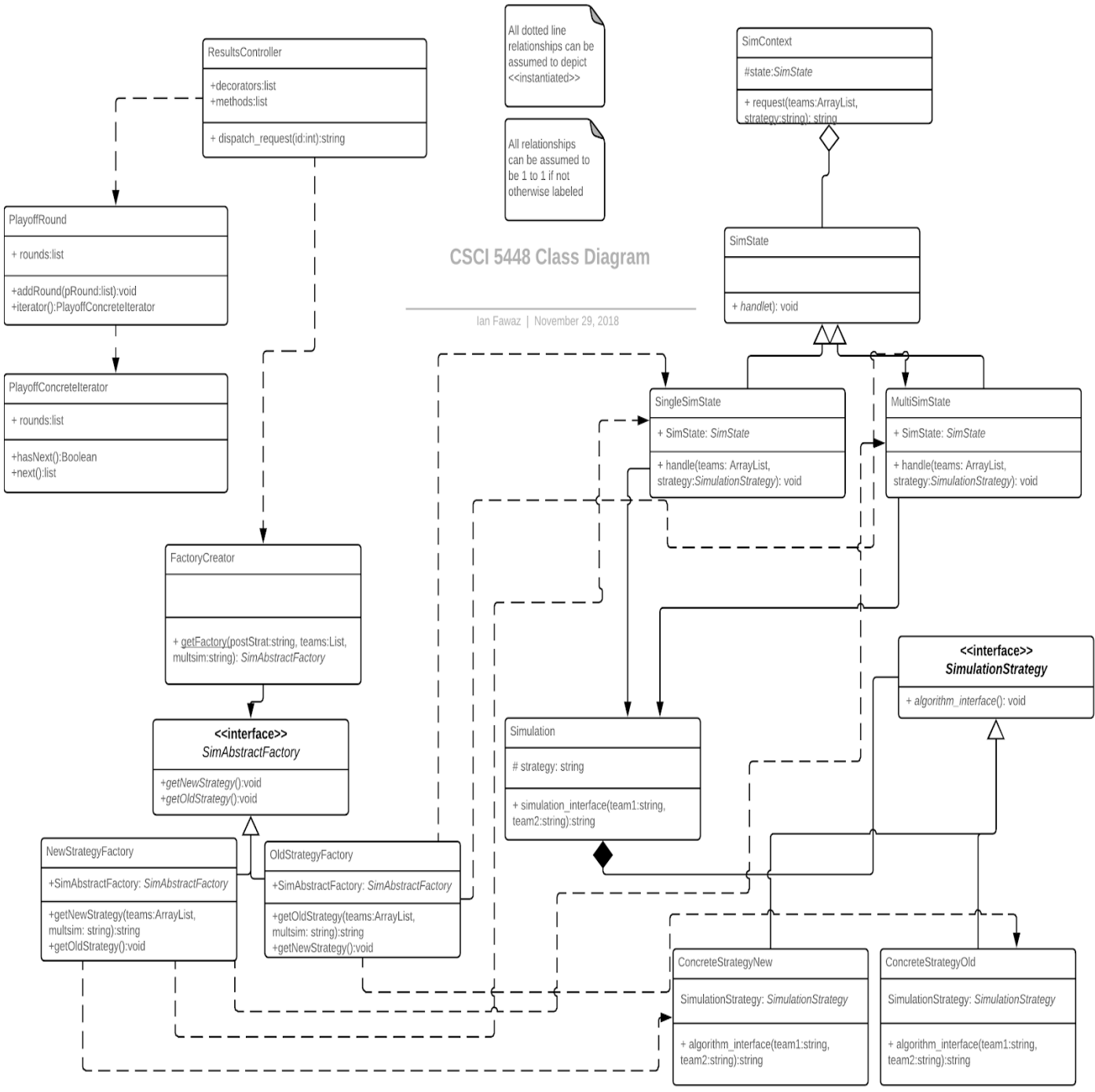
1. Ian Fawaz
2. A Flask app where users can fill out an NBA playoff bracket with any Western conference and Eastern conference teams from last year, and then simulate how the resulting playoff series' would play out. They can pick different strategies that would've helped all teams selected by the user in the playoff bracket fare better in the older NBA, or in the modern NBA.
3.

Req ID	Requirement
UR-02	Fans can set up playoff bracket by selecting team for a seed
UR-03	Fans can simulate one playoff series at a time
UR-04	Fans can simulate one round at a time
UR-05	Fans can simulate the entire playoffs at once
UR-06	Fans can choose two different strategies that affect how the bracket is simulated
UR-07	Fans can rerun simulation
UR-08	Fans can view aggregate of certain amount of simulation
UR-09	Fans can undo and redo the selection of different teams in the bracket
UR-10	Fans can save their bracket

4.

Req ID	Requirement
UR-01	Fans can view stats of any team chosen

5.

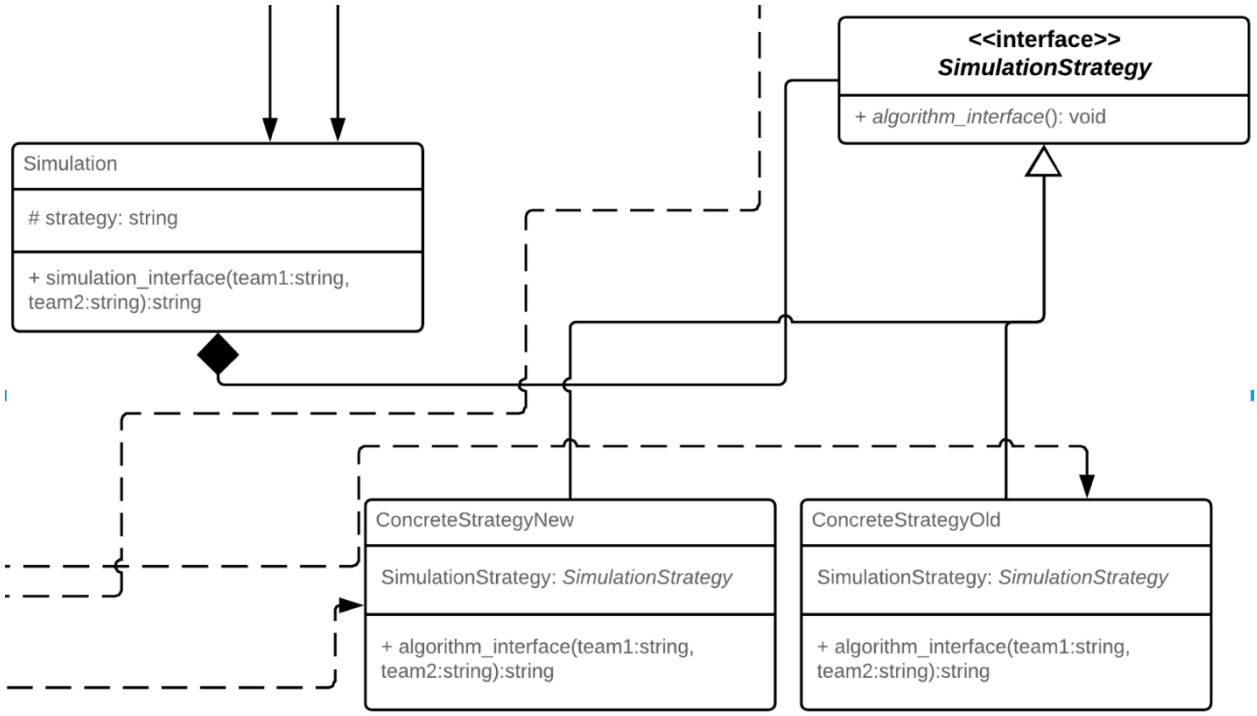


The class diagram is significantly changed from my initial class diagram. When I made the initial one, I did not have a well-rounded idea of how I would go about implementing the project. Once I started to implement the design patterns, the connection between all the different components of the project started to make more sense.

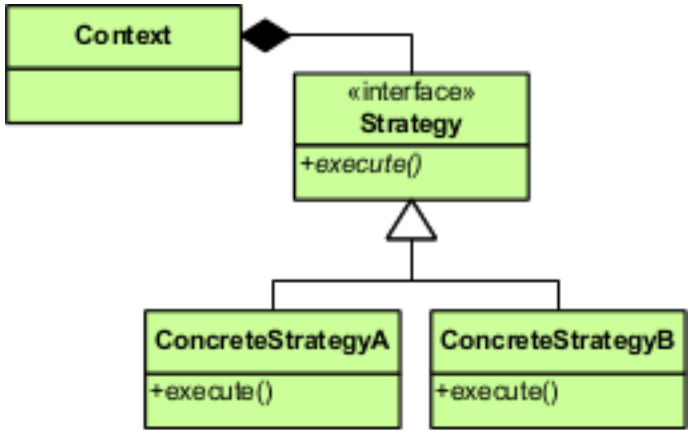
6.

Strategy

My implementation:



Wikipedia class diagram:

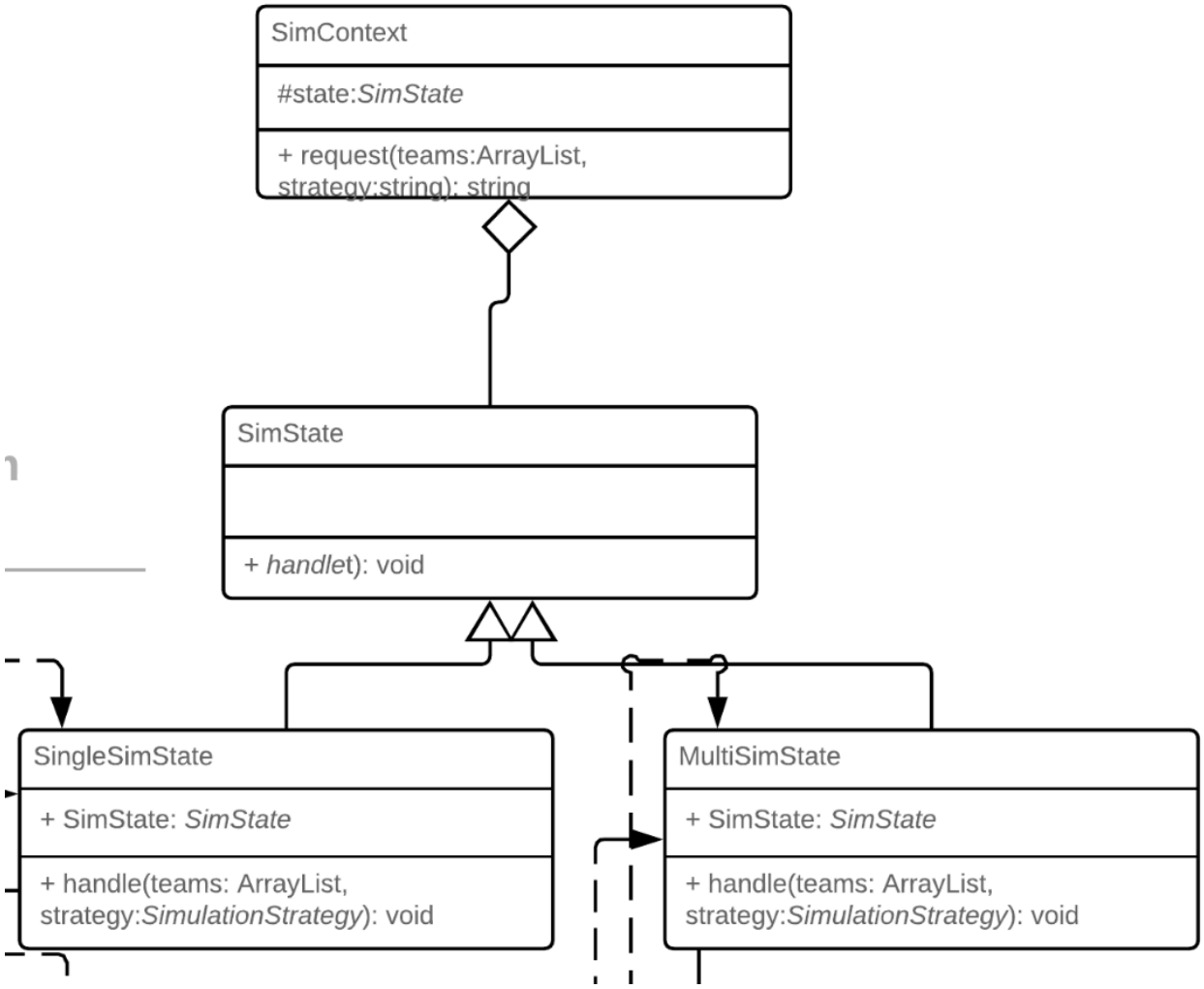


I selected this design pattern because choosing between the 'old' and 'new' simulation strategies was a crucial part of my project, and I needed a design pattern that would help this process. My 'Simulation' class represents the 'Context' class of the Wikipedia class diagram, which uses the abstract SimulationStrategy interface, which contains an abstract algorithm_interface method. This is inherited by the two concrete strategies: ConcreteStrategyNew and ConcreteStrategyOld. These concrete strategies carry out the actual

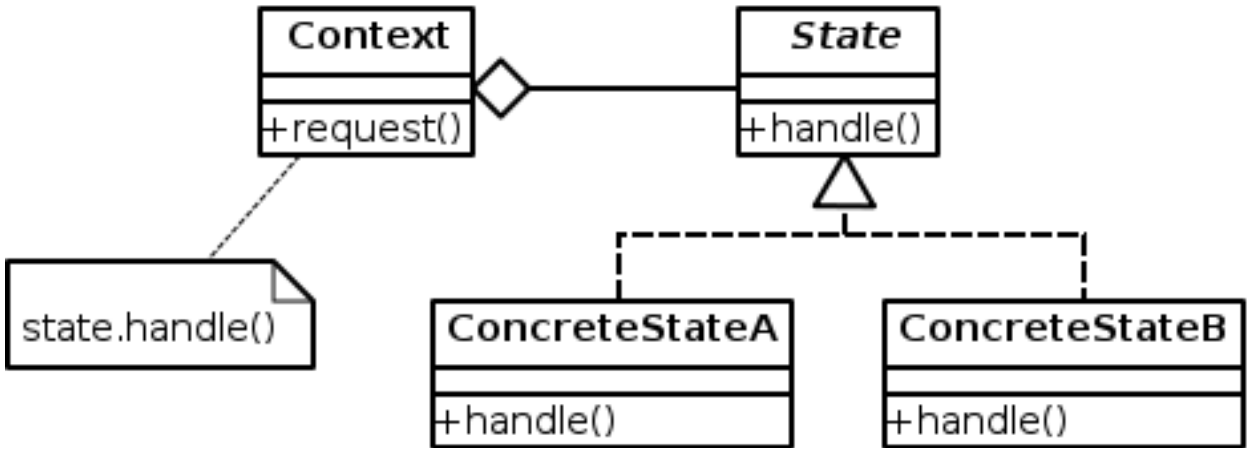
implementation of the two simulation strategies, where the new strategy focuses on using defensive rating to predict simulations and the old strategy uses True Shooting Percentage to predict simulations.

State

My implementation:



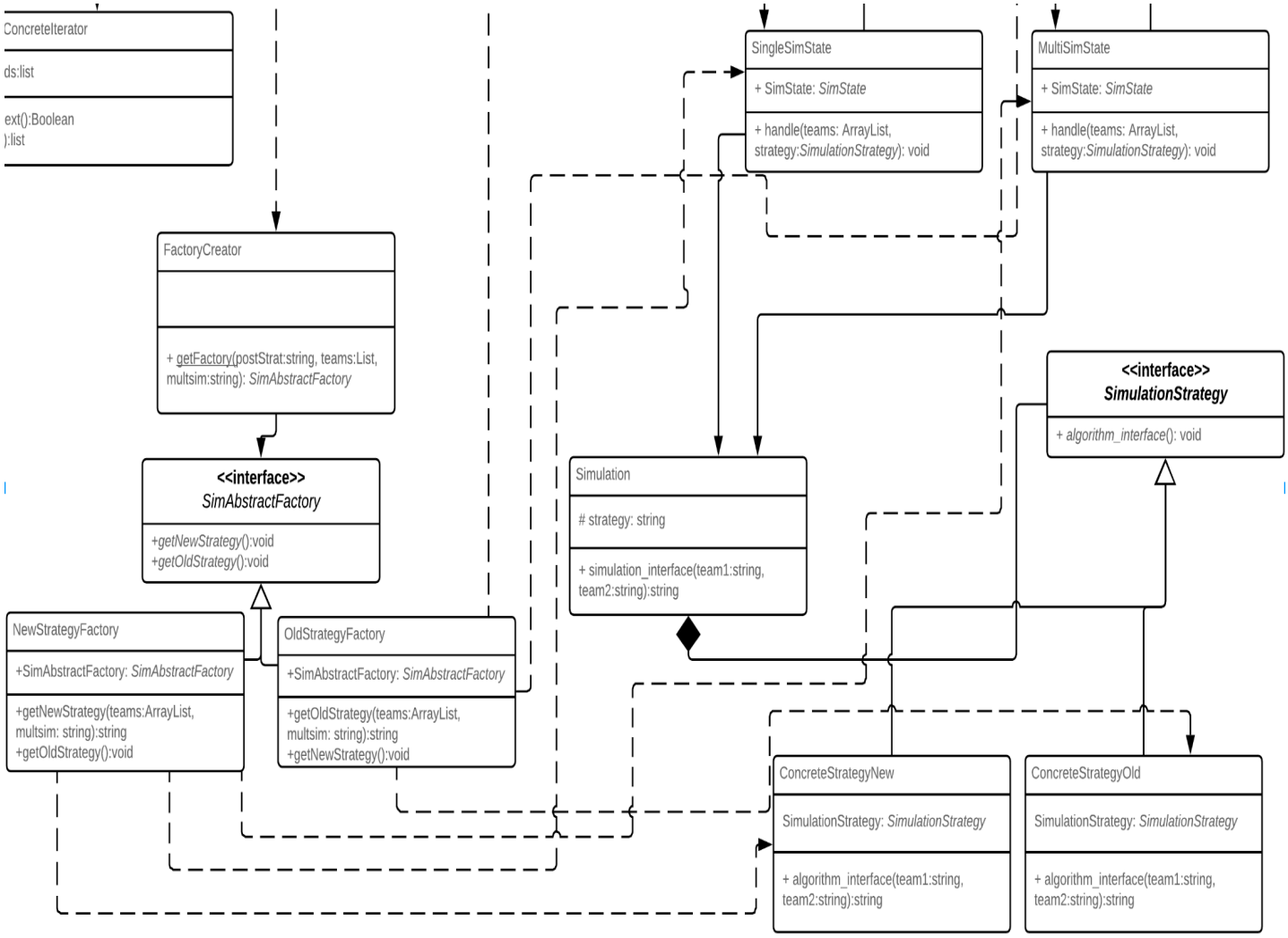
Wikipedia class diagram:



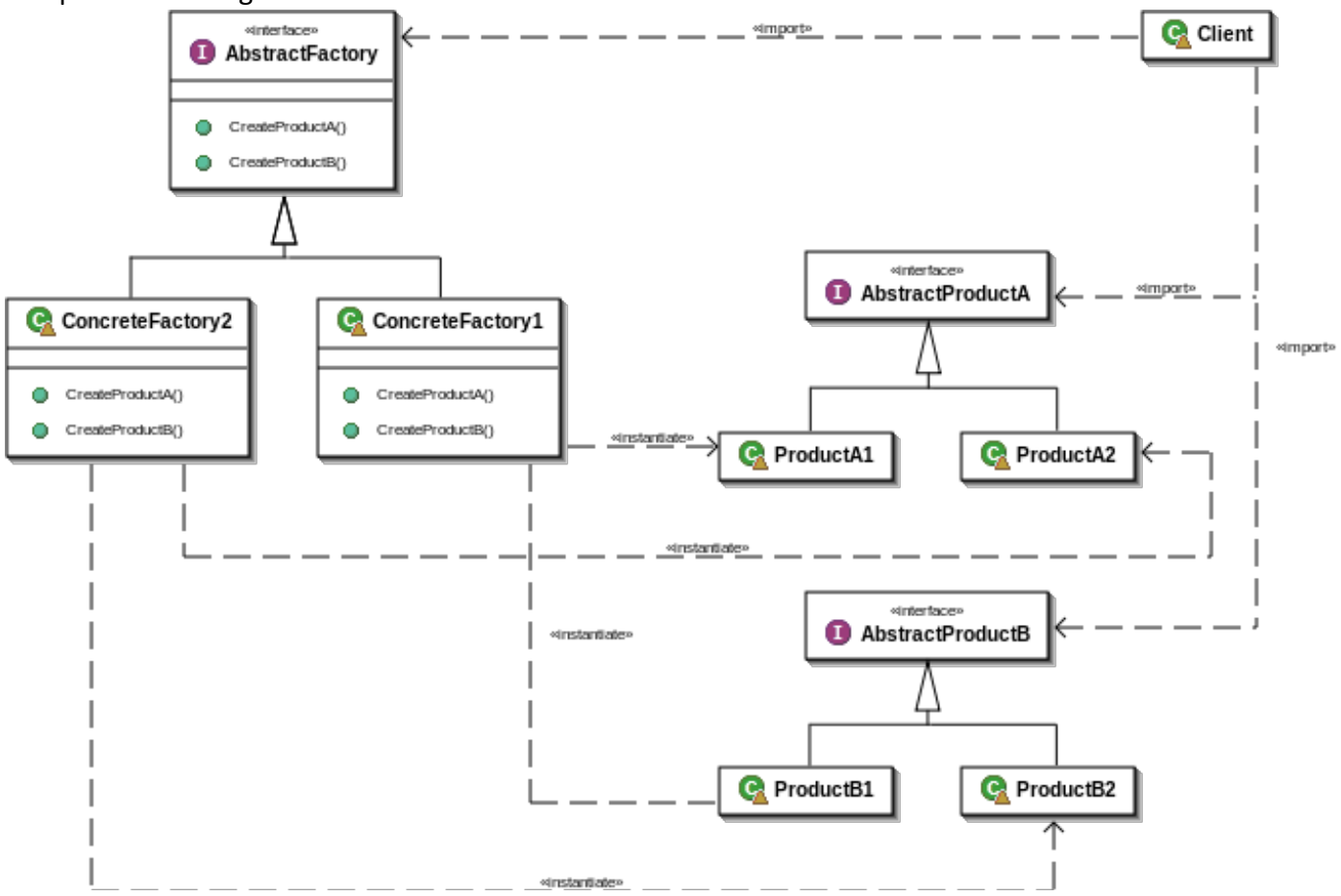
I selected this design pattern because I needed a way to indicate that there should be different states for multi simulation runs and single simulation runs. My SimContext class uses the SimState class, which provides an abstract handle() method. This abstract handle() method is defined in SingleSimState and MultiSimState, where handle() will return the winner of a series in a single simulation, or in a fifty simulation run where the most common winner is returned.

Abstract Factory

My implementation:



Wikipedia class diagram:



I decided to use Abstract Factory because I wanted to have an interface to run all of my related objects together without an extensive conditional with great amounts of class instantiation in my controllers. I have an extra class FactoryCreator (which uses the SimAbstractFactory class) that is used in the controllers to run the simulations with all the possible conditions and object relations met and satisfied in one interface. The concrete NewStrategyFactory and OldStrategyFactory can take in arguments like the type of strategy, whether it's a multi-simulation or single simulation, and the team names.

7.

The process of analysis and design is less intimidating once you start to actually develop the system yourself, and visualizing diagrams start to make more sense as well. I've also learned about the many different ways to clean up or make the process of design more efficient, as the

numerous patterns and principles learned in this course helped me realize that there is a genuine importance in the readability and adaptability of the code.