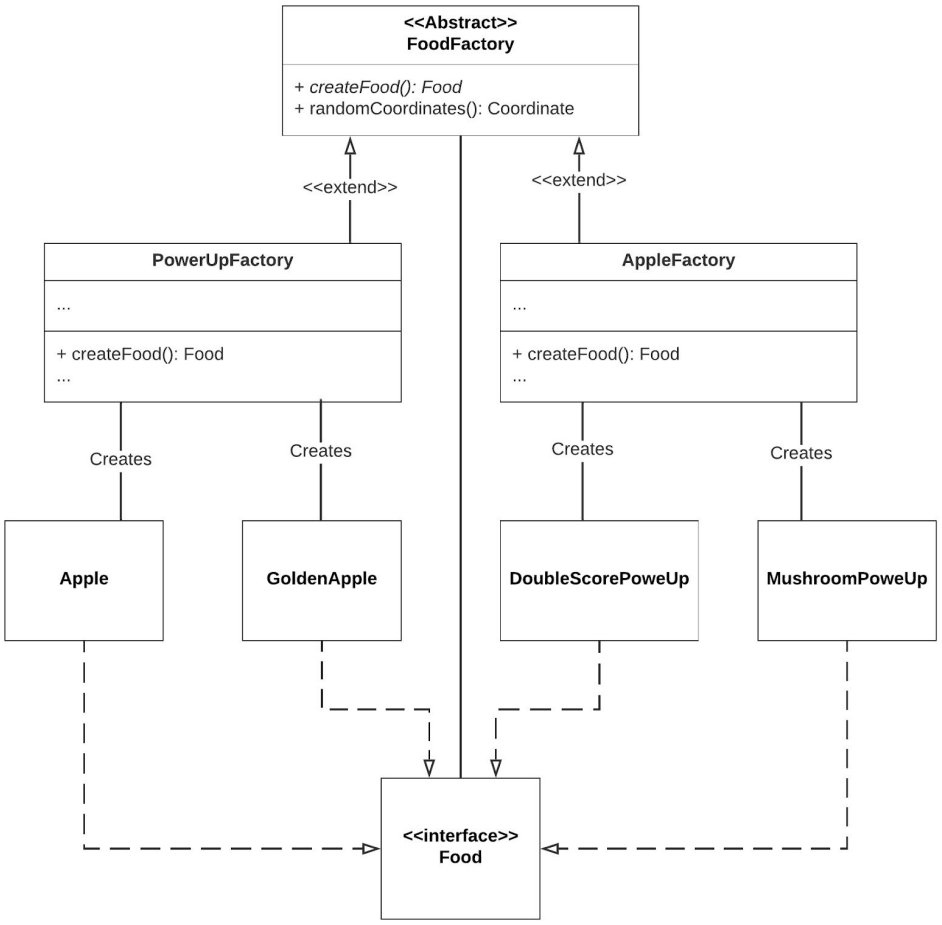
Assignment 3

Group 10

1. Design Patterns

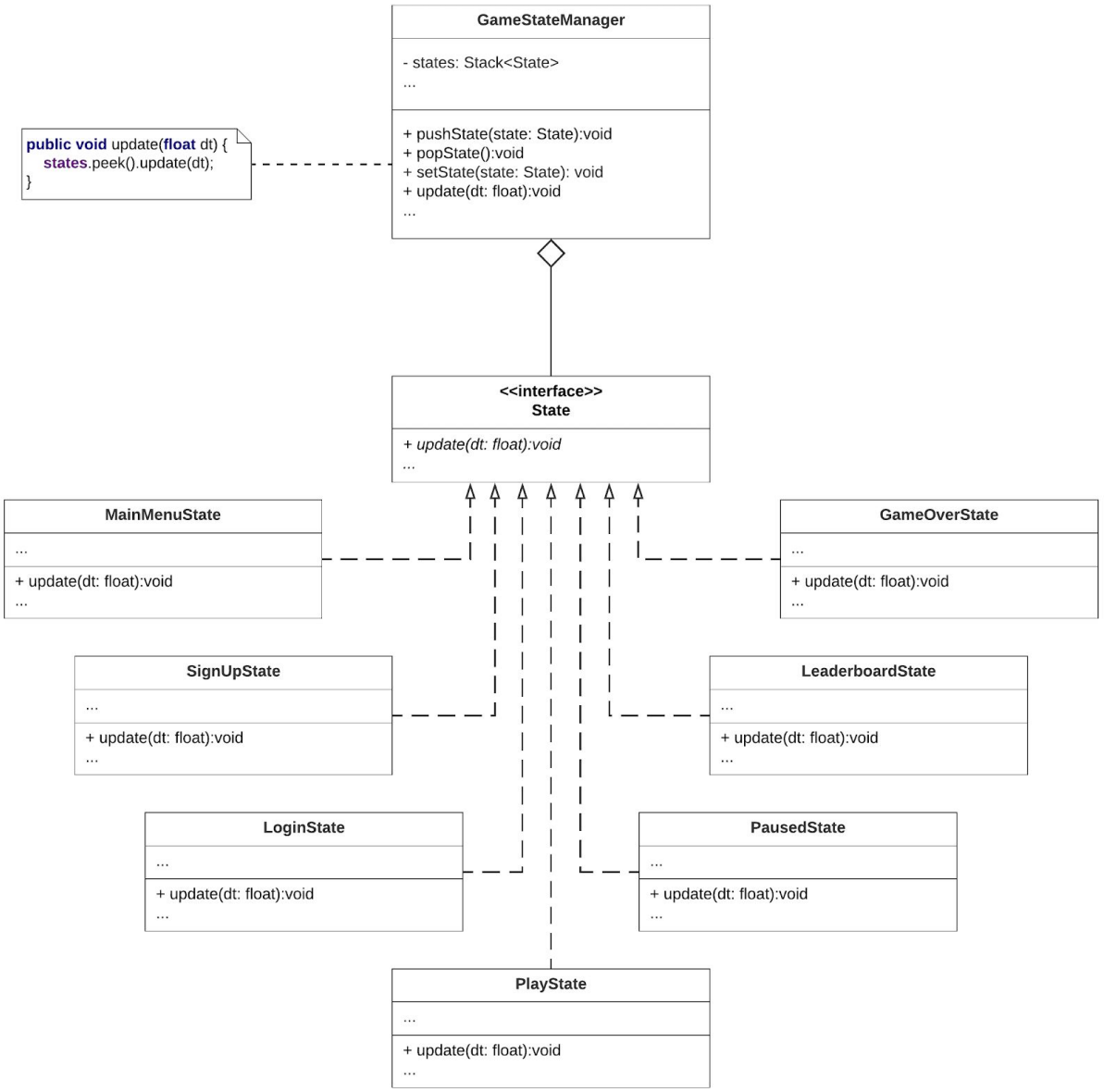
1. Factory Method



We chose to apply the Factory Method for the creation of edible elements in the game. By applying this design patterns we created two factories: PowerUpFacory and AppleFactory. From the beginning of the game until 100 points are reached the only factory in use is the Apple factory which places Apple and GoldenApple objects on the game map. After 100 points are reached, the PowerUpFactory takes over and apart from continuing to generate apples it also starts generating MushroomPowerUp and DoubleScorePowerUp objects at

predefined probabilities. These power-ups are different from apples because they change the appearance and behaviour of the game for 10 seconds and do not grow the snake. The Apple, GoldenApple, MushroomPowerUp and DoubleScorePowerUp implement the Food interface. This design pattern was useful for us in separating the creation of different objects at different stages of the game. We found it suitable for this case because new food elements in our game have to be created randomly at runtime and we can't anticipate the class of objects we must create.

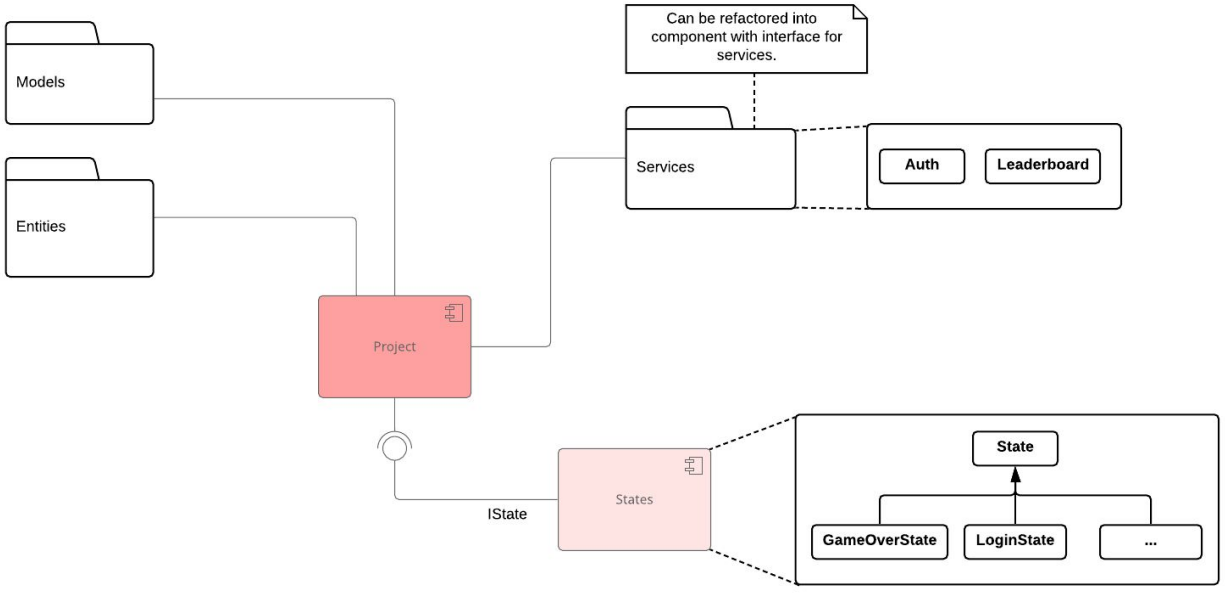
2. State Pattern



We chose to apply the State patterns for the plethora of different states of our game. GameStateManager is the class in which the states of the game are set, changed and initialized. The GameStateManager calls the update method of the current state. The update method is responsible for handling the input and/or physics within the current state. At the beginning of the game GameStateManager is in the LoginState and from then on its state changes depending on what actions the user performs. By applying this design pattern we could partition the behaviour, GUI's and possible user actions of specific states into separate classes. This design pattern was also suitable for us because the GameStateManager has to change state at runtime.

2. Software Architecture

Although the project shares some practices of MVC architecture, it does not use it in the full extent. Instead, the project is divided into multiple modules that take care of different parts of the game based on its overall subject. We do not run the server, thus it is neither a server-client architecture.



For example, the module called 'services' is dedicated to all external communication, mainly database. These so called services, work with data, rather than with the game logics. It makes it very convenient to apply changes using the similar way. For example, when we need to add a new score to the leaderboard, we create a new corresponding service and just create a new entry using the method. There is no need for handling database connections or schema structures in the game logic classes. So, the database is independent of game logics.

The module called 'states' is dedicated to mainly UI part of the project. It handles all of the scene management, transitions and rendering; all buttons, fields, etc are configured in there. We chose to implement this module, because it is convenient to have all scenes and UI elements at one place. Although it may contain some of the game logics, we tried to separate

these parts as much as possible to convey the best practices of MVC. All states are based on a corresponding State class that reflects the base. It is an abstract class that defines all necessary methods that are needed in every single states. This makes all the states coherent and flexible. Thus, the code becomes more maintainable.

We have a module for models. Those are data structures that are used in the project. They usually do not contain any complex logics, but rather they are just containers for data. These structures in turn can be used by other classes in order to operate with data more efficiently by reducing the redundant code in the game logic classes.

We have a module for entities where all interactable objects on the map are implemented, such as snake itself or eatable objects like apple. This module is crucial for the game. Those objects are extendable and must be maintainable. Thus, it is wise to separate them into separate module. In there, there is also a division between snake objects and others. Snake related objects are very crucial by themselves. The functionality can be further extended on later stages (e.g., skin modifications).

The chosen project architecture keeps the crucial parts of the application separated and thus more maintainable. In this way, it is easier to access the wanted functionality and extend it. Needless to say that this way has its own cons. Although it works well for the current project size, it can hurt maintainability in case of a larger codebase, or if more game elements with complex logic appears. Nevertheless, since we do not expect it to grow significantly, the solution works out for our case.

We think that this approach worked for us better than pure MVC, because the main components have been divided according to the responsibilities of each team member and the main part of focus of our project. We realize that this approach could not be applicable in other circumstances, in a different context or with different goals.