Assignment 1

Singleton

1. Write a natural language description of why and how the pattern is implemented in your code.

We implemented the singleton pattern in the `Application`, `GameController`, `GameViewer` and `FoodManager`. We assume there is only one game running at the time. So there is only one application and one controller. Moreover there is only one window so one viewer.

Our objective is to access all three objects from anywhere in the code (in any object) while guaranteeing it is always the same instance. `Application`, `GameController`, `GameViewer` don't use multi-threading so we don't need to `synchronise` constructors. The `FoodManager` on the other hand uses synchronized methods because it works in another thread.

The `FoodManager` was created in order to give less responsibilities to the `GameController`. And since there should be only one `FoodManager`, we made it a Singleton. Moreover it helps with the concurrency problems! Indeed, since this `FoodManager` is a Runnable, its run method can be executed in another thread to generate (at random times) new bad apples. By moving everything food related to the food manager we were able to use `synchronized` methods in it without causing unexpected locks / bugs in the `GameController`.

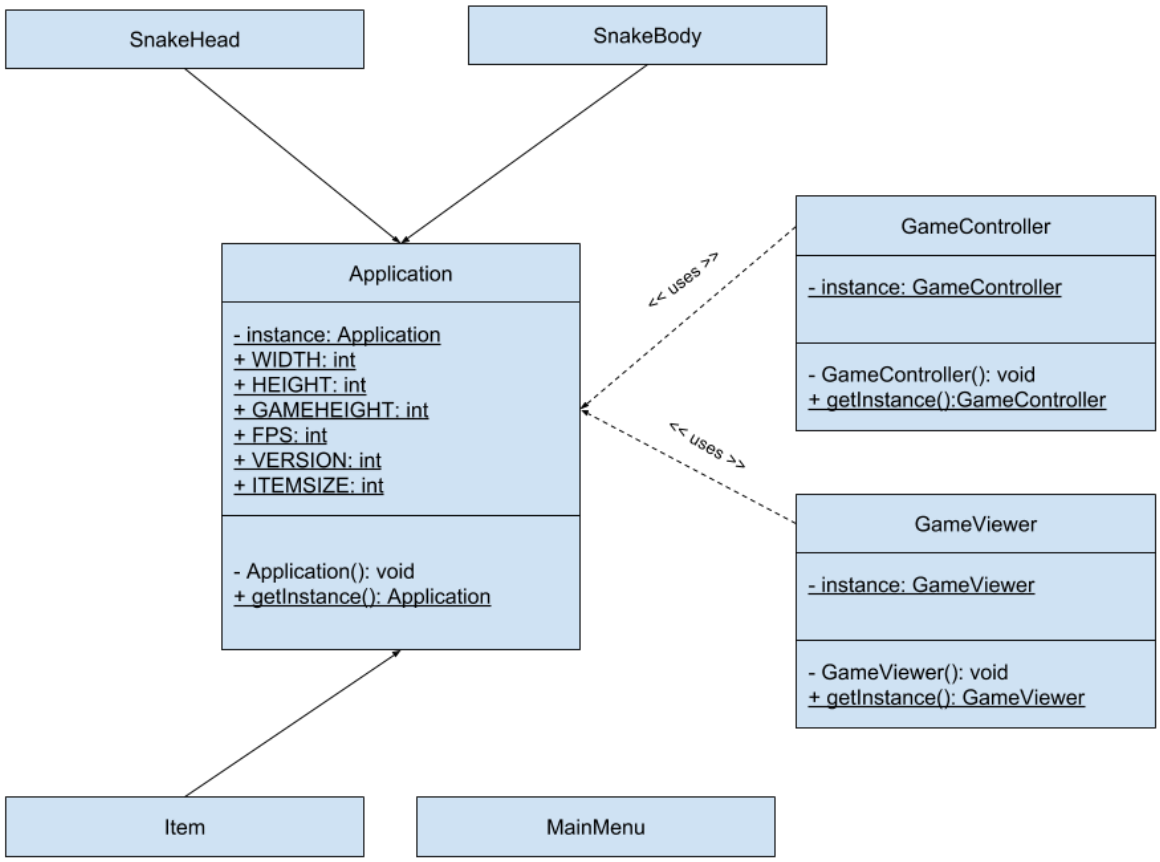
As Application is the starter class for the main, we create the Application singleton in main and simplify it's `getInstance` to simply return the value.

All other singletons look like :

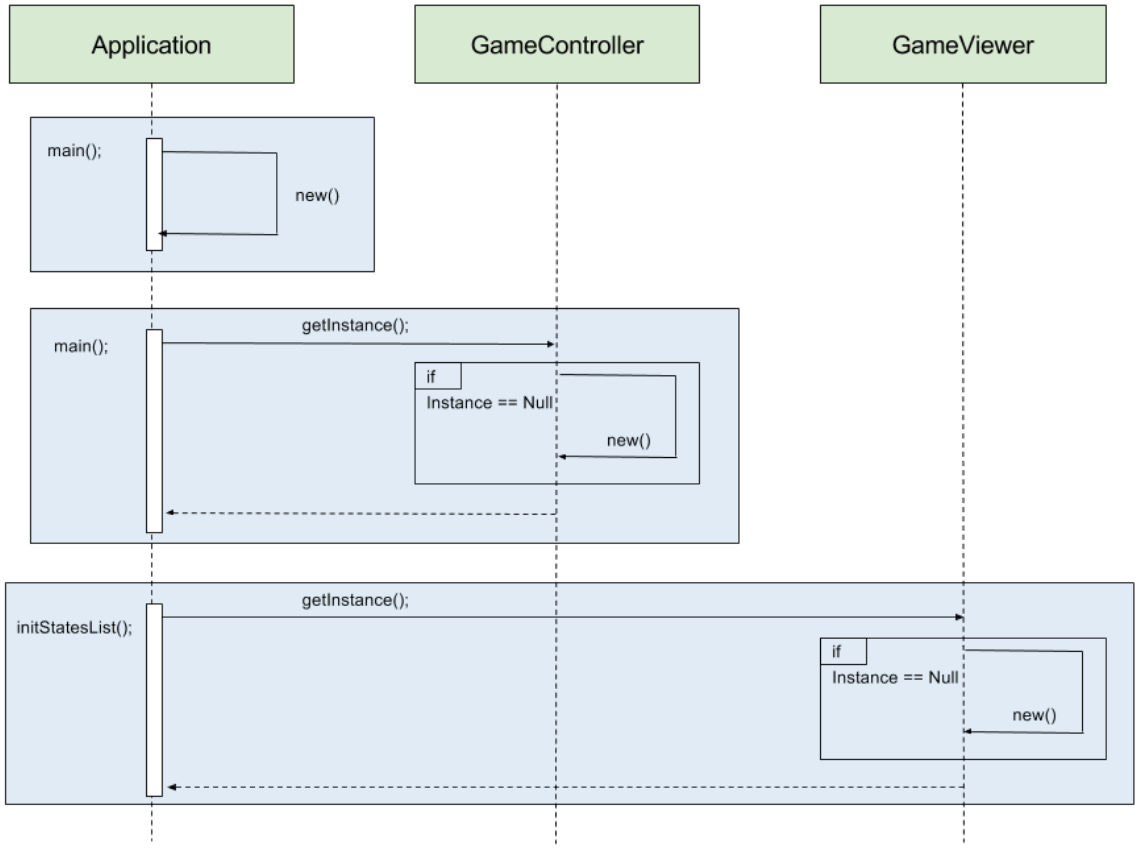
```
private static Class instance;
public static Class getInstance() {
    if(instance == null) {
        instance = new Game();
    }
    return instance;
}
```

2. Make a class diagram of how the pattern is structured statically in your code.

All `public static` variables are final for security reasons.



3. Make a sequence diagram of how the pattern works dynamically in your code.

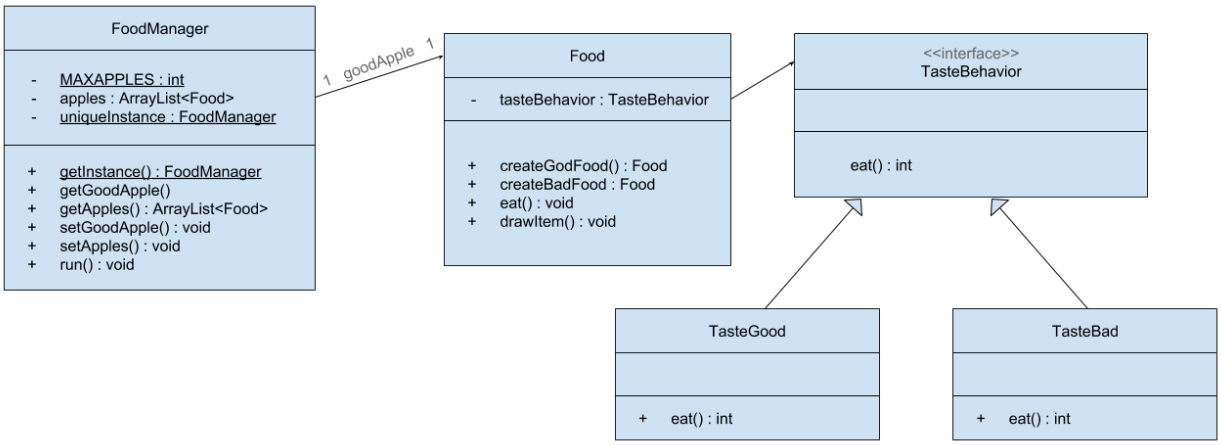


Behaviour

1. Write a natural language description of why and how the pattern is implemented in your code.

We implemented behaviors for the food that the snake is eating. Thanks to our implementation, we can combine multiple behaviors without inheritance. Furthermore, we can change the behaviour during runtime. The pattern is implemented using a `TasteBehavior` interface. `TasteBad` and `TasteGood` implement the interface and define the effect of the food. It also allows to easily integrate future behaviors following the open-closed principle.

2. Make a class diagram of how the pattern is structured statically in your code.



3. Make a sequence diagram of how the pattern works dynamically in your code.