Willie Chew,

OOAD Project 6: ML-Tanks

Third Party Code vs Original Code

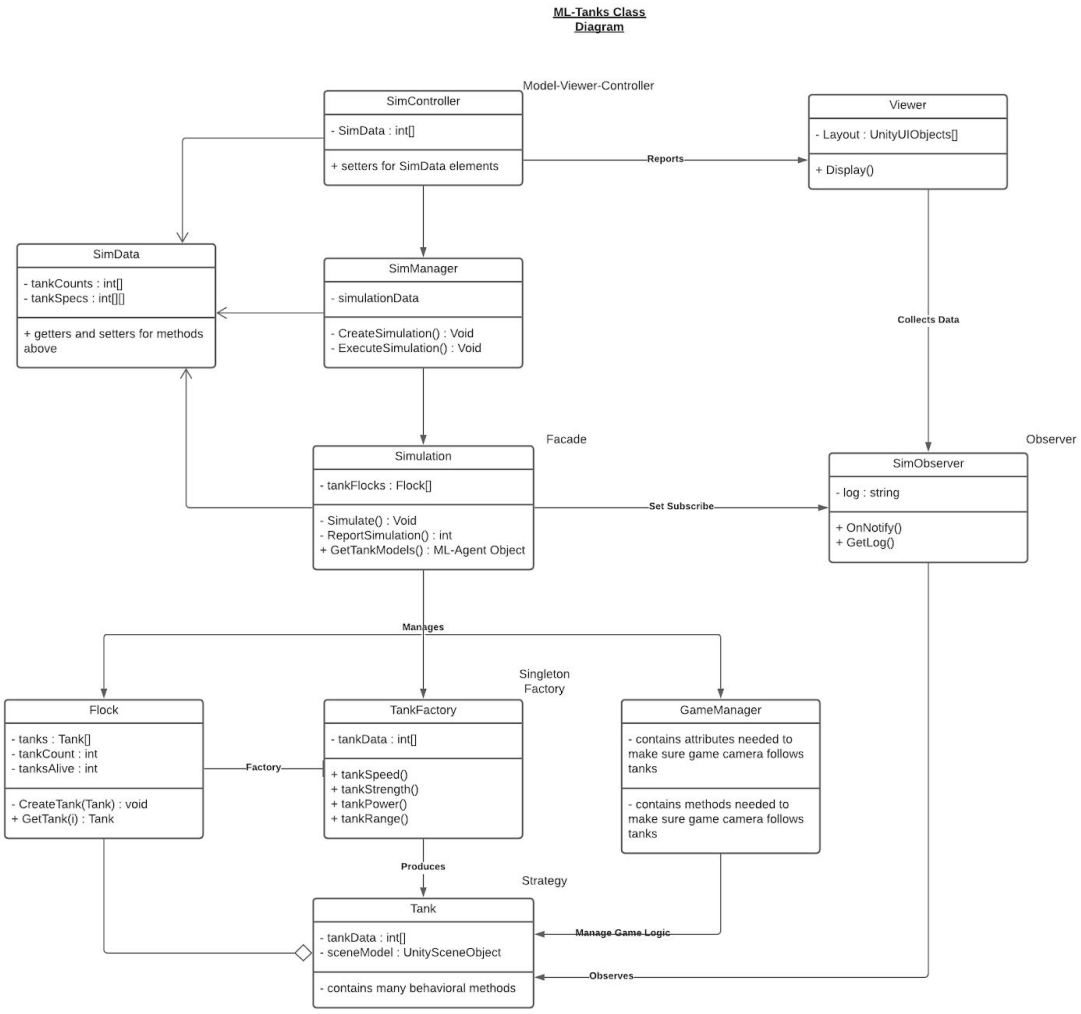
Our project is a modification of existing the following unity tutorial projects: Unity Tanks! tutorial, Tanks! pluggable ai tutorial. Most of the files in the project came from the tutorials. Code that our team added and modified are marked by the following wrappers: <ML-Tanks Code> … </ML-Tanks Code>. The files where significant code changes were made or added are listed below. Changes to the various Unity project files and objects were also made.

Significant Files:
Assets/Scripts/Managers/GameManager.cs
Assets/Scripts/Managers/TankObservable.cs
Assets/Scripts/Managers/TankFlock.cs
Assets/Scripts/Managers/TankHealth.cs
Assets/Scripts/Managers/TankFactory.cs
Assets/Scripts/Managers/Simulation.cs
Assets/Scripts/Managers/SimManager.cs
Assets/Scripts/Managers/SimData.cs
Assets/Scripts/Managers/SimObserver.cs
Assets/Scripts/Managers/SimController.cs

Final State of System

Our initial plan was to produce a simulator that would have machine learning agents control various tanks that belonged to two sides of a battlefield. The leaning agents would battle their tanks against each other and rounds would restart once all tanks on a side were destroyed. The machine learning agents would train as the rounds went by and the client would watch the progress of the leaning agents as they trained. While time constraints did not allow us to implement the machine learning agents aspect of the project, we were able to implement the setup to our initial plan. We produced a system where users would input the parameters of a simulation that they wanted to see play out and the system would set up the Unity scene to the parameters set. A more in-depth description of our system will be in the class diagram section. We did not account for the large amount of time needed to familiarize ourselves with the Unity engine and understand the development kit enough to apply object oriented patterns to it. However, we still managed to produce a product that fits the parameters of the assignment.

Final Class Diagram



Model-Viewer-Controller

The SimController class is the centerpiece of the Model-View-Controller pattern used to take user input and translate it to a SimData object form which is what most of the components in our system understands. The viewer aspect of the pattern is taken care of by Unity UI elements. The UI elements allow users to view and input their desired parameters to a form. The inputs of the form are sent to the SimController which applies the relevant values to a SimData object which acts as the model.

Facade

The Simulation class acts as a facade to the various components of the system that allows the unity simulation to take place. At the start of the simulation scene, the SimulationManager object sends the user set SimData to the Simulation object. The data sent to the Simulation object is needed by many components used to run the simulation.

The Simulation object takes care of routing of the parts of the data to the relevant components. The SimulationManager only needs to send the data to the Simulation object to set up the many components needed for the simulation.

Factory

Each tank object in the simulation has complex components to represent it as a whole. A factory pattern was employed to be responsible for properly initializing all parts of the tank object.

Observer

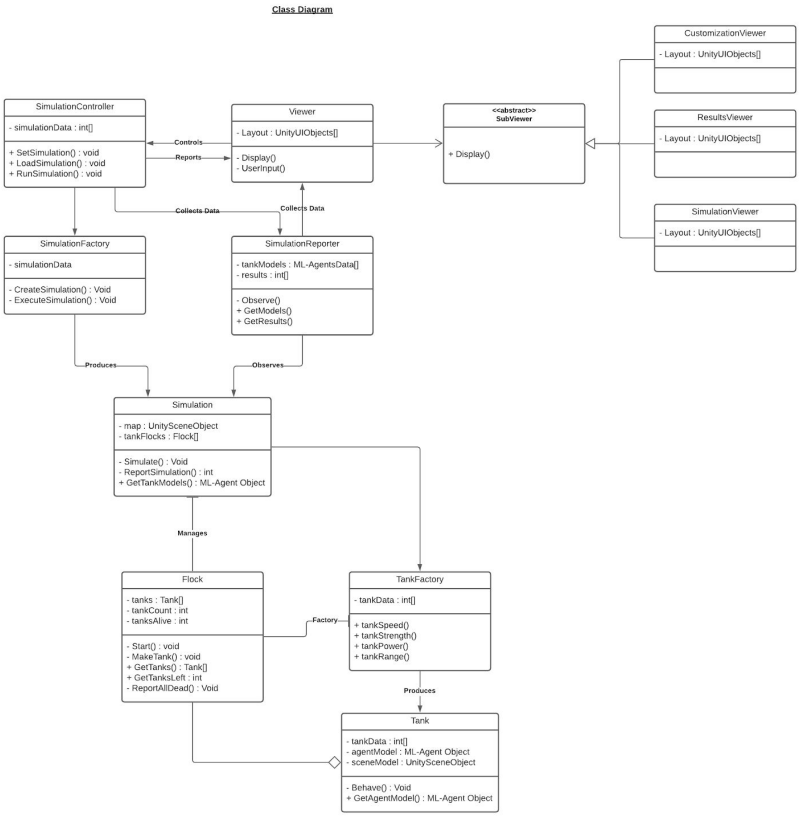
The SimObserver class is a observer pattern that is responsible for reporting on the tanks object of the simulation. Tank objects have a component called TankObservable which SimObservers subscribe to. During the set up phase of the simulation performed by the Simulation object, every tank produced by the Simulation object's tank factory has its observable component set to a reference to the SimObserver to report to.

Singleton

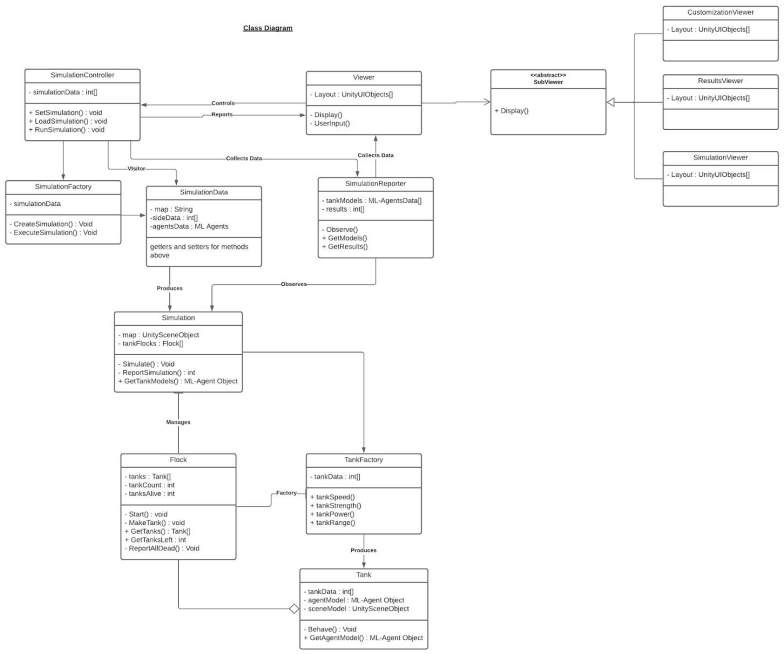
The factory pattern described above employs a Singleton pattern as instantiation of the factory is unnecessary and only one instance of the factory is required.

Strategy

The tutorial for pluggable ai for the tutorial Tanks! project(linked above) uses a strategy pattern to control the behavior of the tanks. While the current iteration of the project does not utilize it much, we intended on having machine learning agents classify which strategy to employ based on the objects close to each of the tanks. The idea was to have the learning agent dictate which strategy was swapped in or out.



UML Class Diagram for Project 4



UML Class Diagram for Project 5

The overall structure of the class diagrams stayed the same as our iterations went by. The main changes we made were the switching out various patterns used when implementing certain parts of our project as our ideas of how to implement certain aspects of our project matured. The changes were due to our changing views of the various problems each component needed to solve. A significant change since our diagram from project 4 was to implement the Simulator object as a facade. We believed that the components needed for the simulation were getting too complicated and wanted a reliable endpoint to send initialization parameters to instead of individually sending the data to all of the components directly. With that redesign of Simulation, SimulationFactory was removed as we did not think it was necessary to make many instances of Simulation.We also slimmed down our user interface design as we learned more about how the Unity UI system simplified the process of setting up an interface.

OOAD Process

One of the issues that we encountered during the OOAD process was the inheritance with the ML-Agents Agent class, as the classes that needed the TankML class to operate on also required some MonoBehavior designs which significantly increased the challenge of converting the original *Tanks!* code into the format that ML-Agents desired. In addition this caused instructions for the MLTank to be spread across multiple different files instead of put together tightly in a single script. This leads to issues when trying to train the model.

We employed Pattern Driven Design when designing the components of our project. We spent a large amount of time at the start of the project timeline planning out what problems had presented themselves and what object oriented patterns could be used to solve each problem. The patterns we utilized were changed as our understanding of our project evolved. We chose this design method due to the nature of the project having a pattern count requirement.

A design issue we had was using object oriented patterns in a Unity environment. We learned to apply the patterns in a Java sense from our course work. Implementing the patterns was not a 1-1 conversion from Java. Converting the patterns from one language to another was more difficult than simply changing syntax, as Unity's Scriptable nature required a slightly different perspective for us.