Swinburne University of Technology
Faculty of Science, Engineering, and Technology

COS20007: Object Oriented Programming

Artillery 3 Design Document

Date of report submission	11/06/19
Lab Supervisor	Medhi Naseriparsa
Group	10:30am Wednesday

Name	**Student ID**
Jimmy Trac	101624964

This page is mostly blank.

The purpose of this page is to separate the cover page from the report when printed double-sided.

Artillery 3

Design Document, written by Jimmy Trac (101624964)

I. Preface

This report is a quick and concise dive into the many design decisions made within Artillery3 (A3). Due to the complexity of the program, a written and described form would have proven more useful than just the Unified Modelling Language (UML) Class Diagram on A3 and all its iterations.

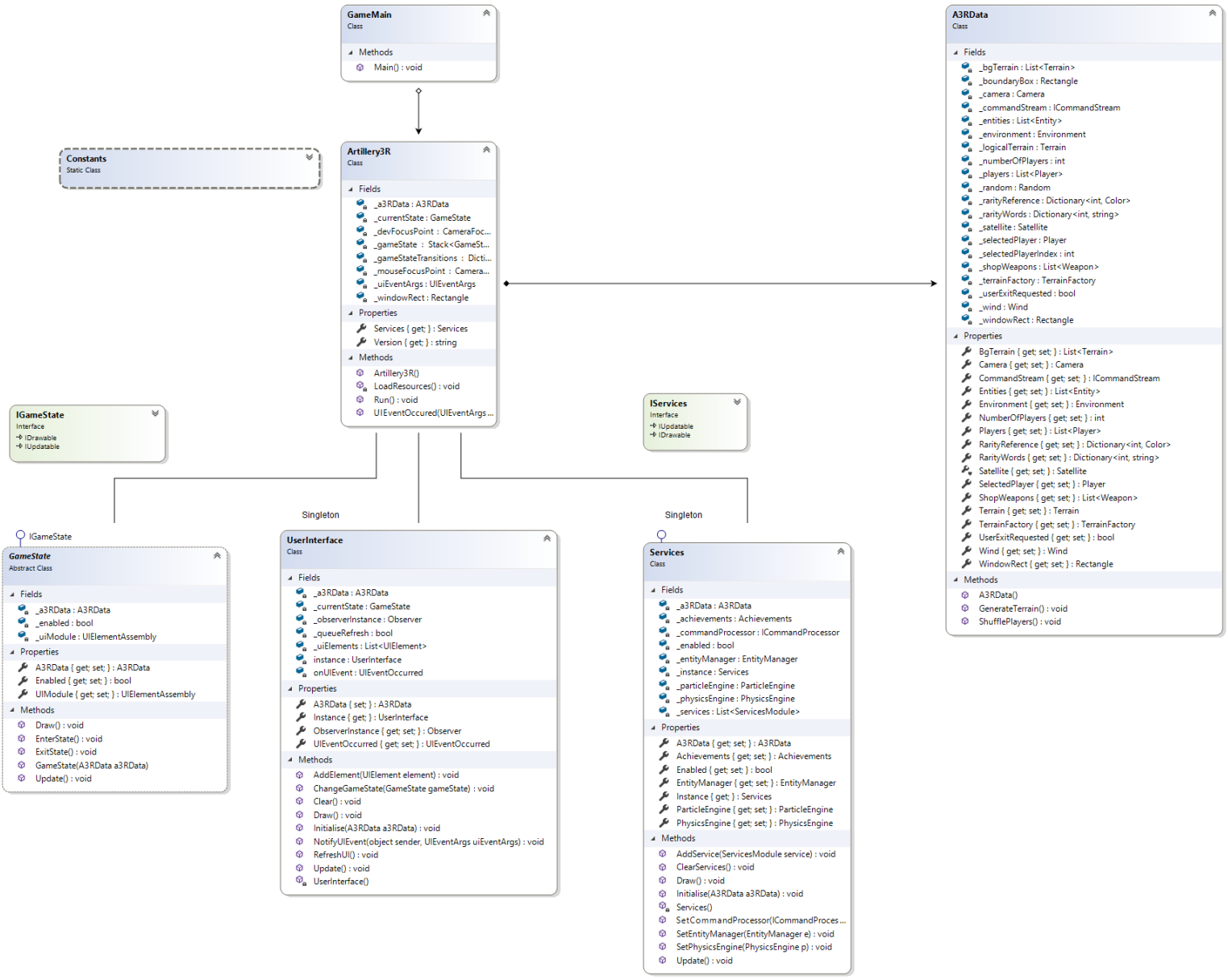
Yes, this report is text-light. That's intentional.

II. Table of Contents

`

I. Top Level Overview

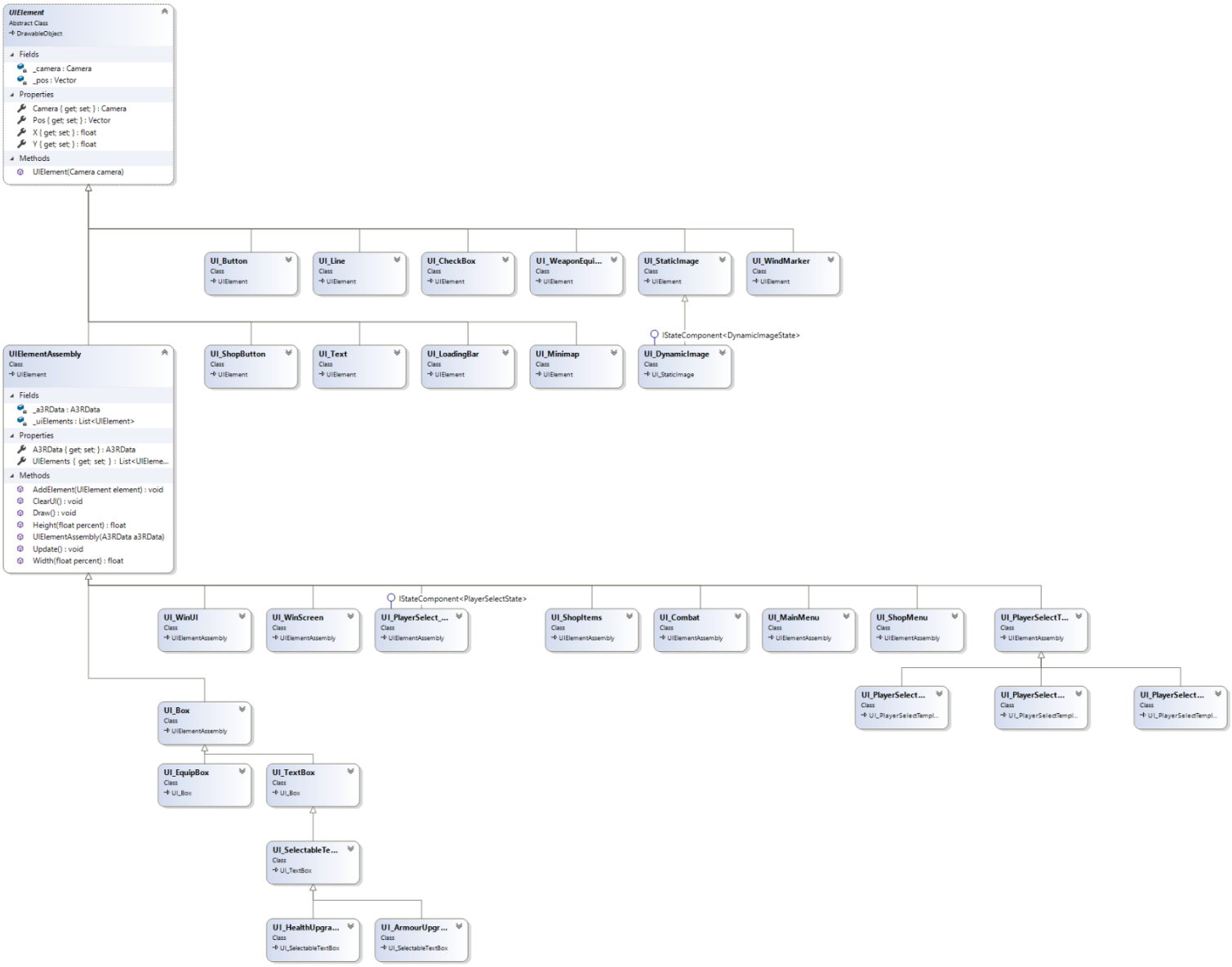
A quick top-level explanation of the major systems within Artillery



The above figure represents a high-level overview showing the adoption of the Model-View-Controller paradigm, albeit in a non-ideal fashion. The Model would be A3RData, whereas the views would encompass the User Interface and Game state, whilst the Game State contains most of the controller.

Of note are services, which includes an extensible Service Module System which was intended to house the Console Commands system but was never implemented.

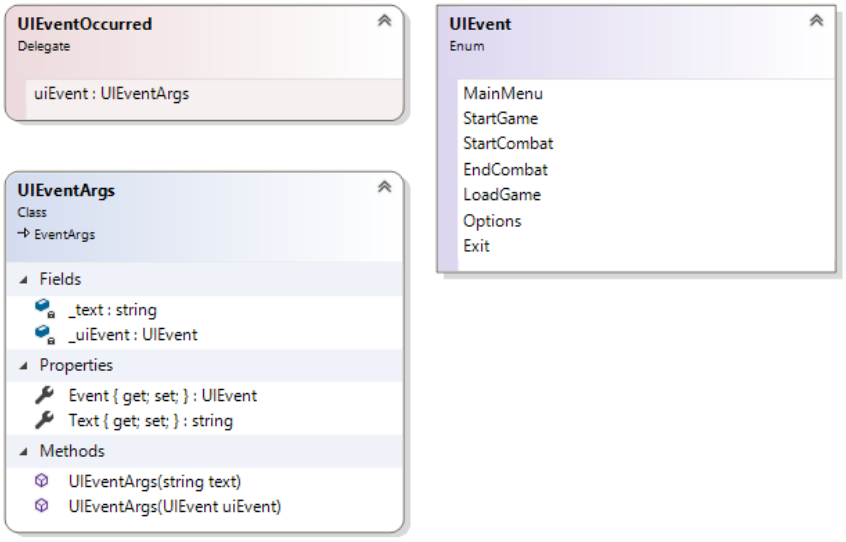
1.1. The User Interface System



The above image only shows the inheritance within the UI system to show the extent at which it utilises the composite pattern. In actuality, the User Interface Element Assembly (UIElementAssembly) “uses” (one-way association) the User Interface Element, creating a composite pattern in which all higher-level modules simply employ a root UIElementAssembly as the base class which contains the elements within it. A good example of this is the first line under the UIElementAssembly, which represent the UI Interfaces a player sees in-game, including the UI during Combat, the Win Screen, Main Menu, Shop Menu, and even Player Select menus.

The composite pattern allows components to compose of other elements such as with the UI_Box element, which itself is used in the UI_TextBox and then the UI_EquipBox (which also uses the WeaponEquipBox and UI_Text elements).

This form of composition is highly useful when designing modular interfaces as it facilitates code reuse and ease of designing new interfaces as a new class that inherits from the assembly can act as an entire UI.

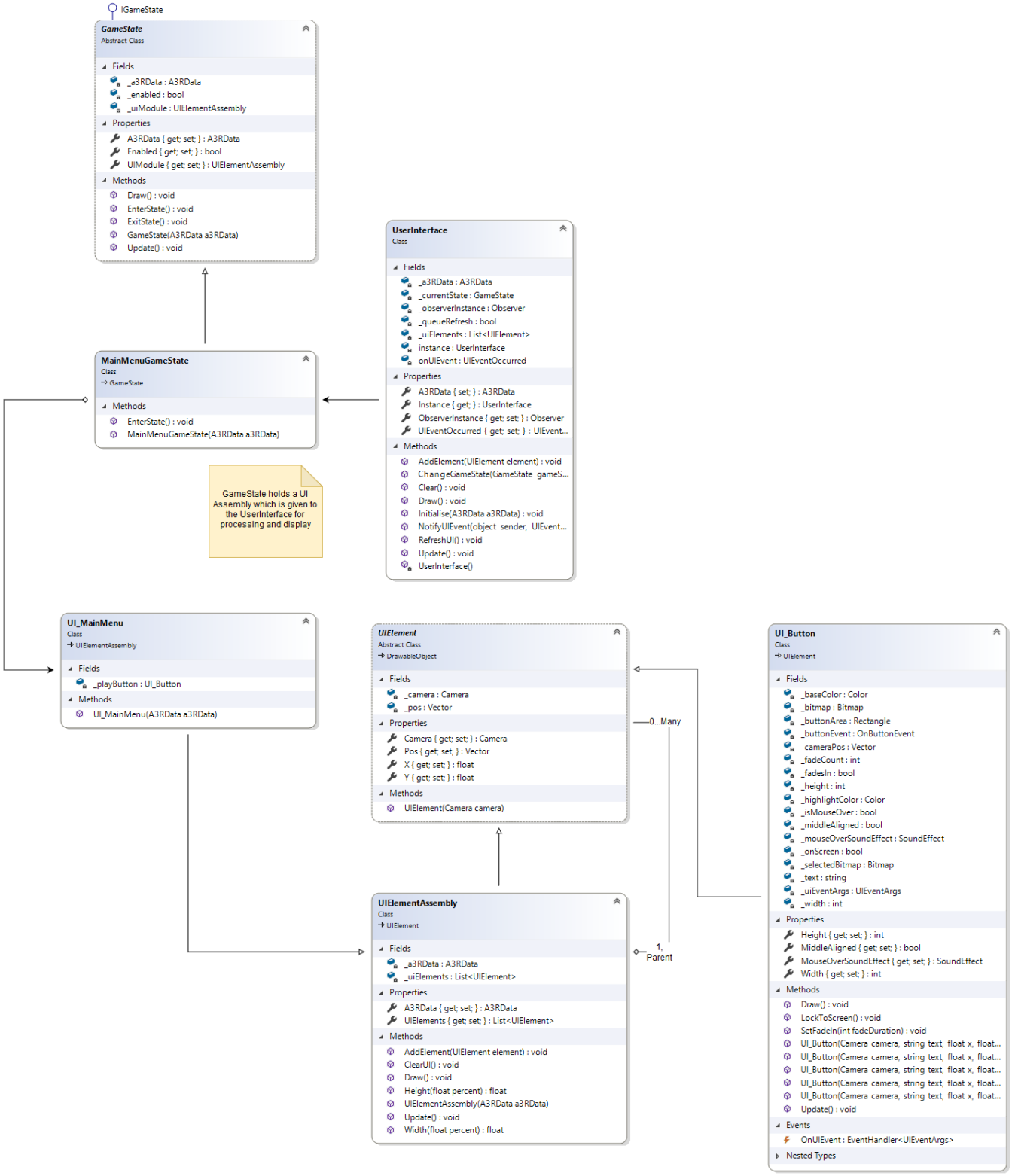


The diagram above shows the event-based system (though realistically it acts simply as a delegate) that allows the User Interface root class (the singleton) to communicate with the base game class. This both facilitates the use of the Dependency Inversion Principle (DIP). The DIP Implies that the UI is not coupled to the base A3 system and can operate in its absence as it only calls the delegate it is assigned – if there is no assigned delegate, the calls simply come unanswered.

The important part here would be the use of UIEventArgs, or User Interface Event Arguments, which specify to the subscriber (the game) if an important UI event has occurred, such as a player pressing the "Start Game" button. The UI Event enumeration is tied directly to a dictionary that would allow the game to queue a change of game state without disturbing the UI system (as immediately changing the game state would disturb the delegate call from a UI element), and allow the game to properly exit and enter states (for example, stopping and starting music).

As the game uses a dictionary to link the UI event to a game state, the game itself is never fully aware of the state that it is in, giving way to the principles of encapsulation and information hiding. It is only aware that it should run the current state.

An interesting aspect that was implemented in A3L and A3R but not re-implemented in A3RX nor A3s is the Escape-Menu system, which would've required the Services Module to directly interact and interface with the UI System and the Game State system. This will be further discussed in section 1.3. Game States.



The above image shows the UML Class Diagram for the Main Menu with its 4 buttons. Each button is a UIElement, which is a Button.

A good example of the flexibility of the composite system is the new background in A3s. In the constructor of the Main Menu, I simply added a Static Image UI Element passing into that constructor the name of the background that I wanted and the position 0, 0:

```
AddElement(new UI_StaticImage(Camera, 0, 0, SwinGame.BitmapNamed("fullBg")));
```

The reason for passing the camera is such that if the camera moves, the UI elements stay “On screen”, which is utilised extensively in the shop menu.

The code for adding a new button is as follows:

```
_playButton = new UI_Button(a3RData.Camera, "New Game", _windowRect.Width * 0.026f,
_windowRect.Height * 0.417f, UIEvent.StartGame,
                SwinGame.BitmapNamed("startButton"),
SwinGame.BitmapNamed("startButtonSelected"));
            _playButton.OnUIEvent += UserInterface.Instance.NotifyUIEvent;
            _playButton.MouseOverSoundEffect =
SwinGame.SoundEffectNamed("menuSound");
            AddElement(_playButton);
```

Following the signature:

```
public UI_Button(Camera camera, string text, float x, float y, UIEvent uiEvent, Bitmap
bitmap, Bitmap selectedBitmap)
```

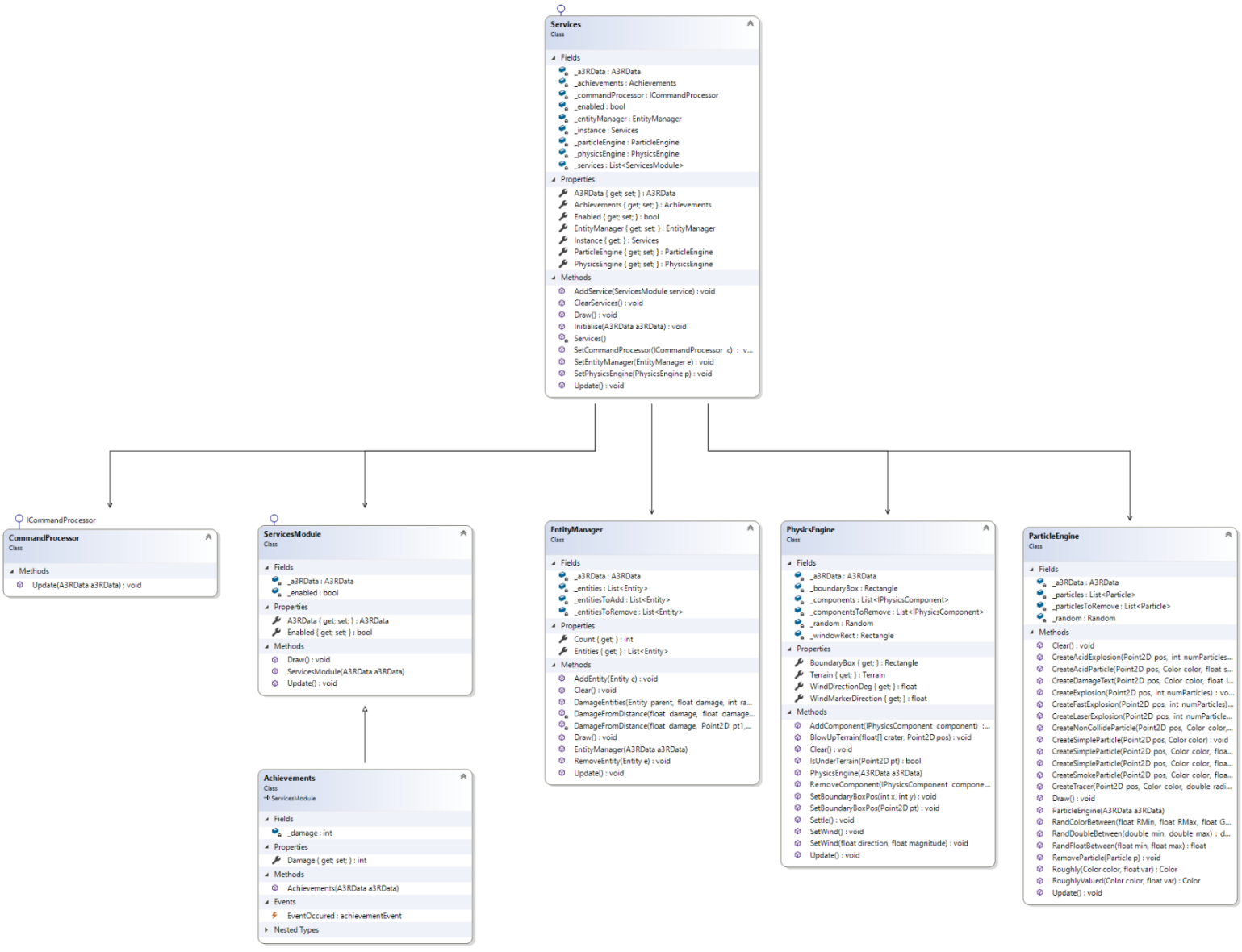
All of the UI is constructed of UI elements, notably the shop uses UI elements extensively due to their flexible nature of being updated by the UserInterface Singleton, they can contain transient information such as a count for animations or moving animations based on targets co-ordinates such that they ease into position, as seen with most of the shop UI.

1.2. The Services System

The services system is a relic of A3L (Legacy) which had the Physics Engine, Entity Manager, and Particle Engine as individual singletons. By collapsing all the singletons into one, we can allow for the use of flexible services. A service in this case, is defined by a singleton class that can be 'enabled or disabled' by a game state. For example, the Physics Engine doesn't run during the Main Menu nor the Shop state but does so during the Combat state.

The Services System also implements an extensible Service Module system which would allow custom-defined services to run and modify the model in later revisions.

A very good example of the Service Module System would have been an achievements system; however, some coupling still exists due to time constraints.

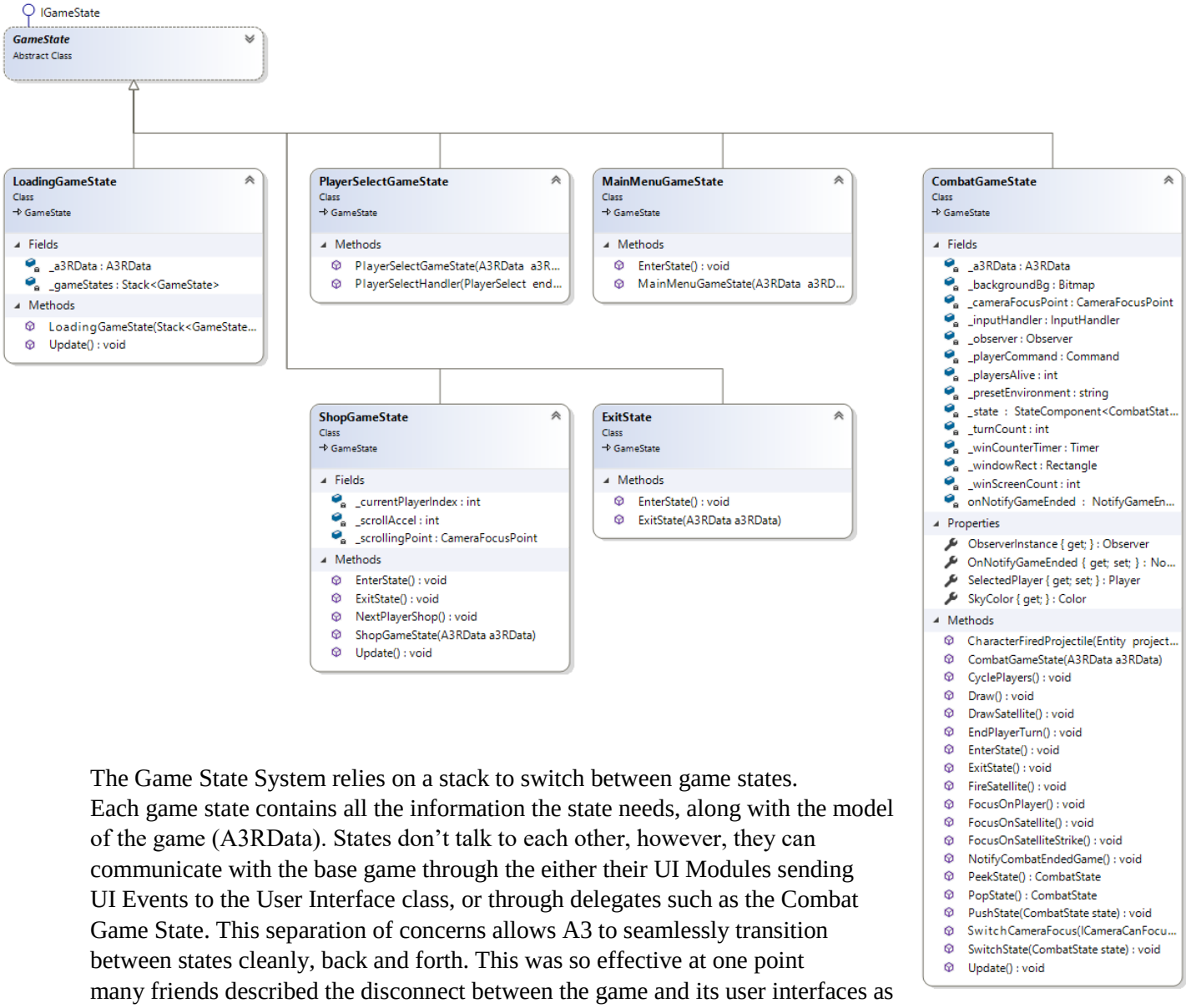


If anything, it's the services module that requires the most re-work to reduce coupling, but for now, it is a functional but old system.

1.3. The Game State System

We've saved the best for last! This system is the biggest.

Let's talk about the main game states first:



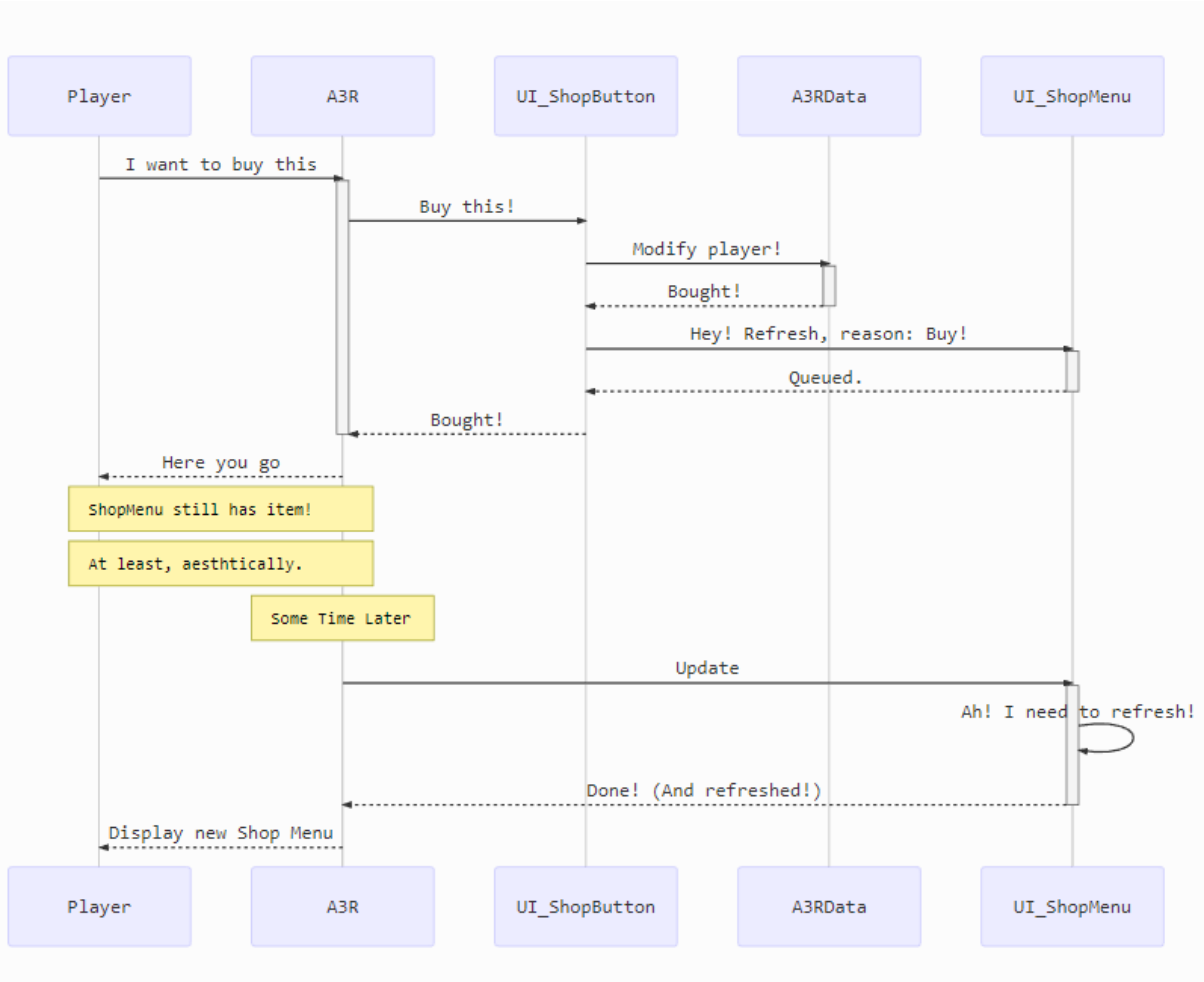
The Game State System relies on a stack to switch between game states. Each game state contains all the information the state needs, along with the model of the game (A3RData). States don't talk to each other, however, they can communicate with the base game through the either their UI Modules sending UI Events to the User Interface class, or through delegates such as the Combat Game State. This separation of concerns allows A3 to seamlessly transition between states cleanly, back and forth. This was so effective at one point many friends described the disconnect between the game and its user interfaces as uncanny.

The State System (Sequence Diagrams!!)

The game doesn't know what state it's in. It only knows to run the current state as the parent Game State object. This means you can add new game states by inheritance from the Game State Base Class if there is a dictionary entry to allow the game to queue the transition.

While we're here, let's talk about transitions. Modifying collections whilst inside them is a royal pain, hence, a loading system is used for states and a queuing system is used for the UI to facilitate for ease of transitions between states and modifying collections without crashing the game.

The Refresh Queue



The UI buttons operate by checking for a mouse click during their update sequence, hence, if the UI collection needs to be modified (such as removing an item from the shop) after a button is pressed, we can't just call the event to modify the collection.

The event simply queues the refresh and waits until the next update call in which it does do the refreshing. This is very quick and non-distinguishable in normal use; variations of this are used whenever a collection requires modification—it is queued for later, though it sees most obvious use in the UI.

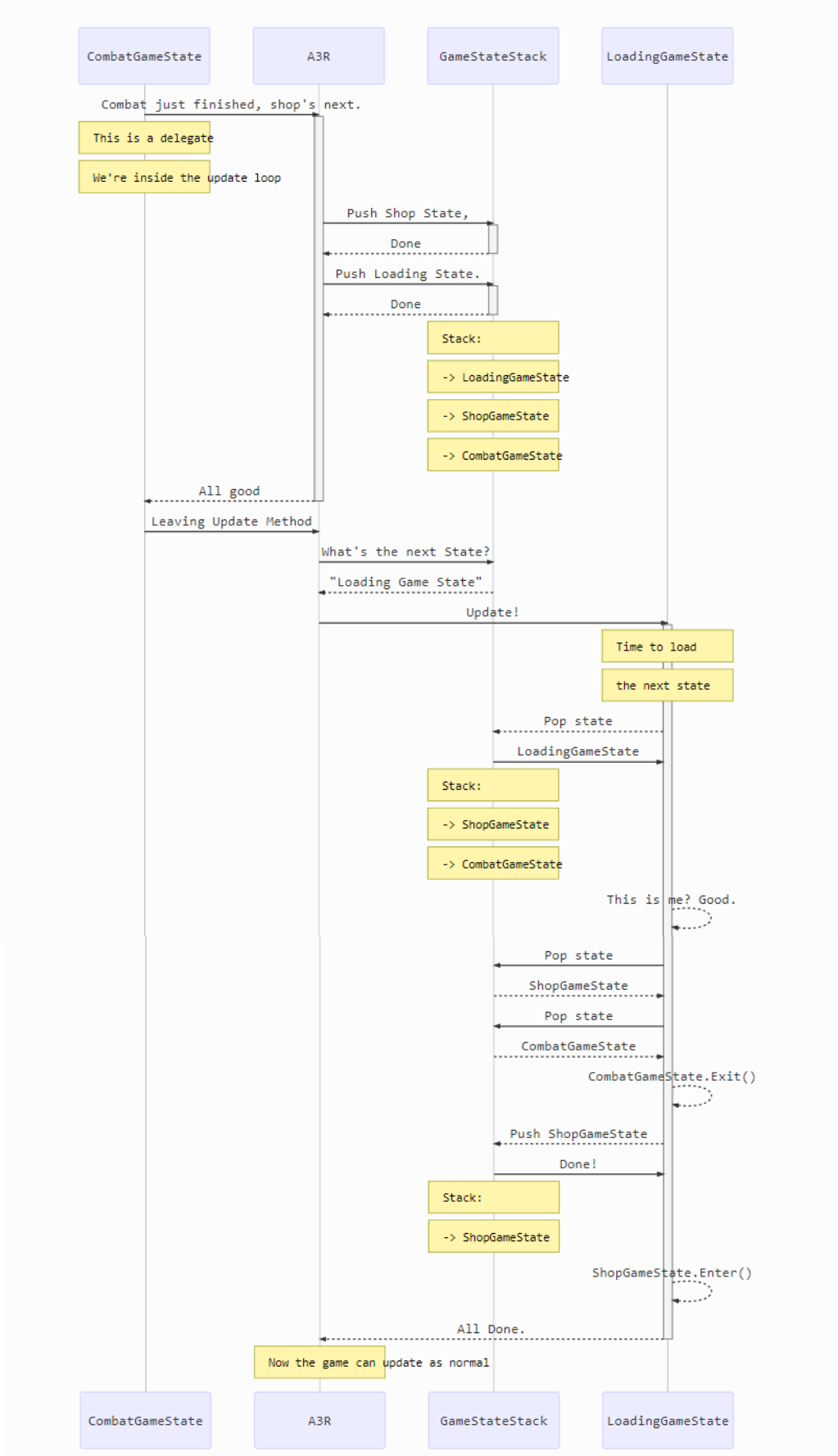
The Loading State and State Stacks

For the same problem, one can't simply change the state as a request from the state since the program pointer is still within that state. It's difficult to simply topple a flag and tell the game to refresh. A simple solution employed by all A3 is the use of the State Stack and a transient loading state.

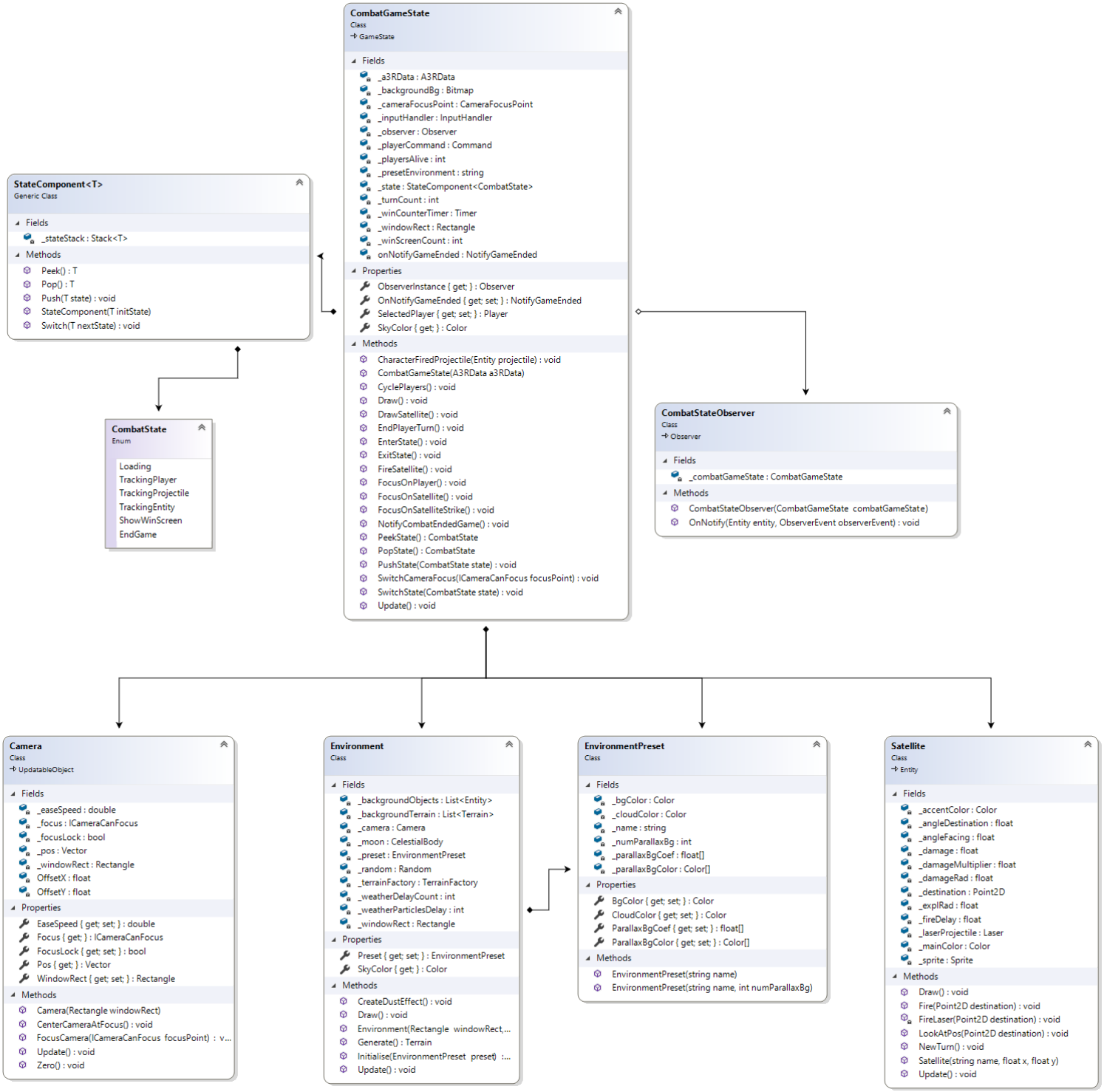
The sequence diagram is on the next page.

The general idea is that the state stack holds the next state and pushes a loading state on top of that. This implies that the next update method will call the loading state's method, which will modify the current game state and exit then enter states before returning the program pointer to the game., This ensures that the change-of-state is done outside of any other state potentially causing issues with modifying active stacks. This system is also scalable, as the loading state realistically knows nothing about the states, only that they are game state and therefore have an enter and exit method.

This allows A3 to transition to and from combat easily, with the enter and exit methods used to start music, heal and reload players, and clearing all objects from the physics engine once the game is over.


CombatGameState
A3R
GameStateStack
LoadingGameState
Combat just finished, shop's next.
This is a delegate
We're inside the update loop
Push Shop State,
Done
Push Loading State.
Done
Stack:
-> LoadingGameState
-> ShopGameState
-> CombatGameState
All good
Leaving Update Method
What's the next State?
"Loading Game State"
Update!
Time to load
the next state
Pop state
LoadingGameState
Stack:
-> ShopGameState
-> CombatGameState
This is me? Good.
Pop state
ShopGameState
Pop state
CombatGameState
CombatGameState.Exit()
Push ShopGameState
Done!
Stack:
-> ShopGameState
ShopGameState.Enter()
All Done.
Now the game can update as normal
CombatGameState
A3R
GameStateStack
LoadingGameState

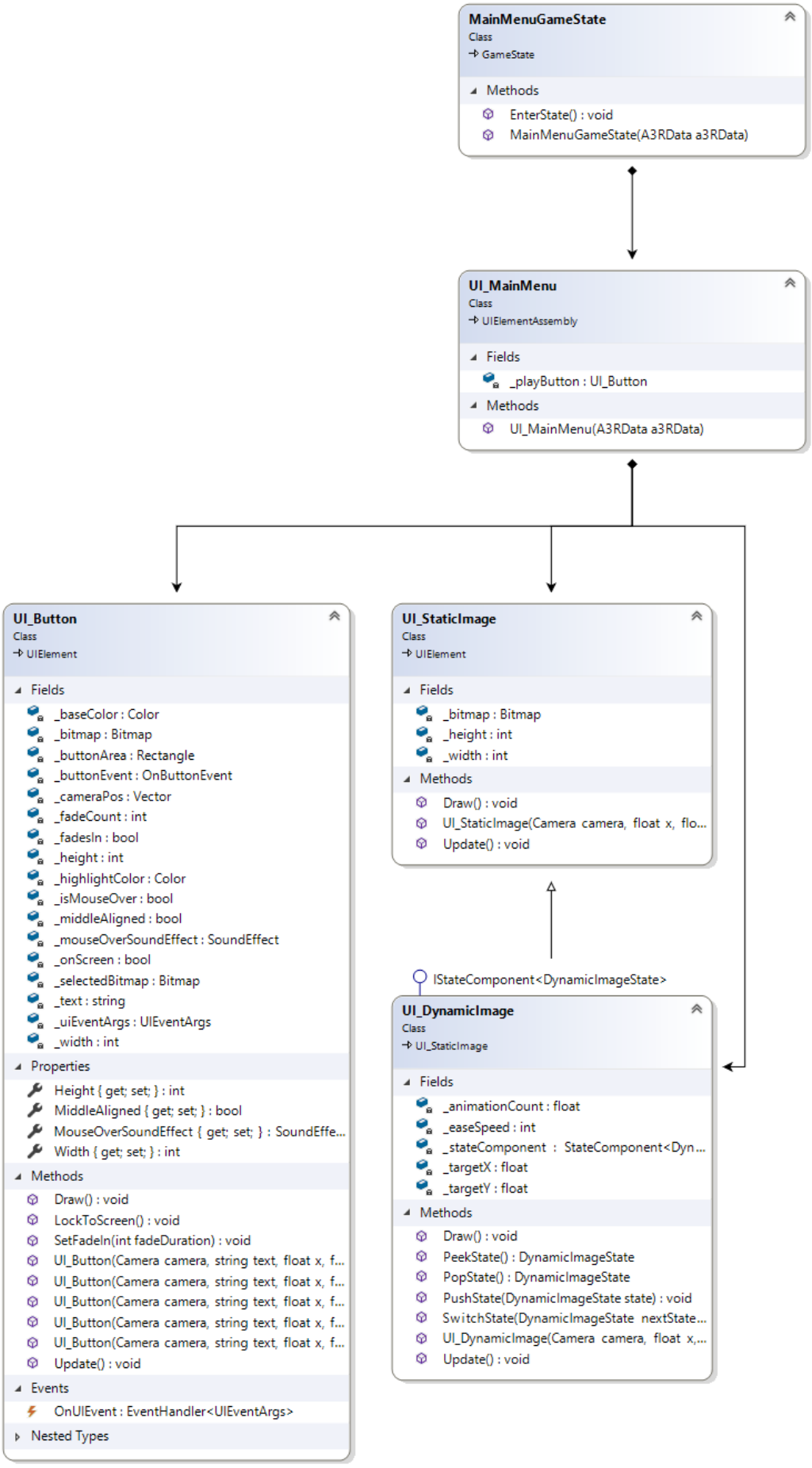
1.3.2. The Combat Game State



The combat game state contains all the information related to combat, and uses the Entity, Physics, and Particle Services by enabling them through Artillery3 Services. The state also draws those services; the game state, despite close ties with the logical terrain, does not manage the terrain, however, it does prompt the physics engine and players to move about.

The combat state contains its own state-component and therefore manages the camera through the current state.

1.3.3. The Main Menu State



The Main Menu game state is a lot simpler than the UML Diagram implies since it relies heavily on the button throwing UI events. Hence, the code for the entirety of the main menu is the following:

```
namespace ArtillerySeries.src
{
    public class MainMenuGameState : GameState
    {
        public override void EnterState()
        {
            SwinGame.PlaySoundEffect("entryboomCombat");
            SwinGame.StopMusic();
            SwinGame.PlayMusic("shopDrone");
            base.EnterState();
        }
        public MainMenuGameState(A3RData a3RData) : base(a3RData)
        {
            UIModule = new UI_MainMenu(A3RData);
        }
    }
}
```

And to think of it from a logical point of view, the Main Menu does not need to know any information regarding the rest of the game, only to know that state to transition to next. The Game State Class is used for logic and composition more than it is for the User Interface, hence, the UI Main Menu class contains most of the logic (events) utilised by the main menu, as seen below:

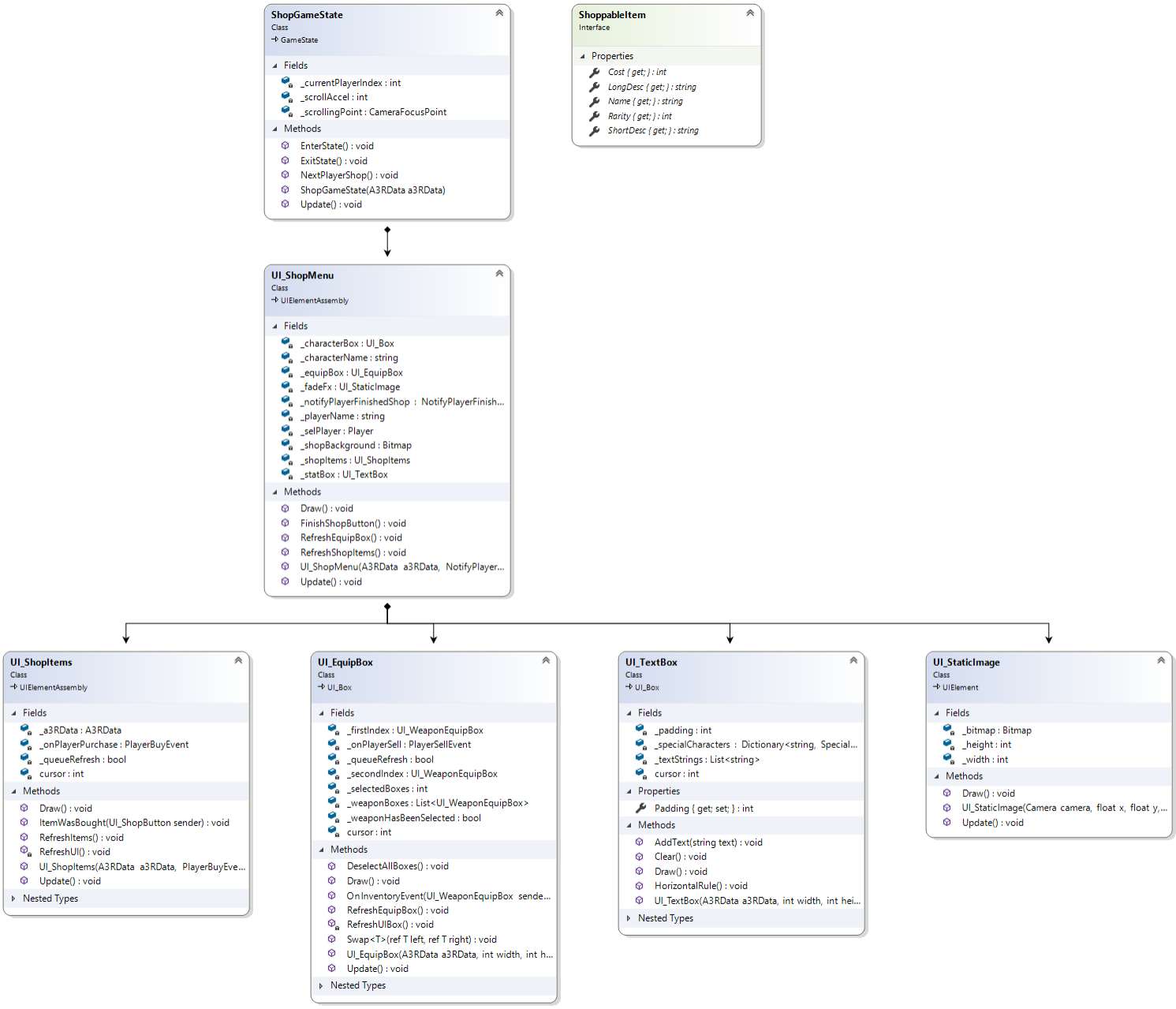
```
        public UI_MainMenu(A3RData a3RData) : base (a3RData)
        {

            AddElement(new UI_StaticImage(Camera, 0, 0,
SwinGame.BitmapNamed("shopBg")));
            Rectangle _windowRect = A3RData.WindowRect;

            AddElement(new UI_StaticImage(Camera, 0, 0,
SwinGame.BitmapNamed("fullBg")));

            UI_DynamicImage _menuGradient = new UI_DynamicImage(A3RData.Camera, -20,
0, -5000, 0, 10, SwinGame.BitmapNamed("menuLeftWhite"));
            AddElement(_menuGradient);

. . .

            _playButton = new UI_Button(a3RData.Camera, "New Game",
_windowRect.Width * 0.026f, _windowRect.Height * 0.417f, UIEvent.StartGame,
                SwinGame.BitmapNamed("startButton"),
SwinGame.BitmapNamed("startButtonSelected"));
            _playButton.OnUIEvent += UserInterface.Instance.NotifyUIEvent;
            _playButton.MouseOverSoundEffect =
SwinGame.SoundEffectNamed("menuSound");
            AddElement(_playButton);
```

Th above code snipper shows the general layout of topmost UI Assembly classes, which contain mostly “Add Element” function calls within the constructor. The rest of taken care of by the composite pattern.

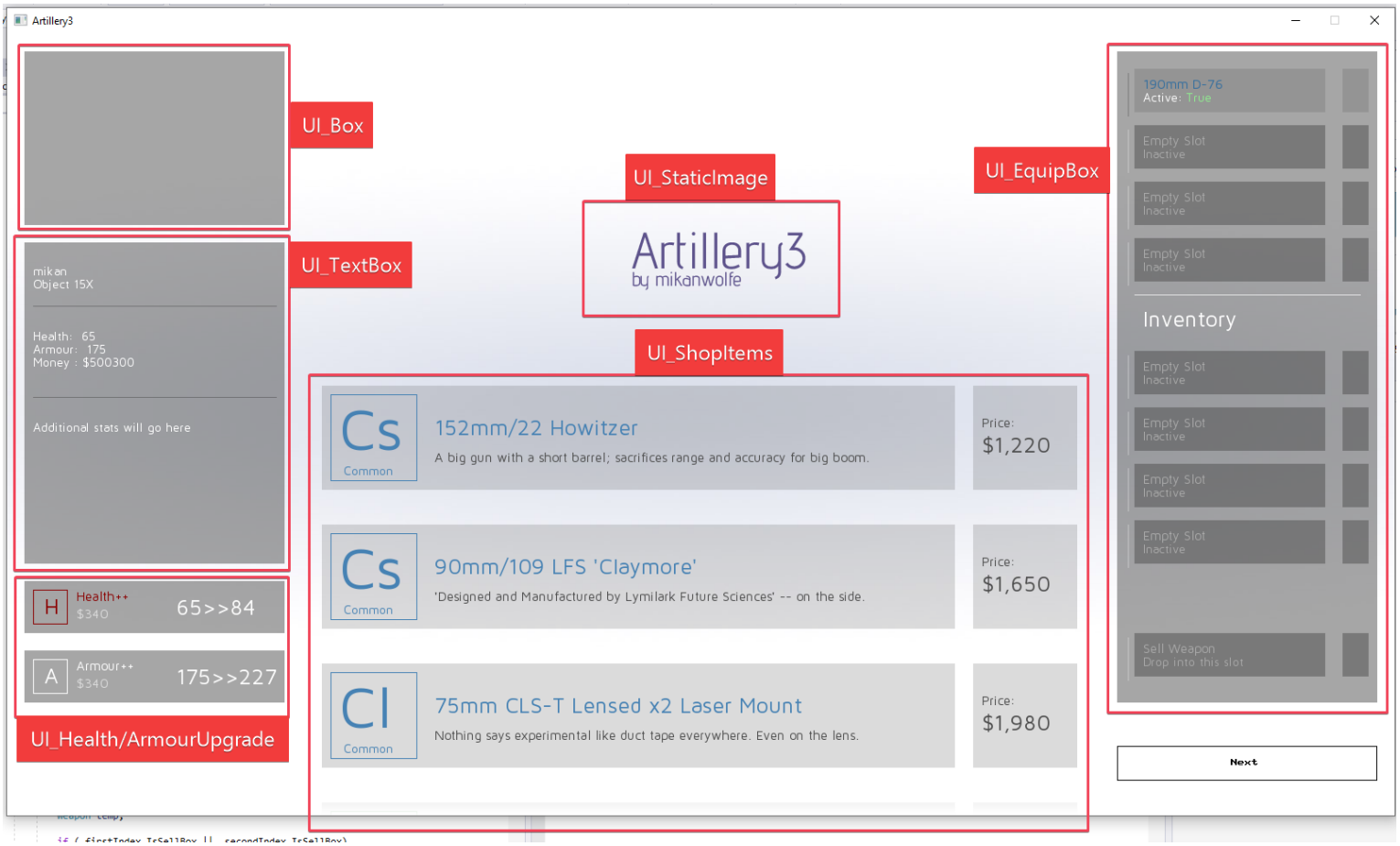
This also means that any other UI Element can be constructed and fit within the Main Menu such that the Main Menu Game State is not coupled to the UI Main Menu, only UI Element Assemblies.

1.3.4. The Shop Game State

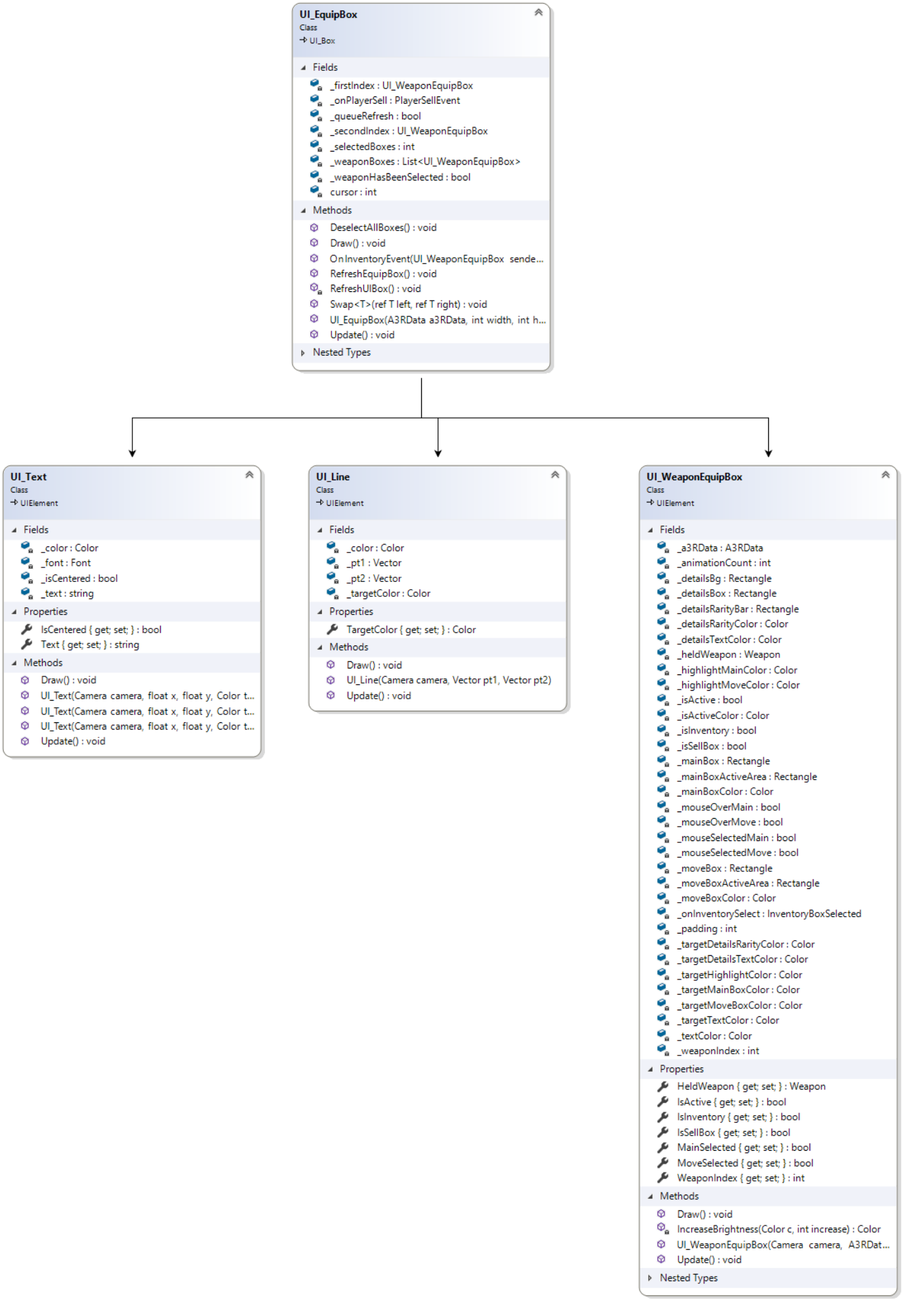
A high-level overview of the shop is below, showing the major components used in the Shop Game State.

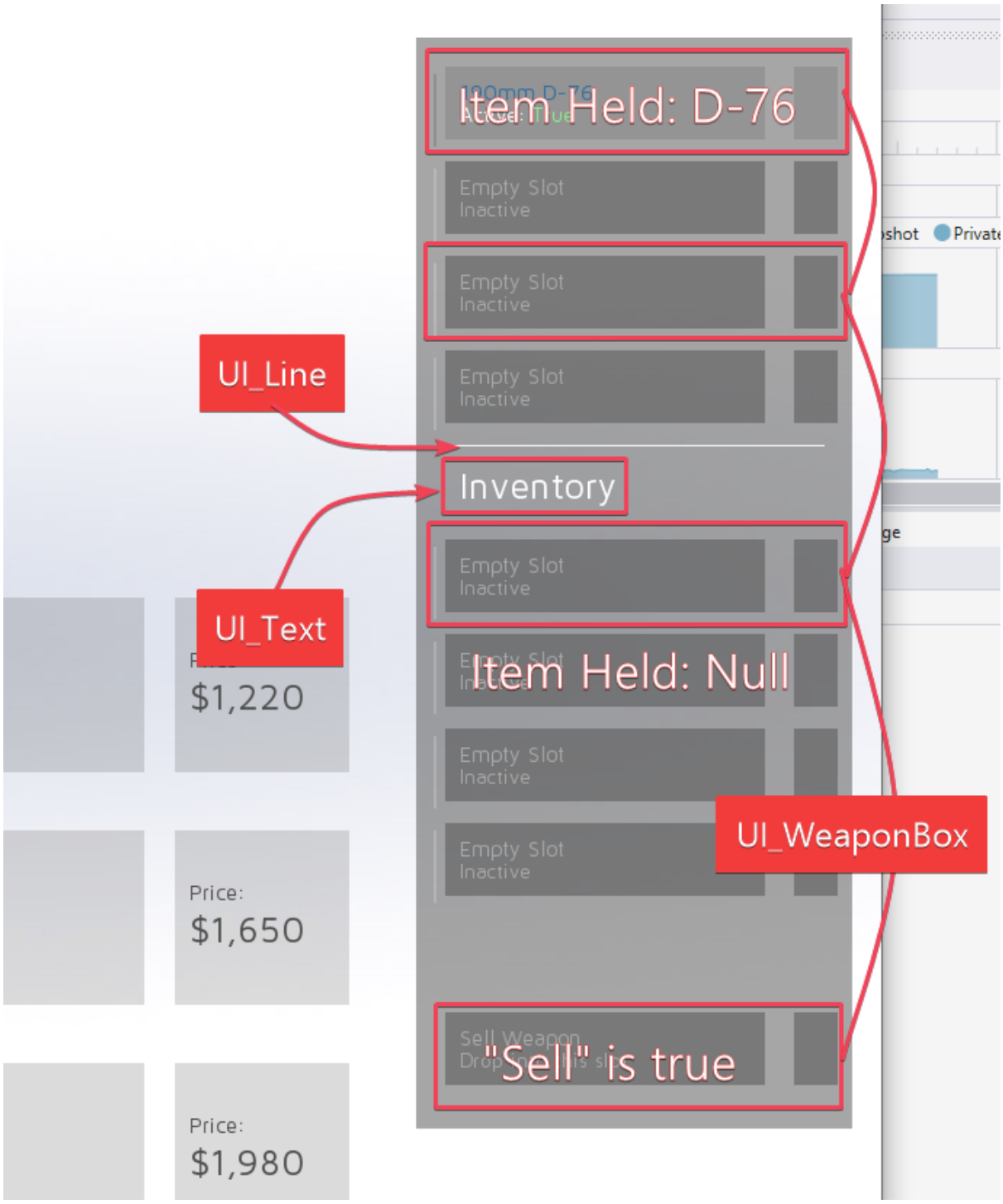


The screenshot of the game below shows the general layout of the high-level overview:



Each of the Sub-Assemblies are also UI Element Assemblies and therefore, follow a similar design to the high-level overview. The next page shows the UI Equip Box's UML Diagram, with the following diagram showing the general layout of each element used.


UI_EquipBox
Class
UI_Box
Fields
_firstIndex : UI_WeaponEquipBox
_onPlayerSell : PlayerSellEvent
_queueRefresh : bool
_secondIndex : UI_WeaponEquipBox
_selectedBoxes : int
_weaponBoxes : List<UI_WeaponEquipBox>
_weaponHasBeenSelected : bool
cursor : int
Methods
DeselectAllBoxes() : void
Draw() : void
OnInventoryEvent(UI_WeaponEquipBox sende...
RefreshEquipBox() : void
RefreshUIBox() : void
Swap<T>(ref T left, ref T right) : void
UI_EquipBox(A3RData a3RData, int width, int h...
Update() : void
Nested Types
UI_Text
Class
UIElement
Fields
_color : Color
_font : Font
_isCentered : bool
_text : string
Properties
IsCentered { get; set; } : bool
Text { get; set; } : string
Methods
Draw() : void
UI_Text(Camera camera, float x, float y, Color t...
UI_Text(Camera camera, float x, float y, Color t...
UI_Text(Camera camera, float x, float y, Color t...
Update() : void
UI_Line
Class
UIElement
Fields
_color : Color
_pt1 : Vector
_pt2 : Vector
_targetColor : Color
Properties
TargetColor { get; set; } : Color
Methods
Draw() : void
UI_Line(Camera camera, Vector pt1, Vector pt2)
Update() : void
UI_WeaponEquipBox
Class
UIElement
Fields
_a3RData : A3RData
_animationCount : int
_detailsBg : Rectangle
_detailsBox : Rectangle
_detailsRarityBar : Rectangle
_detailsRarityColor : Color
_detailsTextColor : Color
_heldWeapon : Weapon
_highlightMainColor : Color
_highlightMoveColor : Color
_isActive : bool
_isActiveColor : Color
_isInventory : bool
_isSellBox : bool
_mainBox : Rectangle
_mainBoxActiveArea : Rectangle
_mainBoxColor : Color
_mouseOverMain : bool
_mouseOverMove : bool
_mouseSelectedMain : bool
_mouseSelectedMove : bool
_moveBox : Rectangle
_moveBoxActiveArea : Rectangle
_moveBoxColor : Color
_onInventorySelect : InventoryBoxSelected
_padding : int
_targetDetailsRarityColor : Color
_targetDetailsTextColor : Color
_targetHighlightColor : Color
_targetMainBoxColor : Color
_targetMoveBoxColor : Color
_targetTextColor : Color
_textColor : Color
_weaponIndex : int
Properties
HeldWeapon { get; set; } : Weapon
IsActive { get; set; } : bool
IsInventory { get; set; } : bool
IsSellBox { get; set; } : bool
MainSelected { get; set; } : bool
MoveSelected { get; set; } : bool
WeaponIndex { get; set; } : int
Methods
Draw() : void
IncreaseBrightness(Color c, int increase) : Color
UI_WeaponEquipBox(Camera camera, A3RDat...
Update() : void
Nested Types



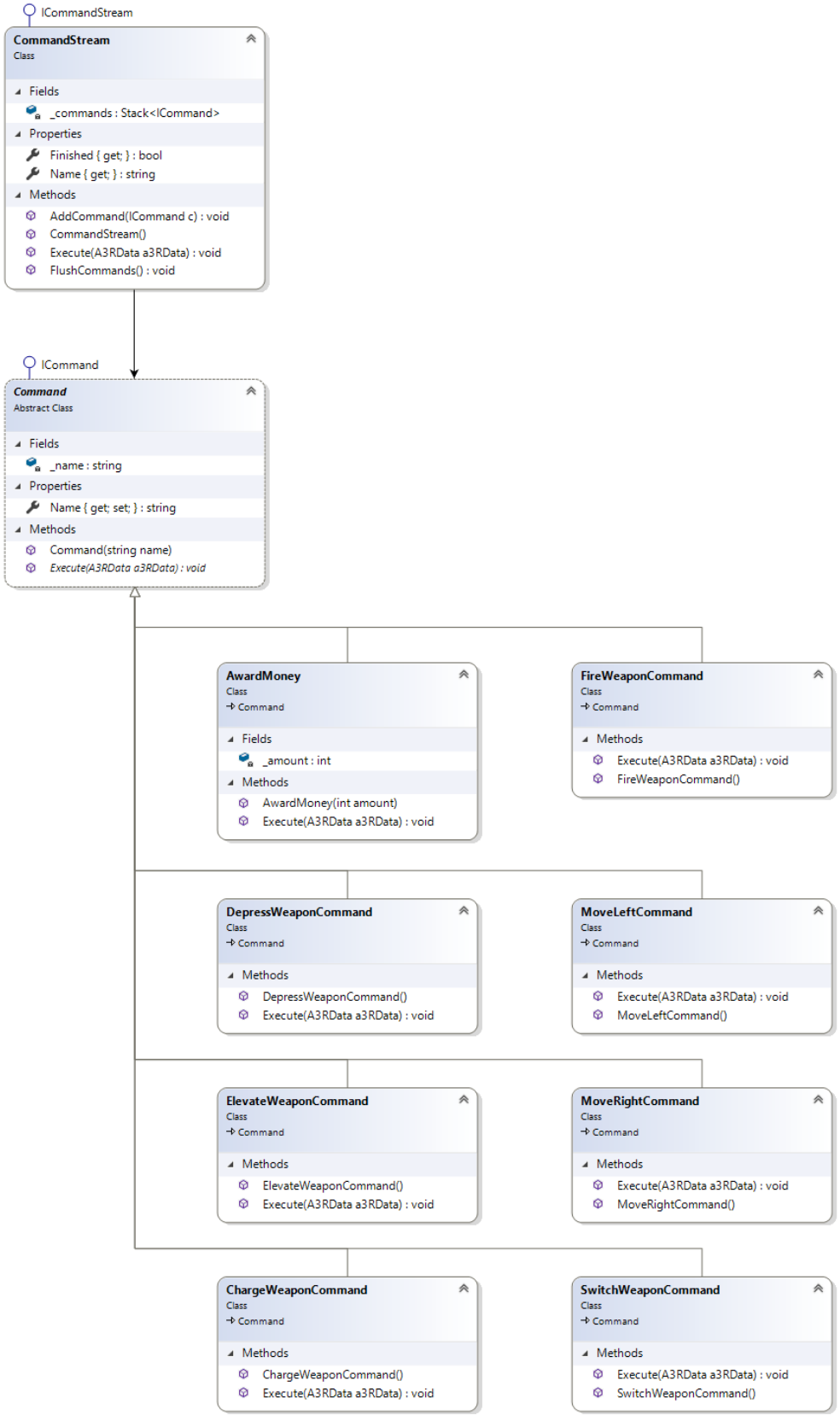
Further refining of this system would be very useful to reduce coupling, however, as it stands it is a functional system that is sufficiently separated to allow reuse.

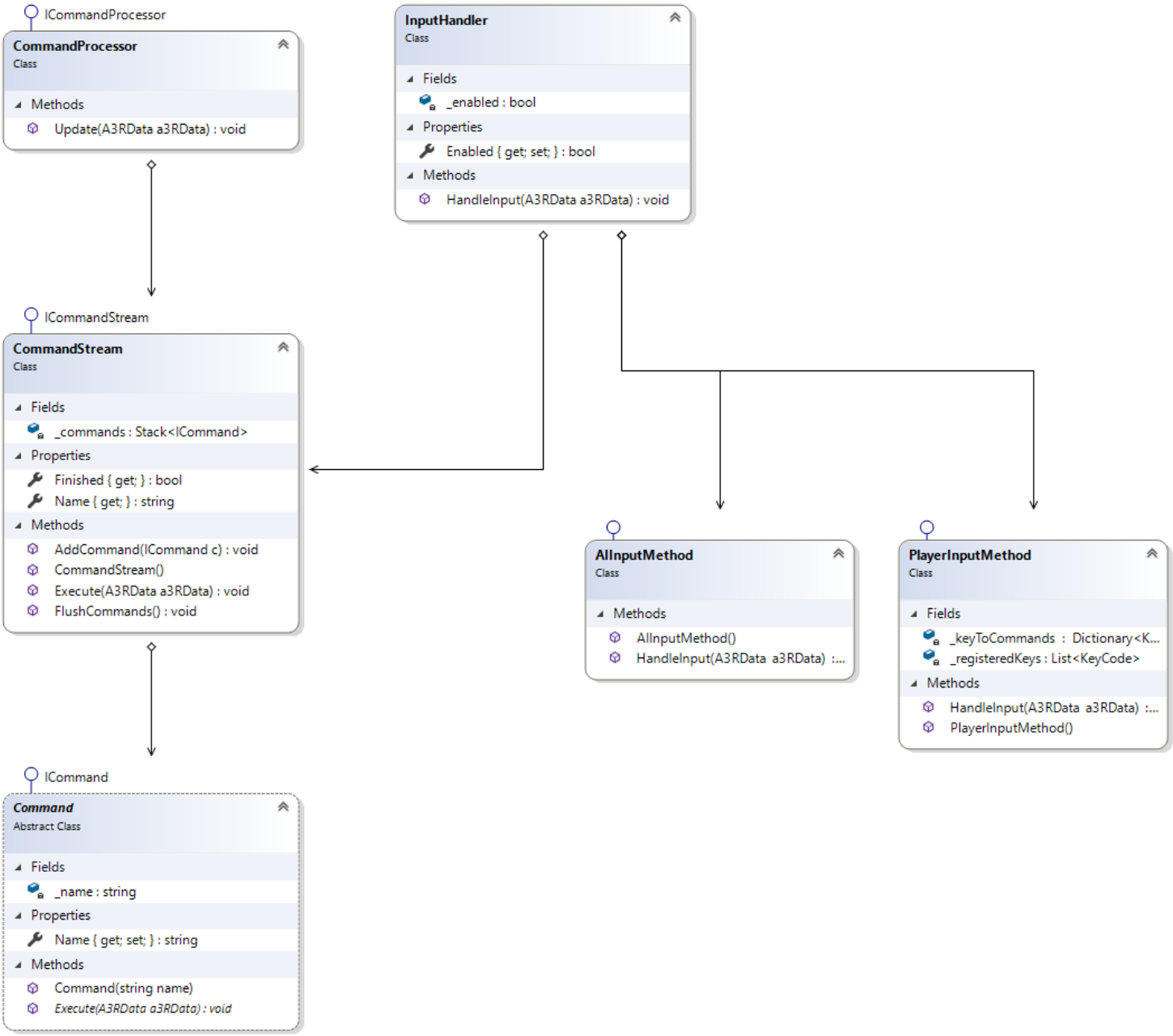
The appearance of the box changes depending on the item held, however, the inversion of dependency can be applied to create a top-level abstraction which inherits from another class, one for each state like the use of Game State as a class—this is all possible, if it weren't for time constraints and the sheer number of already-existing abstractions.

1.4 Design Patterns

This section shows off several major components with Artillery3: Command and Command Streams, and high-level use of inheritance and interfaces.

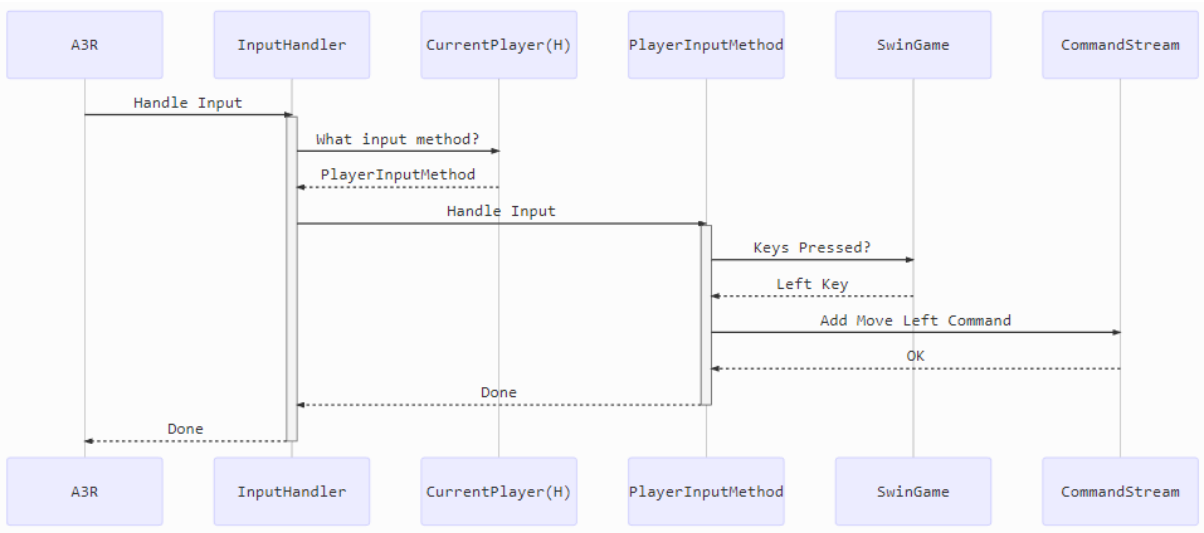
1.4.1 Command and Command Streams

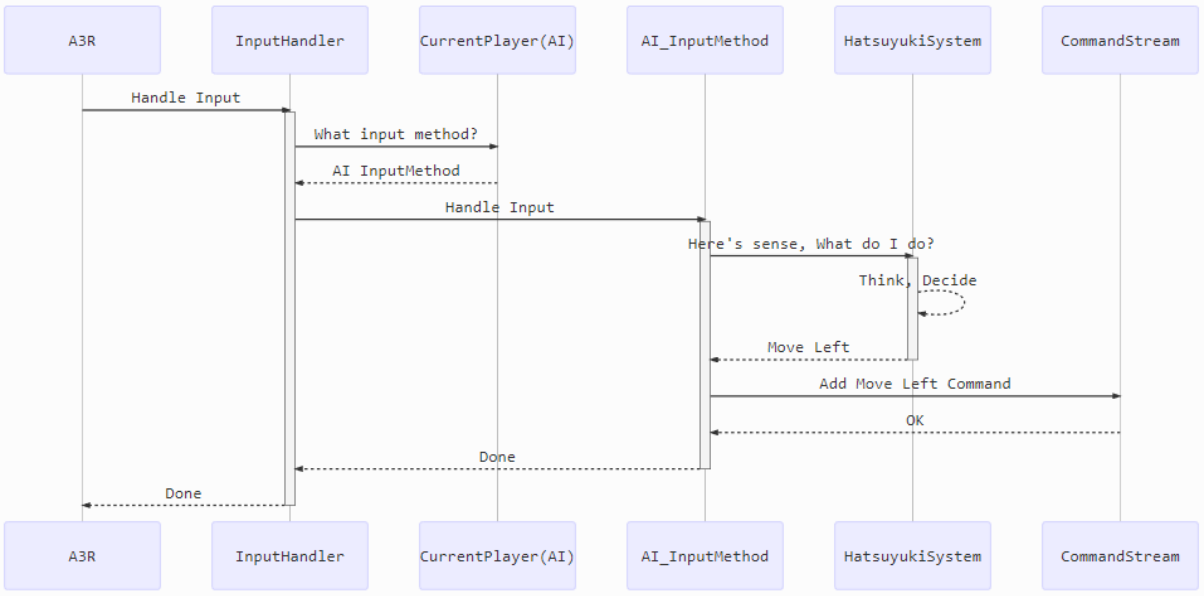




The Input Method is held by each player (not character!) and determines the method of input for the character. During the Handle Input method, the Input Handler will query the current player for their input method and then call that input method accordingly to return commands to the input stream.

The following two Sequence Diagrams show how the Input Handler interacts with both a human and AI player. Note that the AI is not present within A3s, at current, the AI Input Method simply returns a “Move Right” command, however, the system is built such that the sequence is possible.



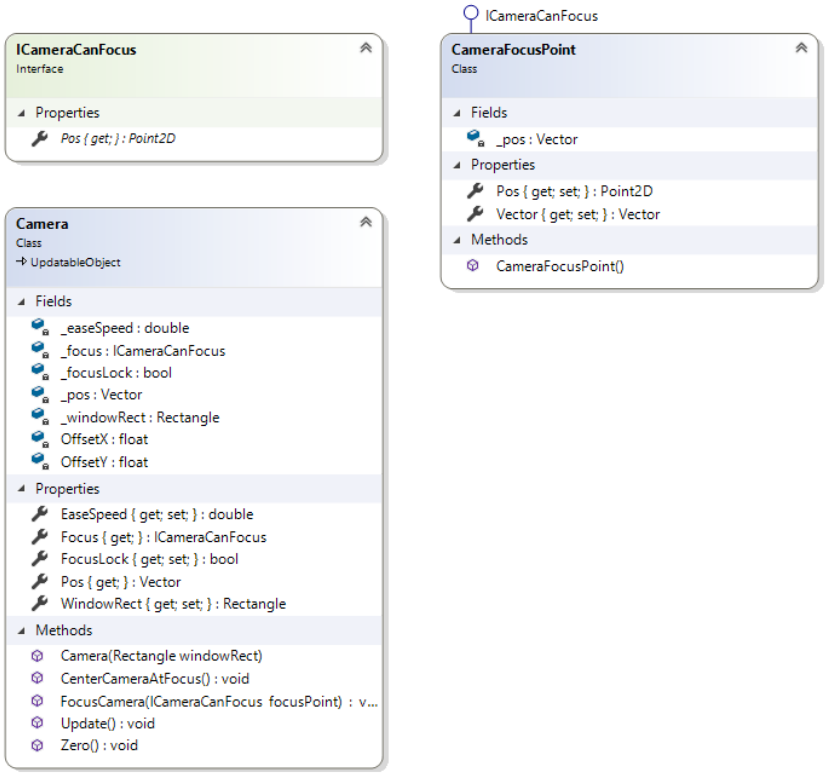


The Command Processor simply executes all the commands from the command stream once the input has been recorded. This decouples the input handler from executing commands, and therefore, the command processor has no knowledge of what object wanted the commands to be executed.

It should also be noted that the implementation of a console command line would also work within this domain as a separate service that is able to inject commands into the command stream would be able to manipulate the game as another input method. For example, injecting an "award" command in the command stream will **always** award the player the specified amount of money—the same would go for any teleport commands. It's a command line, it works like most command lines.

1.4.2. A good example of interfaces and Inheritance

One of the more 'fun' aspects of A3s would be the ability to move the camera by holding down the right mouse button and effectively 'pan'. This is done through the camera and the encapsulation of the camera using a Camera class. The camera cannot be directly controlled – you can only specify a point of focus for the camera and the camera will ease into place.



Anything that inherits the Camera Can Focus interface is required to have a Position. When the camera is told to focus on an object, it follows the object. As the camera is not actively being told all the time to focus on certain position, the camera can do its own thing most of the time.

This means that, by adding in a right-click command, we can have the camera focus on the mouse by creating a new Focus Position if the mouse is held down. When the camera is then told to view the next entity, it simply switches focus and goes back to doing as it was intended to do. All in all, a very simple but neat solution to a very cool feature.

1.5. Further Implementations/Recommendations

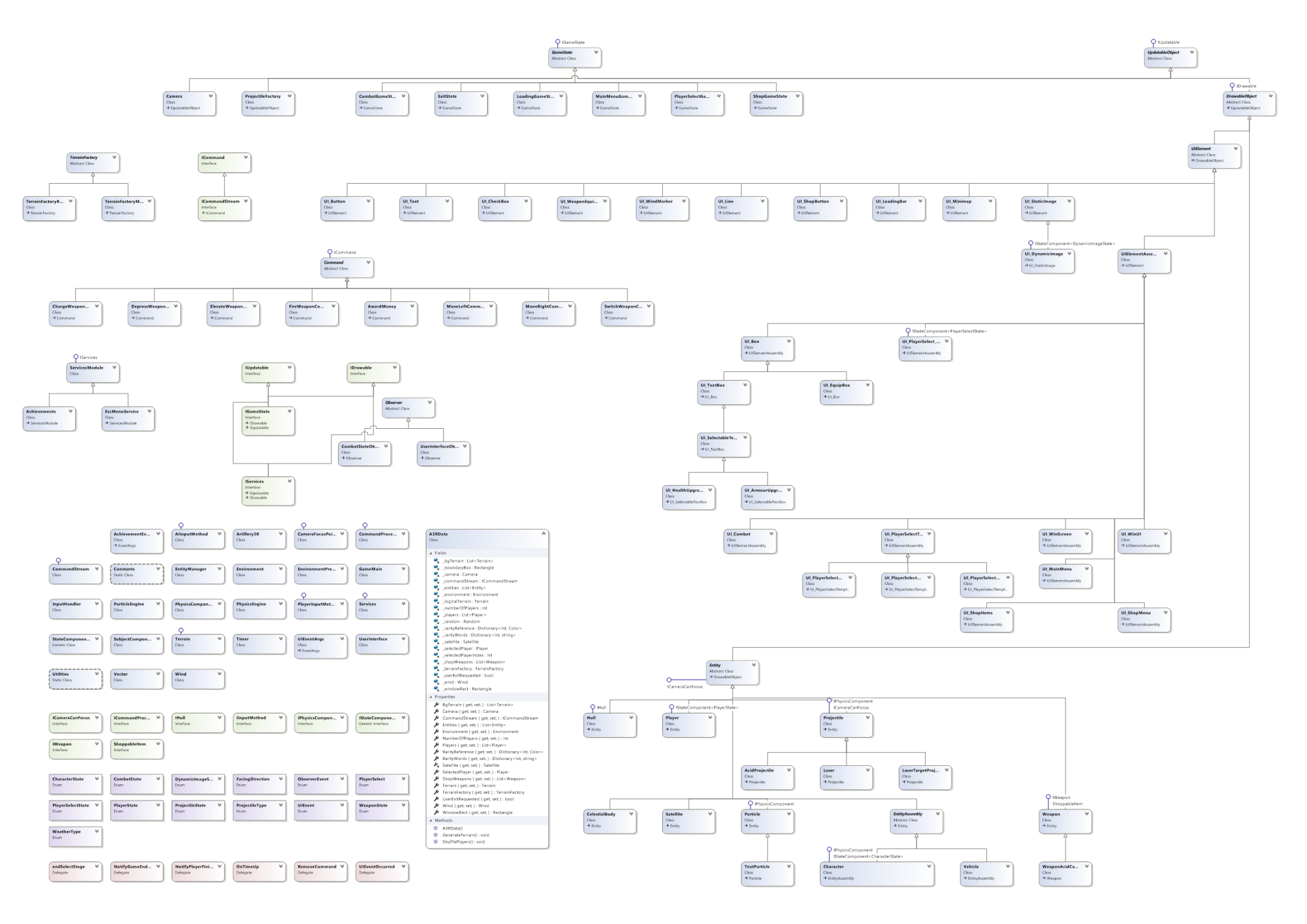
There are several things that could be implemented to improve the general usability or OO-ness of Artillery, notably A3s. They are, in no particular order:

- Using an abstract factory for UI elements and have that factory as a singleton within the Services module.
    - This reduces dependency on the exact UI elements involved, (dependency inversion) and allows aspects such as themed UI elements by creating specific UI elements that all under the abstraction.
    - For example, a "dark" factory produces UI_DarkButton whereas a "light" factory would produce a UI_LightButton, both of which inherit from button and override the appearance.
- Further breaking down and better implementation of UI modules

    - Simply put, due to time constraints, a lot of UI elements are lacking in polish and proper modularisation and encapsulation.
- Proper research and implementation into the use of XML versus JSON, and the ability to Serialise the game data into JSON as a method of saving.
- The ability to alter the terrain generation aspects within Artillery3, notably, by keeping the Abstract Terrain Factory within the model, the game could allow the user, through an UI interface, to alter aspects such as the individual roughness of each terrain or the colour of the terrain.
- Proper serialisation and deserialization of game information.
    - As it stands, I had a "do it right or don't do it at all" attitude towards saving character information within Artillery3. It would have been possible and straightforward to save the information from each character as in Flexible Drawing 5.3D, however, that would have been a rather brash method and short-sighted.
- In following with Artillery2, possibly Effects that would alter the playing field: beams that came down from the sky which would change projectiles going through them, such as a mirror.
- A strategy pattern implementation for projectile flight, which would have been utilised for the Missile weapon type.
- Proper use and utilisation of the Command Pattern such that the player could open a command console and type in command names to inject commands into the command stream for the command processor to handle.
- Networking would be interesting, though it wouldn't be particularly useful.

2. Conclusions

Honestly this document is long enough. Artillery3 is more complex than this document entails. If it were all written down it'd take another 20 or so pages. "Why not include a full class diagram?" – See the next page. Whether or not it's legible is another story. Generated directly from Visual Studio.


IGameState
GameState
Abstract Class
IUpdatable
UpdatableObject
Abstract Class
IDrawable
DrawableObject
Abstract Class
UpdatableObject
Camera
ProjectileFactory
CombatGameSt...
ExitState
LoadingGameSt...
MainMenuGam...
PlayerSelectGa...
ShopGameState
TerrainFactory
Abstract Class
ICommand
Interface
UIElement
Abstract Class
DrawableObject
TerrainFactoryR...
TerrainFactoryM...
ICommandStream
UI_Button
UI_Text
UI_CheckBox
UI_WeaponEqui...
UI_WindMarker
UI_Line
UI_ShopButton
UI_LoadingBar
UI_Minimap
UI_StaticImage
IStateComponent<DynamicImageState>
UI_DynamicImage
UIElementAsse...
ICommand
Command
Abstract Class
ChargeWeapon...
DepressWeapon...
ElevateWeapon...
FireWeaponCo...
AwardMoney
MoveLeftComm...
MoveRightCom...
SwitchWeaponC...
IServices
ServicesModule
Achievements
EscMenuService
IUpdatable
Interface
IDrawable
Interface
IGameState
Interface
Observer
Abstract Class
CombatStateOb...
UserInterfaceOb...
IServices
Interface
IStateComponent<PlayerSelectState>
UI_Box
UI_PlayerSelect_...
UI_TextBox
UI_EquipBox
UI_SelectableTe...
UI_HealthUpgra...
UI_ArmourUpgr...
UI_Combat
UI_PlayerSelectT...
UI_WinScreen
UI_WinUI
UI_PlayerSelect...
UI_PlayerSelect...
UI_PlayerSelect...
UI_MainMenu
UI_ShopItems
UI_ShopMenu
AchievementEv...
AIInputMethod
Artillery3R
CameraFocusPoi...
CommandProce...
CommandStream
Constants
Static Class
EntityManager
Environment
EnvironmentPre...
GameMain
InputHandler
ParticleEngine
PhysicsCompon...
PhysicsEngine
PlayerInputMet...
Services
StateComponen...
Generic Class
SubjectCompon...
Terrain
Timer
UIEventArgs
UserInterface
Utilities
Static Class
Vector
Wind
ICameraCanFocus
Interface
ICommandProc...
IHull
IInputMethod
IPhysicsCompon...
IStateCompone...
Generic Interface
IWeapon
ShoppableItem
CharacterState
Enum
CombatState
DynamicImageS...
FacingDirection
ObserverEvent
PlayerSelect
PlayerSelectState
PlayerState
ProjectileState
ProjectileType
UIEvent
WeaponState
WeatherType
endSelectStage
Delegate
NotifyGameEnd...
NotifyPlayerFini...
OnTimeUp
RemoveCommand
UIEventOccurred
A3RData
Class
Fields
_bgTerrain : List<Terrain>
_boundaryBox : Rectangle
_camera : Camera
_commandStream : ICommandStream
_entities : List<Entity>
_environment : Environment
_logicalTerrain : Terrain
_numberOfPlayers : int
_players : List<Player>
_random : Random
_rarityReference : Dictionary<int, Color>
_rarityWords : Dictionary<int, string>
_satellite : Satellite
_selectedPlayer : Player
_selectedPlayerIndex : int
_shopWeapons : List<Weapon>
_terrainFactory : TerrainFactory
_userExitRequested : bool
_wind : Wind
_windowRect : Rectangle
Properties
BgTerrain { get; set; } : List<Terrain>
Camera { get; set; } : Camera
CommandStream { get; set; } : ICommandStream
Entities { get; set; } : List<Entity>
Environment { get; set; } : Environment
NumberOfPlayers { get; set; } : int
Players { get; set; } : List<Player>
RarityReference { get; set; } : Dictionary<int, Color>
RarityWords { get; set; } : Dictionary<int, string>
Satellite { get; set; } : Satellite
SelectedPlayer { get; set; } : Player
ShopWeapons { get; set; } : List<Weapon>
Terrain { get; set; } : Terrain
TerrainFactory { get; set; } : TerrainFactory
UserExitRequested { get; set; } : bool
Wind { get; set; } : Wind
WindowRect { get; set; } : Rectangle
Methods
A3RData()
GenerateTerrain() : void
ShuffflePlayers() : void
Entity
Abstract Class
ICameraCanFocus
IHull
IStateComponent<PlayerState>
IPhysicsComponent
ICameraCanFocus
Hull
Player
Projectile
AcidProjectile
Laser
LaserTargetProj...
CelestialBody
Satellite
IPhysicsComponent
Particle
EntityAssembly
Abstract Class
IWeapon
ShoppableItem
Weapon
TextParticle
IPhysicsComponent
IStateComponent<CharacterState>
Character
Vehicle
WeaponAcidCa...