OOAD Project 6

Project Name: **BattleCOD 3: Eastern Front 2**
Team Members:

- Austin Albert
- Ian Meadows
- Andrew Hack

Github Link:
https://github.com/aual1780/OOP_BigProject
Demo Link:
https://drive.google.com/file/d/1tWUO1-6G1WzQOAK0woFV_VDLbZuW1VRk/view?usp=sharing

Final State of System Statement

Our team has created a local network controller system and a game to connect to and play with it. The network controller system was completed around the time Project 5 was turned in. The majority of the game was done around the time of Project 5 as well. Since Project 5 we have added a high score system, better images for the zombies and the tank, explosions, revamped zombie spawn system, improved wall placement, a shadow for the primary bullet, and bug fixes and tweaks. The majority of features listed in Project 5 were implemented. The only features not implemented were features that were later deemed as not desirable for the game we had created. The major change from project 4 was our realization that Unity and class diagrams don't work well together.

Features that were implemented:

- ArdNet implementation [**Austin**]
  - Low latency networking solution for interacting between client and server
- Package: TankSim [**Austin**]
  - Extensible wrapper library to support code sharing between client and server
- Package: TankSim.Client [**Austin**]
  - Extensible client library to support multiple UI types. Includes fully encapsulated API endpoints for easily interacting with the networking subsystem. Manages game connection, failure recovery, command send/receive, dynamic role binding
- TankSim.Client.GUI [**Austin**]
  - GUI client to connect to the gamehost. Uses TankSim.Client subsystem. Player can join a game, set their name, and input commands. A simple HUD shows game state relevant to the user role. Includes gamepad integration
- TankSim.Client.CLI [**Austin**]
  - CLI client to connect to the gamehost. Uses TankSim.Client subsystem. Player can join a game, set their name, and input commands.
- Package: TankSim.GameHost [**Austin**]
  - Extensible server library to support multiple UI types. Includes fully encapsulated API endpoints for easily interacting with the networking subsystem.
- TankSim.GameHost.CLI [**Austin**]
  - Testing platform for server/client connectivity and event binding. Since both this and the Unity project share the same TankSim subsystems, we were able to test the process interactions separately from the game.
- Unity UI implementation [**Ian**]
  - Create a UI implementation in Unity using the TankSim.GameHost subsystem. This allows players to create a lobby, join, and start the game.
- Unity tank operation implementation [**Ian**]
  - Bind on-screen tank operations to the TankSim.GameHost API. This allows a tank to be draw on-screen and operated by the controller modules
- Unity enemy implementation [**Andrew**]
  - Create game enemies and define their gameplay rules. This includes enemy spawning, movement, attacks, damage, and score rewarded
- Unity game field features [**Ian, Andrew**]
  - Limited play field
  - Wall and cobble stone placement to add life to the scene
    - Improved wall placement so they are not on top of themselves or where the player spawns

- Scoreboard [**Ian**]
    - Viewing high scores
        - Shows top 10 high scores
        - Gold panel that shows if player's score is in top 10
    - Placing High scores
    - Shows player's score after a game ends
        - Visual if they just score the highest overall score

- Game prettification [**Ian, Andrew**]
    - Real tank sprite
    - Explosions
    - Bullet shadow for the primary gun. This is supposed to represent a mortar shot.
    - Blood splatter where a zombie died.
        - Slowly fades over time
    - Zombie visuals
        - Real zombie sprite
        - Movement animation
        - Scale to show threat

- Game QoL improvements [**Ian, Andrew**]
    - Better visual feedback
        - When a misfire happens
        - Explosion when the primary gun lands (large or small explosion)
        - Massive explosion when the tank's health runs out.
        - Primary gun bullet shadow to give the players a better ability to time when it will land.
    - Better Zombie spawning

Features that were not implemented:

- Cross platform GUI client [**Austin**]
    - This was deemed low priority and removed from project scope

- Game stat tracking [**Ian, Andrew**]
    - Info about tank operations during the game
        - This was deemed unfit for the game we had created. The only statistic we want players to care about is score.

- Ammo and health pickups [**Ian, Andrew**]
    - These were deemed unfit for the game we had created. We wanted players to focus on the fun part of the game. Which was dodging the zombies and also shooting at them the 3 different weapons. Adding ammo pickups would ruin that part of the game because then the players would be limited for the fun part. Health pickups would increase the length of the game which is bad for this game because we wanted the game to be a fun, hard, and short experience.

Class Diagram

The class diagrams have remained mostly consistent since their inception in Project 4. The largest changes are found in TankSim.GameHost and the Unity project. The GameHost library received some additions to internally manage more of the networking system. This simplified the startup process for consumers.

The core modules and controllers were feature complete at the end of Project 5 and did not receive major changes during Project 6 (only debugging).

The Unity class diagram has changed drastically from Project 4 to Project 5. The reasoning behind this was because of how hard it was to actually do some of the design patterns. One such design pattern that suffered from this was the decorator pattern for the zombies. It still exists, but uses Unity's component system to decorate a Zombie game object with script components that modify the zombie's stats. We also did not have a good understanding of how the Gamehost would operate.

The Unity class diagram has changed a little from Project 5 to Project 6. The main edition to the class diagram was the implementation of the high score system. There were other smaller additions as well. These additions were added to make the game more readable and more visually appealing.

Unfortunately, the Client and Gamehost project is not conducive to diagramming. Every aspect of the project relies on some form of DI, service locator, or remote procedure call. That means that it's impossible to get the full picture from a simple class diagram. Sequence diagrams and activity diagrams convey the system complexity more accurately.

Client and Gamehost Patterns

Mediator/Command: The core of the networking system is a mediator/command structure, but the complex parts are abstracted away by ArdNet, so it doesn't appear in the diagrams.
DI/Flyweight/Singleton: The ArdNet system does appear in the diagrams in other ways, though. You will notice many actors take an IArdNetClient or IArdNetServer as an argument. This is a shared object injected via DI Container wherever it is needed.

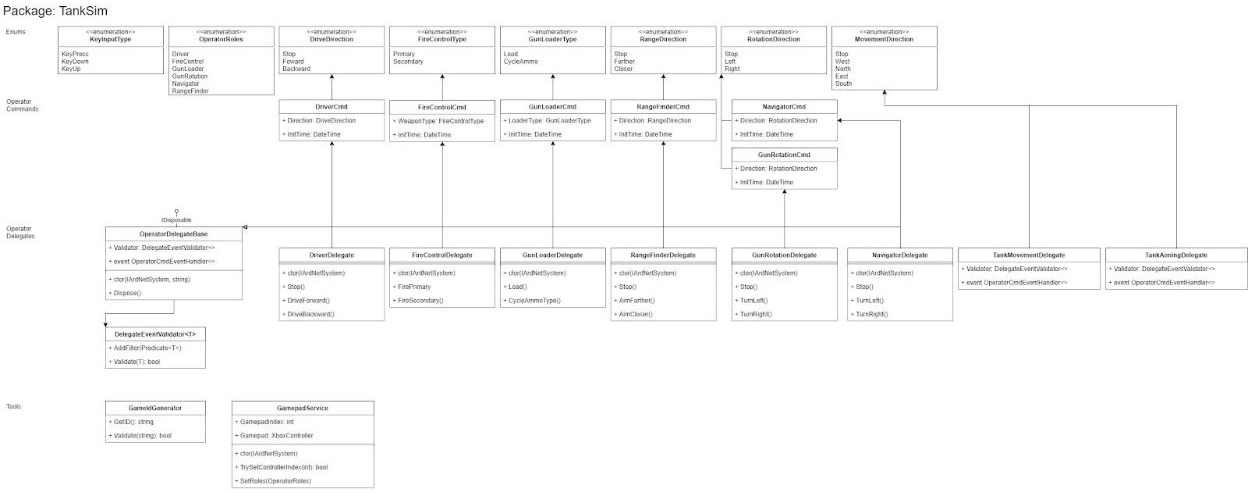
Proxy: We use proxy objects to interact with ArdNet named topic channels. This is another internal feature of ArdNet that we are consuming

DI/Builder: We use the ArdNet fluent builder API to prepare and configure the network interface. This also does not appear in the UML, but can be seen in the client startup code.
Facade: This pattern does appear in the UML. TankSim.GameHost.OperatorCmdFacade simplifies access to ArdNet event hooks to make them easier to access.

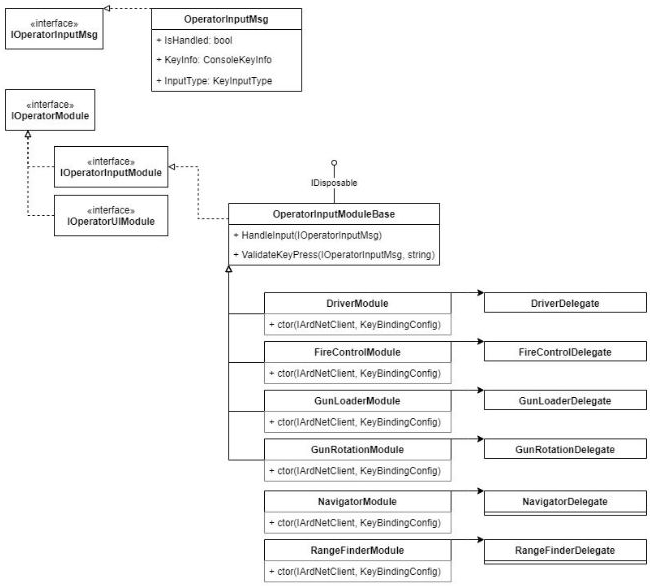
Chain of Responsibility: ArdNet.Client uses a CoR to send user input to a collection of operator processor modules. One (and only one) of the processors will respond to the input depending on what the user does.

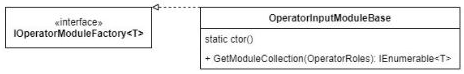
Original Diagrams (from Project 4&5)

Package: TankSim

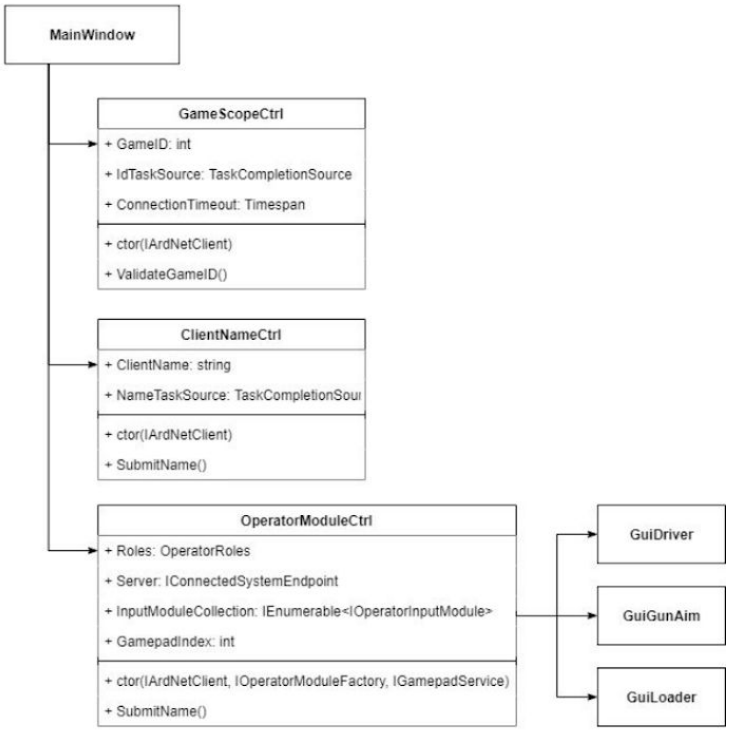


Package: TankSim.Client

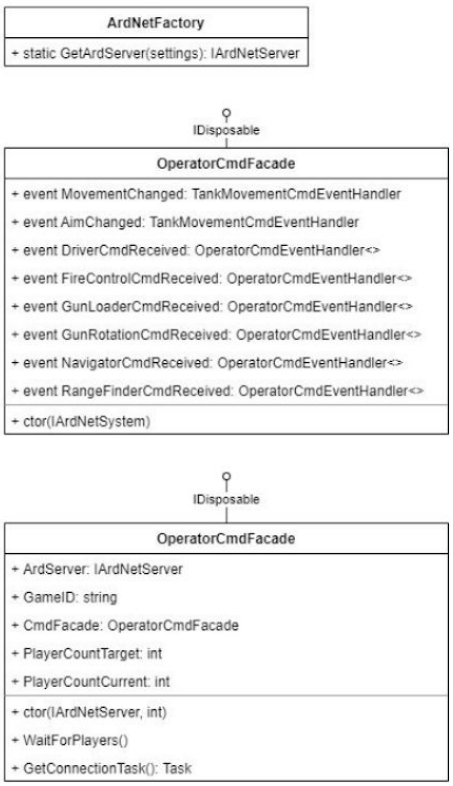




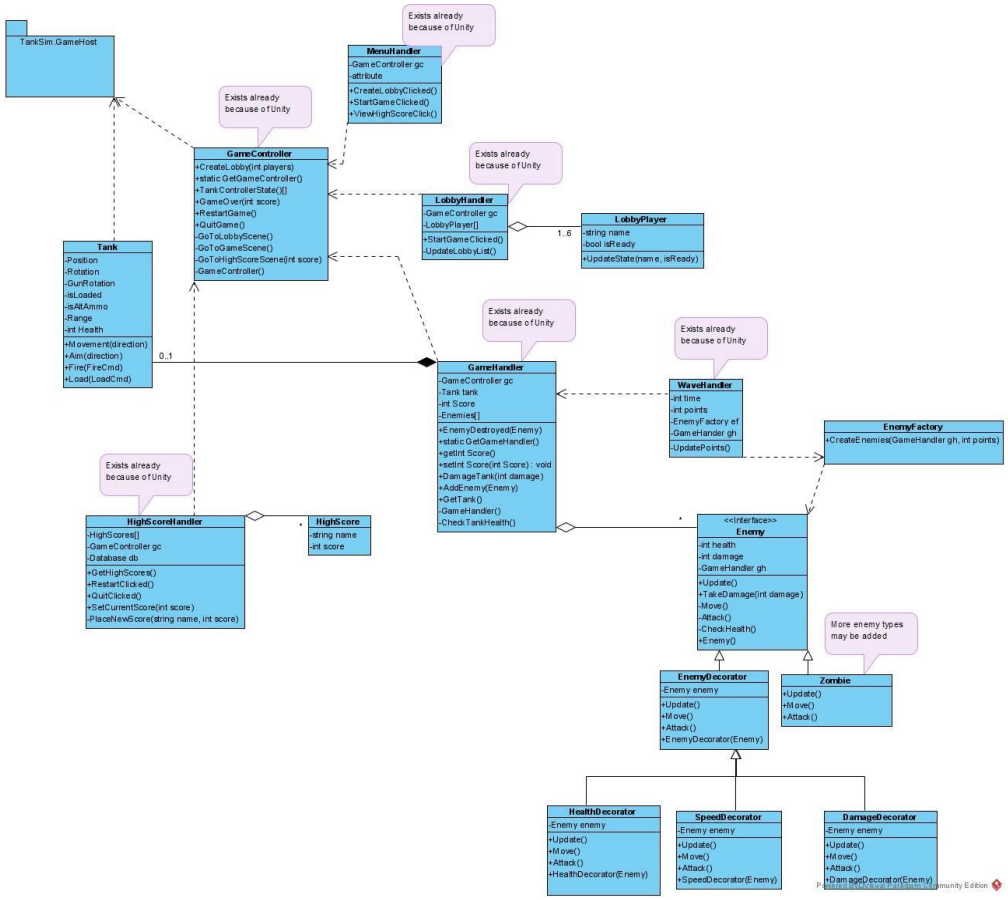
Package: TankSim.Client.GUI



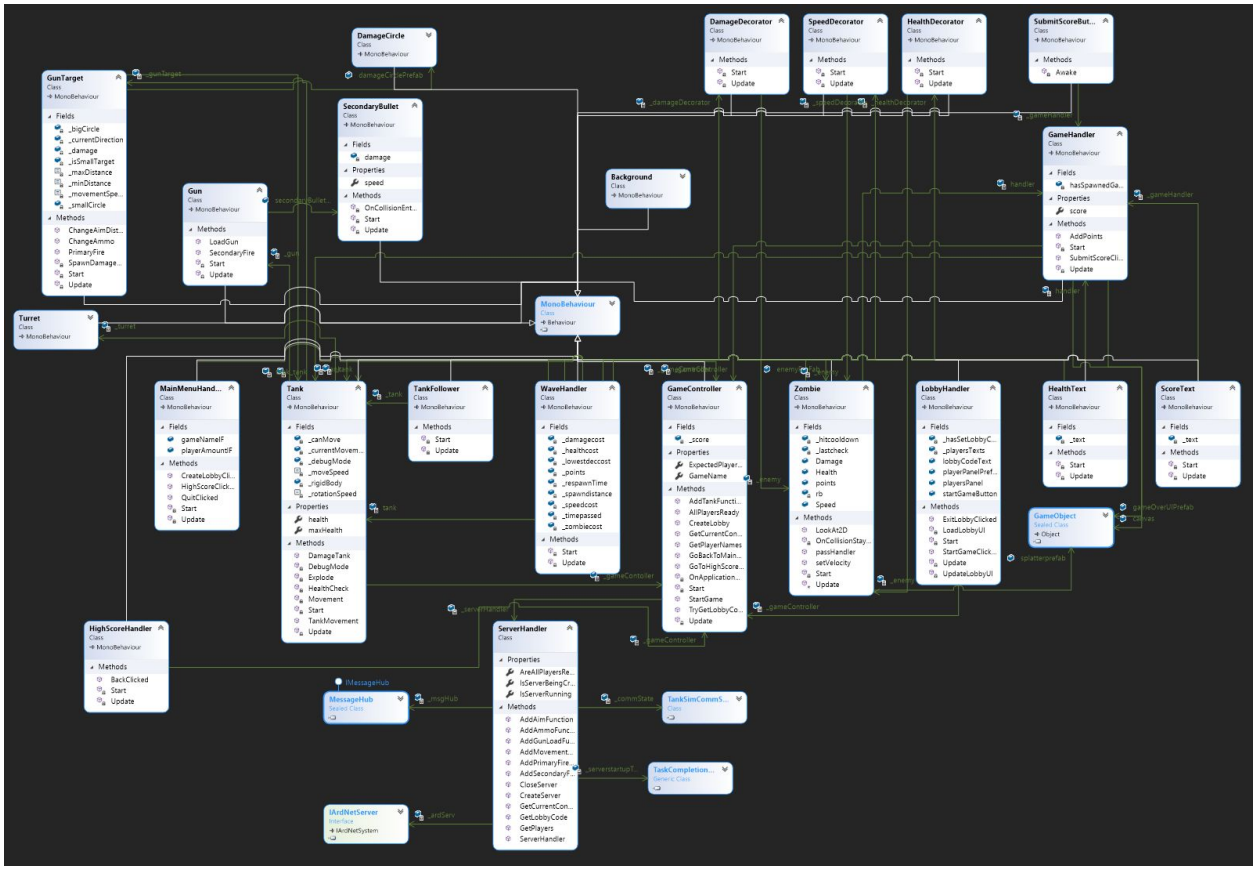
Package: TankSim.Gamehost



Project 4 Unity Class Diagram:

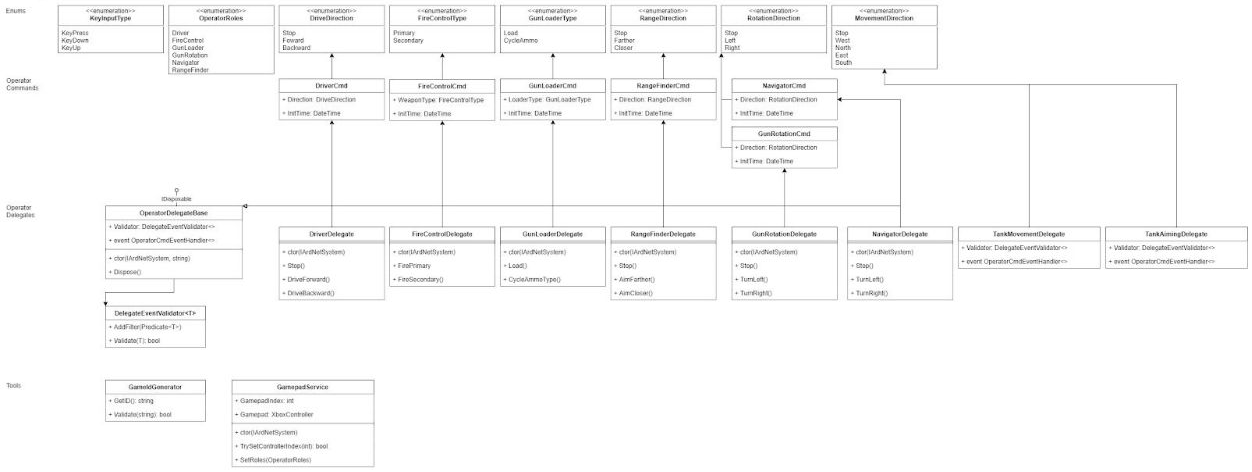


Project 5 Unity Class Diagram:
Note: Unity does not translate well to a class diagram, because of its game object and component system. This is our best shot at representing our Unity project as a class diagram.

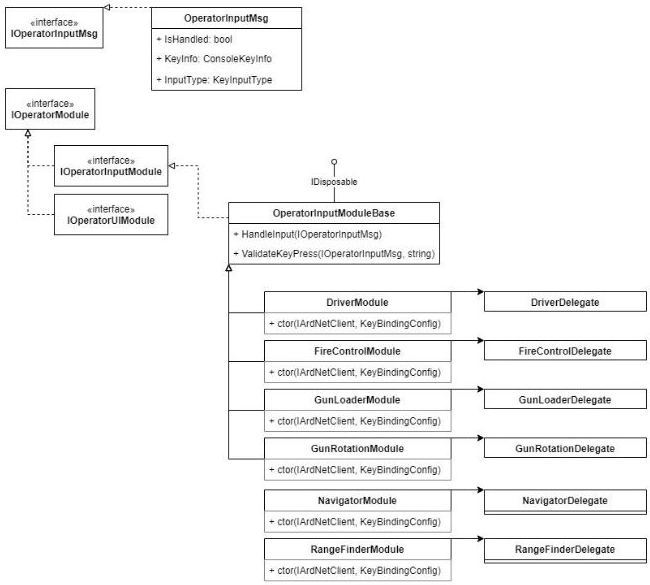


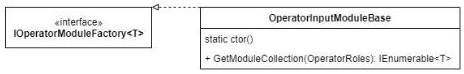
Current Diagrams

Package: TankSim



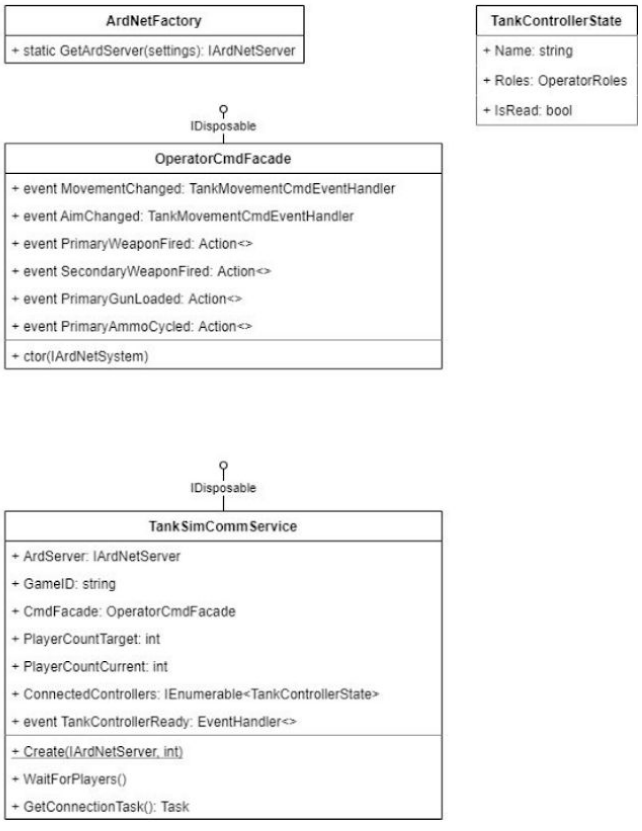
Package: TankSim.Client





Package: TankSim.Client.GUI

Package: TankSim.Gamehost



Project 6 Unity Class Diagram:

Note: Unity does not translate well to a class diagram, because of its game object and component system. This is our best shot at representing our Unity project as a class diagram.

Game Design patterns:

1) Kind of a Decorator using Monobehaviour
2) Façade for server handling
3) We would have had some singletons for our handlers, but Unity does not handle singletons well.

Third Party Code Statement

We are using MS .Net and Unity to support our applications.

All TankSim packages are original work. J2i.Net.XInputWrapper was originally taken from a tutorial at one point many years ago that has since been lost. The project relies heavily on the ArdNet and TIPC.Corelibraries - these are maintained by Austin, but are considered out of scope for this project. For a more indepth listing, see the github dependency insights.

The only two main third party sources used in the game are the Utf8Json.dll file and the UnityMainThreadDispatcher.cs. This dll is used to save and load Json files. This is used for our high score system. The cs file is used to fix Unity threading issues with the main Unity thread and separate threads. This is used for our tank controller system.

All other 3rd party sources are commented in the code.

OOAD Design Process

The initial design was a simple 3 component architecture diagram: Client, Middleware, and Server. The middleware component was the first to be outlined. This included networking spec, message types, service endpoints, and rough consumer API. Having all of this in place early allowed the 3 components to be siloed from each other. As long as the middleware networking remained stable, the client and server could be implemented separately from each other. This allowed the project to advance even if some components lagged behind.

(written by Austin) Immediately following the architecture diagram was a storyboard combining sequence diagrams, activity diagrams, and UI diagrams. I believe these aspects are more important than class diagrams or class breakdown structures during the early process. This allowed us all to get on the same page about the project. Since the server and client were isolated so well, we were able to continue with more detailed planning as separate teams. In theory, it had all the ingredients to work perfectly. In practice, I question the hands-off approach I took with the rest of the team given their lack of responsibility.

When we started working on Project 5 we realized that our diagram for Project 4 was naive. Unity has its own way of doing things that does not work well with a traditional class diagram. As a result, some of the patterns that we planned out ended up being implemented completely differently, even if they were functionally the same. For example, the enemy decorators still modify the damage, health, and speed of the zombies, but they are unable to do it as decorators in the traditional sense due to limitations placed on us by Unity.