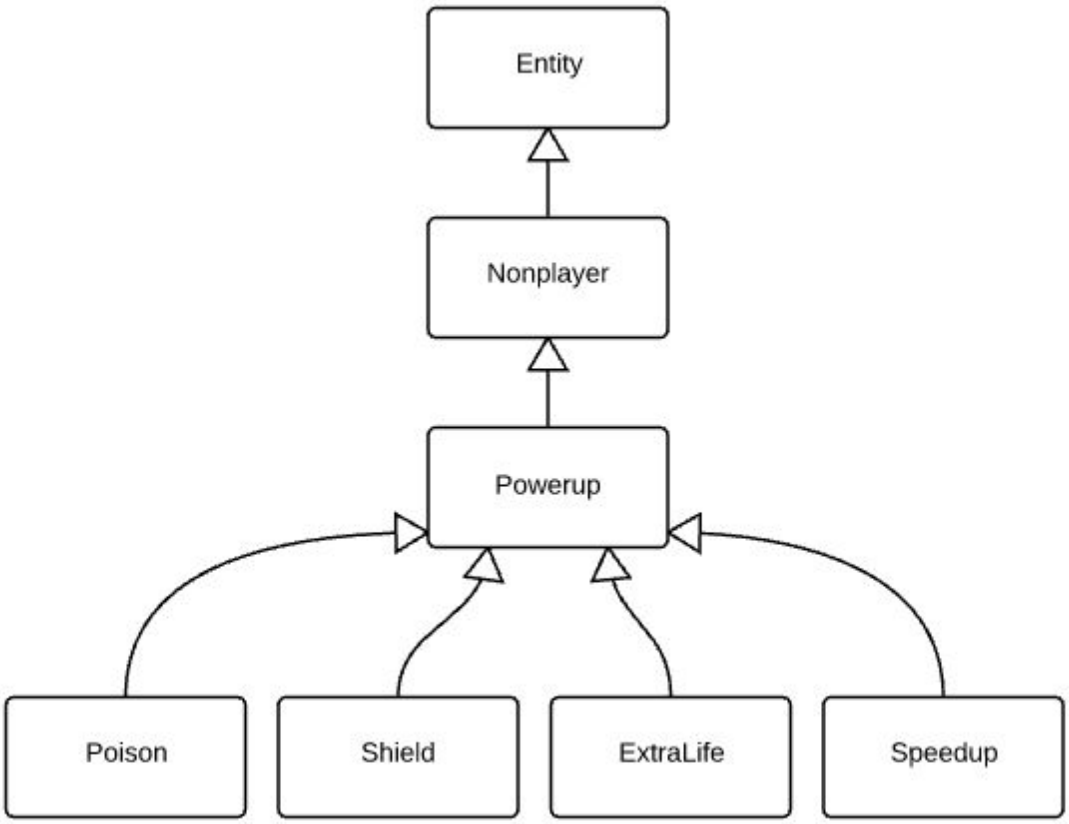
Exercise 1

UML for the power-up feature.



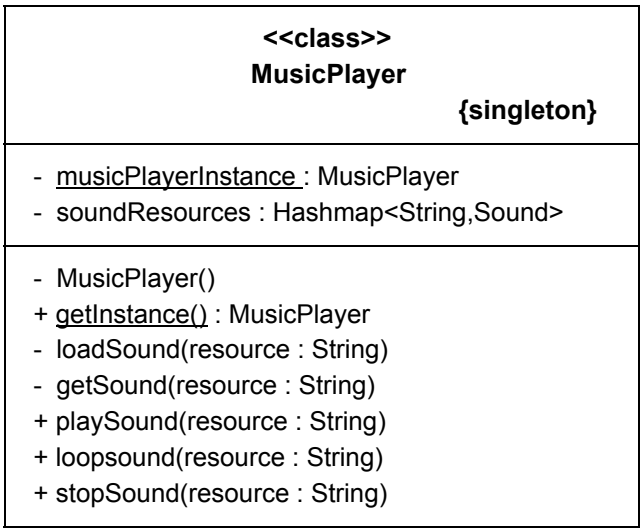
Exercise 2:

Design pattern: Singleton

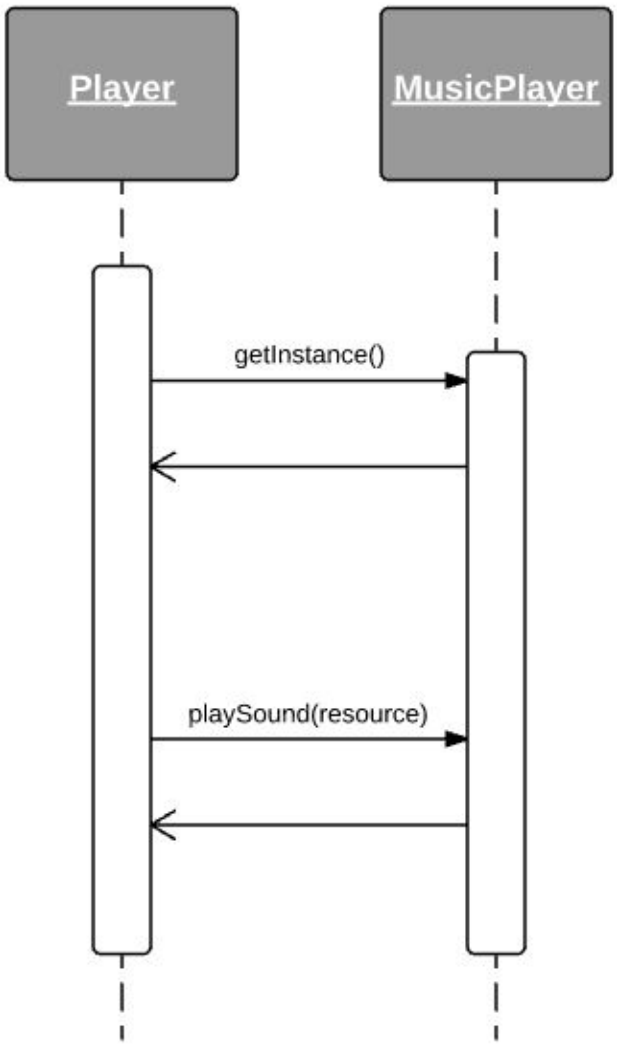
1. Write a natural language description of why and how the pattern is implemented in your code

The Singleton pattern is implemented in the *MusicPlayer* class to ensure there is only one class responsible for interacting with sounds. Having only one instance of this class is necessary, because it gives the ability to preload sounds before playing them. The class also provides a global access point to start, loop and stop background or game music, which could happen at any point in the code. The singleton pattern is implemented by setting the class constructor to private and using a static public method *getInstance* to create and store an instance. When this method is called again, the stored instance is simply returned.

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



Design pattern: State

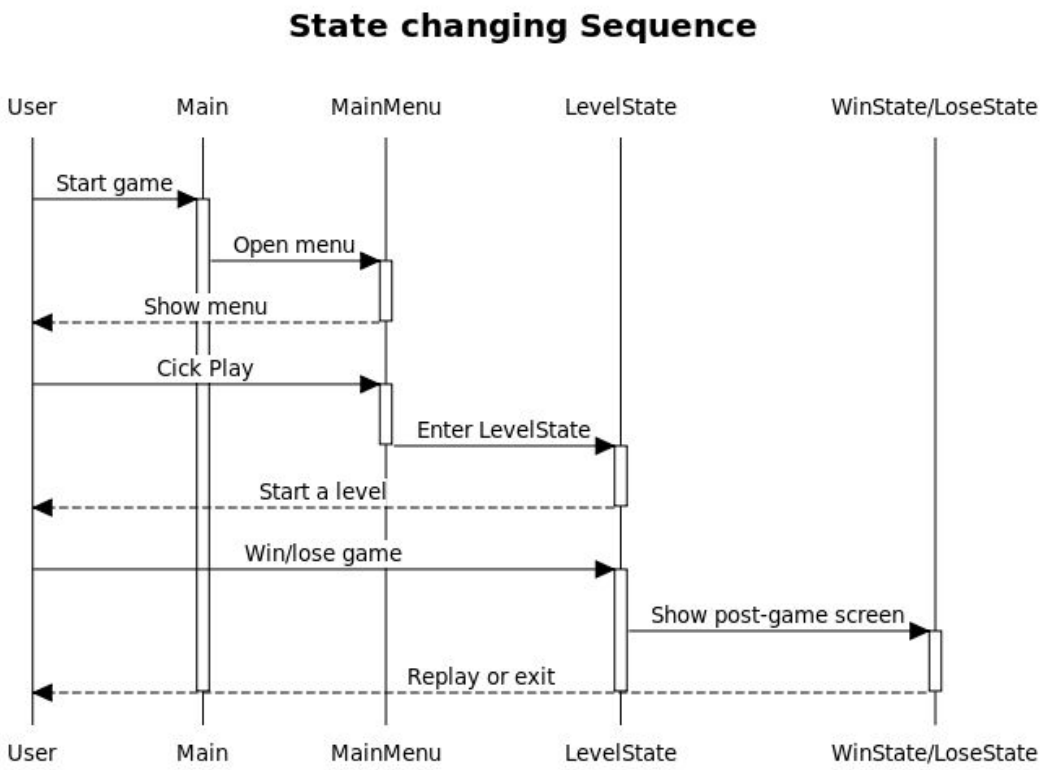
1. Write a natural language description of why and how the pattern is implemented in your code

Any game which goes beyond the very basics will consist of multiple screens, each with different information displayed. For this implementation of Fishy, the handling of these different screen is done with the use of various ‘States,’ following the State design pattern. The used framework, slick2D, facilitates this, which is the main reason we used this method. The basic idea is that every kind of screen is contained in a certain game state. Our implementation currently contains 4 ‘States’: the *MenuState*, which display the main menu; the *WinState*, which is the screen one faces after winning the game; the *LoseState*, which is the screen one faces after losing; and finally, the *LevelState*, which is the screen in which the player actually plays the game.

Our *Main* class is the actual game window, and the class that can have a number of internal states. Whenever the *enterState()* method is called, the current state is changed by triggering the state’s *enter()* method and the end user is presented with the desired screen. All the implemented states extend *BasicGameState*, which is part of the slick2D library. A large part of the logic is baked in the library we use which made it very convenient to implement.

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



Exercise 3:

1. According to the research study, there are 7 success factors that can be used to recognize Good Practices. The four factors that were tested to be strongly significant to Good Practices are:
    - Steady Heartbeat
    - Fixed, experienced team
    - Usage of agile (SCRUM)
    - Release-based development

    There were also 9 factors that were tested to be significant to influencing project failure, the 4 most strongly significant are:
    - Rules & Regulations driven
    - Dependencies with other systems
    - Technology driven
    - Once-only project.

2. Visual Basic being in the good practice group is a not so interesting finding of the study because firstly there were only 6 projects in the measurement repository (which existed of 352 projects) which used the programming language Visual Basic and of these 6 projects, 5 scored as Good Practice and 1 scored as Cost over Time. Because there are only 6 projects who use this programming language, and Visual Basic project environments on average are less complex than others, we can conclude that Visual Basic being in the Good Practice group is a not so interesting finding of the research study.
3. 3 other factors that could have been studied in the paper are:
    - The Bad Practice factors that occur when a project is prematurely stopped or failed. The only projects that are studied were finished projects, but a lot of projects fail or are prematurely stopped and thus the study of these projects is relevant for the bad practice factors.
    - The difference between programming languages and the relation to bad/good practices.
    -
4. Once-only project: When starting up a once-only project, a lot of things have to be done for the first time, and for that one project only. This leads to a high probability of ending in Bad Practice.
    Many team changes, inexperienced team: When the team changes continuously and the team members are inexperienced, a lot of things are done for the first time etc. These members are not experienced in working in these project surroundings and thus the chance of failure is a lot bigger.
    Dependencies with other systems: When the built systems are dependent on other systems a lot of things can go wrong. As soon as something is changed in the system the project is depending on, a lot of things need to be changed etc. Thus resulting in a lot of extra work and time needed.