# Technical Documentation

## I. Justification of our choices:

Our team has chosen C# and Unity for the development of our game.

Firstly, C# is a modern, object-oriented programming language that is both versatile and flexible. It allows for a great deal of modularity in code, which makes it easier to maintain and extend. This is particularly important for a project like ours, where we expect to add and modify features over the given time to complete the project.

In addition, C# and Unity both have large and active communities of developers, which was particularly helpful when troubleshooting our game as problems arose often in the production period of our game.

Finally, Unity is known for its high performance and scalability. It can handle complex, high-fidelity graphics and physics simulations. C# also, being a statically typed language, offers better performance than dynamically typed languages.

In conclusion, Unity and C# provided a powerful, flexible, and user-friendly platform for our project development. We believe it was the best choice for our project and we hope you will like the result of our endeavors.

## II. Data structures:

For our project's data structure, we chose to use Scriptable Objects and "Player Prefs" as they are two powerful features in Unity that can be used to store data in two separate ways.

Scriptable Objects are a type of Unity object that can be saved as an asset in the project. They allowed us to create custom data types that can be easily edited and managed in the Unity editor. This was particularly useful for things like our brains (which we talk about in more detail below), the stats of the units, turrets, and castles, music and more, where we want to be able to easily adjust the values for different game objects.

Player Prefs, on the other hand, are a simple and easy-to-use system for storing and retrieving player preferences or game data. It uses a key-value pair system, where we can store a value (such as a high score or a player's name) under a specific key, and then retrieve that value later using the same key. In our project, they were used to store the player’s chosen volume and mute state of music and sound and the chosen language.

In addition to the two of them, we also created classes like Stat and StatModifier to hold data of our units, castles, and turrets statistics, all of them can be found in our scriptable objects.

In conclusion, Scriptable Objects and Player Prefs are two powerful features in Unity that were used, in our project, to create a flexible and easy to maintain basics data structure.

## III. AI behaviors:

### Overview

The functionality of the AIs is based on Game Events within the game, managed by an interface called "IEventFoudations". Another class, named "IAThinker", encapsulates all the logic for the AI actions.

For instance, functions such as Spawns, Launch Capacity, etc., are implemented to call the Game Events from IEventFoudations. These functions handle all the necessary checks for XP/Gold, update data like the AI's current age and the number of turrets, and access game data and all possible AI actions through scriptable objects.

All these functions operate similarly, and here is an example using the spawn function: These functions are of Boolean type.

The spawn function takes the type of unit as a parameter (a SerializeField scriptable object instantiated in the class and then drag-and-dropped into Unity). It first checks if it is possible to spawn a unit (by verifying that the AI's gold is greater than the unit's cost). If the check passes, it spawns the unit and subtracts the unit's cost from the AI's gold. If all these actions are performed correctly, it returns "true". If the actions cannot be completed, it returns "false".

This Boolean logic allows for control over whether the actions have been performed by the AI brains and if it has the capability to execute them. Additionally, for certain functions, there is a Boolean parameter "buff" which allows for the refunding of the action cost or making the actions free, thus increasing the AI's difficulty level.

In the AI brains, a counter system has been implemented to prevent spamming certain actions and to prioritize others. For instance, the age increase counter must be equal to 20 to trigger the age increase function, allowing other actions to be performed in the meantime. Once the action is completed, this counter resets to 0.

Every half second, the counters are incremented, and the AI gains gold and XP. The amount of XP increases based on the difficulty level. Whenever an action is performed, the counters reset to 0.

**AI 1**: This AI is completely random and chooses all its actions randomly. It has a small counter to reduce the frequency of upgrades that were happening too often.

**AI 2:** This AI is defense oriented. It calls the spawn function only when the player takes actions, prioritizing defense actions such as buying and upgrading turrets or purchasing defense upgrades. It prioritizes abilities based on the counter but will advance in age after a certain number of ticks.

**AI 3:** This AI focuses on offense, spawning units continuously. When the player places a certain number of units, the AI uses abilities. The AI prioritizes spawning units and attacking the player. All these actions are managed based on specific conditions (e.g., the number of enemies on the field, the number of allies) and counters. Some actions start to get buffed to increase the game's difficulty.

**AI 4:** This AI follows a counter strategy, spawning units that counter the player's units and focusing on tanks. When a player places a unit, the AI counters by placing a unit that counters the player's unit. When the player does nothing, the AI places one melee and two ranged units. When the player places a certain number of units, the AI uses abilities. All actions are buffed except for turrets, upgrades, and age advancement. The AI gains money every time it spawns a unit, and the amount of XP and gold it gains every half second is high, allowing it to advance in age much faster. Consequently, it will have higher-level units than the player. This AI is much harder to beat due to its counter strategy, tank focus, and higher gold/XP gain. Upgrades are chosen randomly.

## IV. Game design choices:

For our game design choices, we chose to take the basics of the “Age of War” game. Units, castles, and turrets statistics are the same with some modification. The only difference is the fact we chose to add a small amount of gold given to the player to make the game more interesting and easier to play.

Here is a recap of all statistics of the game:

## Upgrades:

| Upgrade type | Number of upgrades | Level 1 Cost (Gold) | Level 1 Stat change | Level 2 Cost (Gold) | Level 2 Stat change | Level 3 Cost (Gold) | Level 3 Stat change |
|---|---|---|---|---|---|---|---|
| Support Dammage | 3 | 300 | +5 | 1500 | +5 | 3500 | +5 |
| Infantry Dammage | 3 | 300 | +5 | 1700 | +5 | 3500 | +5 |
| Anti-Armor Dammage | 3 | 400 | +5 | 800 | +5 | 2600 | +5 |
| Armor Dammage | 3 | 300 | +5 | 1700 | +5 | 3500 | +5 |
| Turrets Dammage | 3 | 300 | +5 | 1700 | +5 | 3500 | +5 |
| Income Level | 3 | 500 | +10 | 1100 | +10 | 2000 | +10 |
| Support Range | 3 | 300 | +1 | 700 | +1 | 1500 | +1 |
| Infantry Health | 3 | 300 | +2 | 1700 | +2 | 3500 | +2 |
| Anti-Armor Health | 3 | 300 | +2 | 1700 | +2 | 3500 | +2 |
| Armored Health | 3 | 300 | +2 | 1700 | +2 | 3500 | +2 |
| Turrets Range | 3 | 300 | +1 | 1700 | +1 | 3500 | +1 |

## Units Age 1:

| Unit | Support | Infantry | Anti-Armor | Armored |
|---|---|---|---|---|
| Cost (Gold) | 100 | 130 | 200 | 600 |
| Hp | 100 | 140 | 145 | 150 |
| Dammage | 15 | 15 | 30 | 15 |
| Hit Speed | 5sec | 6sec | 3 sec | 1 sec |
| Build Time (s) | 1sec | 1sec | 1.5 sec | 2 sec |
| Type | Support | Infantry | Anti-Armor | Armored |
| Walk Speed (m/s) | 2 | 2 | 3 m/sec | 1 m/sec |
| Range | 7 | 1 | 1 | 1 |
| Gold Given on Death | 80 | 75 | 70 | 90 |
| XP given on death | 350 | 250 | 320 | 350 |
| Counter | Anti-Armor | Support | Armored | Infantry |

## Units Age 2:

| Unit | Support | Infantry | Anti-Armor | Armored |
|---|---|---|---|---|
| Cost (Gold) | 125 | 140 | 210 | 600 |
| Hp | 250 | 340 | 220 | 250 |

| | | | | |
|---|---|---|---|---|
| **Dammage** | 20 | 20 | 35 | 20 |
| **Hit Speed** | 5sec | 6sec | 3 sec | 1 sec |
| **Build Time (s)** | 1sec | 1sec | 1.5 sec | 2 sec |
| **Type** | Support | Infantry | Anti-Armor | Armored |
| **Walk Speed (m/s)** | 2 | 2 | 3 m/sec | 1 m/sec |
| **Range** | 8 | 1 | 1 | 1 |
| **Gold given on death** | 90 | 95 | 80 | 100 |
| **XP given on death** | 365 | 265 | 335 | 365 |
| **Counter** | Anti-Armor | Support | Armored | Infantry |

## Units Age 3:

| Unit | Support | Infantry | Anti-Armor | Armored |
|---|---|---|---|---|
| **Cost (Gold)** | 150 | 150 | 220 | 620 |
| **Hp** | 400 | 540 | 295 | 350 |
| **Dammage** | 25 | 25 | 40 | 25 |
| **Hit Speed** | 5sec | 6sec | 3 sec | 1 sec |
| **Build Time (s)** | 1sec | 2sec | 1.5 sec | 2 sec |
| **Type** | Support | Infantry | Anti-Armor | Armored |
| **Walk Speed (m/s)** | 3 | 3 | 4 m/sec | 2 m/sec |
| **Range** | 9 | 1 | 1 | 1 |
| **Gold given on death** | 100 | 105 | 90 | 110 |
| **XP given on death** | 380 | 280 | 350 | 390 |
| **Counter** | Anti-Armor | Support | Armored | Infantry |

## Units Age 4:

| Unit | Support | Infantry | Anti-Armor | Armored |
|---|---|---|---|---|
| **Cost (Gold)** | 175 | 160 | 230 | 640 |
| **Hp** | 550 | 740 | 370 | 450 |
| **Dammage** | 30 | 30 | 45 | 30 |
| **Hit Speed** | 5sec | 6sec | 3 sec | 1 sec |
| **Build Time (s)** | 1sec | 1sec | 1.5 sec | 2 sec |
| **Type** | Support | Infantry | Anti-Armor | Armored |
| **Walk Speed (m/s)** | 3 | 3 | 4 m/sec | 2 m/sec |
| **Range** | 10 | 1 | 1 | 1 |
| **Gold given on death** | 110 | 115 | 100 | 120 |
| **XP given on death** | 395 | 295 | 365 | 405 |
| **Counter** | Anti-Armor | Support | Armored | Infantry |

Units Age 5:

| Unit | Support | Infantry | Anti-Armor | Armored |
|---|---|---|---|---|
| Cost (Gold) | 200 | 170 | 240 | 660 |
| Hp | 700 | 940 | 445 | 550 |
| Dammage | 40 | 40 | 50 | 35 |
| Hit Speed | 1sec | 6sec | 3 sec | 1 sec |
| Build Time (s) | 1 | 1sec | 1.5 sec | 2 sec |
| Type | Support | Infantry | Anti-Armor | Armored |
| Walk Speed (m/s) | 4 | 4 | 5 m/sec | 3 m/sec |
| Range | 11 | 1 | 1 | 1 |
| Gold given on death | 120 | 125 | 110 | 130 |
| XP given on death | 410 | 310 | 380 | 420 |
| Counter | Anti-Armor | Support | Armored | Infantry |

Units Age 6:

| Unit | Support | Infantry | Anti-Armor | Armored |
|---|---|---|---|---|
| Cost (Gold) | 225 | 180 | 250 | 680 |
| Hp | 850 | 1140 | 520 | 650 |
| Dammage | 50 | 50 | 55 | 40 |
| Hit Speed | 2sec | 6sec | 3 sec | 1 sec |
| Build Time (s) | 1 | 2sec | 1.5 sec | 7 sec |
| Type | Support | Infantry | Anti-Armor | Armored |
| Walk Speed (m/s) | 4 | 4 | 5 m/sec | 3 m/sec |
| Range | 12 | 1 | 1 | 1 |
| Gold given on death | 130 | 135 | 120 | 140 |
| XP given on death | 425 | 325 | 395 | 415 |
| Counter | Anti-Armor | Support | Armored | Infantry |

Units Age 7:

| Unit | Support | Infantry | Anti-Armor | Armored |
|---|---|---|---|---|
| Cost (Gold) | 250 | 190 | 260 | 700 |
| Hp | 1000 | 1340 | 595 | 750 |
| Dammage | 65 | 65 | 60 | 45 |
| Hit Speed | 2sec | 6sec | 3 sec | 1 sec |
| Build Time (s) | 2 | 2sec | 1.5 sec | 12 sec |
| Type | Support | Infantry | Anti-Armor | Armored |
| Walk Speed (m/s) | 5 | 5 | 6 m/sec | 4 m/sec |
| Range | 13 | 1 | 1 | 1 |
| Gold given on death | 140 | 145 | 130 | 150 |

| | | | | |
|---|---|---|---|---|
| **XP given on death** | 440 | 340 | 410 | 430 |
| **Counter** | Anti-Armor | Support | Armored | Infantry |

## V. Physics handling:

Raycasting is a fundamental technique used in our game for detecting units at both long and medium distances. It involves projecting a straight line (or ray) from a specified origin in a particular direction and determining if it intersects with any objects in the game world.

**Long-distance Raycasting**

For long-distance detection, we utilize raycasting to scan vast areas of the game world. This allows units to detect distant objects such as enemies or castles, enabling strategic decision-making and AI behaviors.

**Medium-distance Raycasting**

In addition to long-distance detection, raycasting is also employed for medium-range interactions within the game environment. This includes detecting nearby units, castles or allies that may affect gameplay dynamics.

## VI. Design pattern choices:

### 1. Strategy Pattern

#### Overview

The Strategy Pattern is a behavioral design pattern that enables an object to select a strategy or an algorithm at runtime. In our implementation, we have used this pattern to create a modular and flexible system for game objects to "Think".

#### Classes

#### *Brain Class*

The `Brain` class is the Strategy interface, defining a common method `Think()` that all concrete strategies (specific Brain implementations) should implement. This class is abstract and is meant to be extended by concrete Brain classes.

```
public abstract class Brain : ScriptableObject
{
    public abstract void Think(Thinker thinker);
}
```

#### *Thinker Class*

The Thinker class is the Context class, which maintains a reference to a Strategy object (Brain in this case). The Context class is responsible for executing the strategy, which it does by calling the Think() method on the Brain object.

```
public class Thinker : MonoBehaviour
{
    [SerializeField] private Brain brain;

    private void FixedUpdate()
    {
        brain.Think(this);
    }
}
```

### Usage

To use this system, you can create a new Brain class that extends the Brain interface and implements the Think() method. Then, you can attach the Thinker script to a game object and assign the Brain object to the Thinker's brain field. The Thinker will then call the Think() method on the Brain object every frame.

This system allows for a great deal of flexibility and modularity. You can create different Brain classes, each with their own implementation of the Think() method, and the Thinker class can use any of these Brain classes without needing to know the specifics of their implementation. This makes it easy to switch between different strategies or algorithms at runtime, and it also makes the code easier to maintain and extend.

### Conclusion

The Strategy Pattern is a powerful tool for creating modular and flexible systems, and our implementation of this pattern for game objects to "Think" is a great example of its utility. We hope that this system will be useful to you in your own projects.

## 2. Observer Pattern

### Overview

The Observer Pattern is a behavioral design pattern that allows an object to notify other objects when its state changes. In our implementation, we have used this pattern to create a system for game objects to listen to and respond to events.

## Classes

### *GameEvent Class*

The `GameEvent` class is the Subject interface, maintaining a list of all `GameEventListener` objects that subscribe to the event. The Subject interface provides methods for adding and removing observers (`GameEventListener` in this case), and for notifying all observers of a state change, which it does by calling the `OnEventRaised()` method on each `GameEventListener` object.

```
public class GameEvent : ScriptableObject, IEvent
{
    private List<GameEventListener> _gameEventListeners;

    private void Awake()
    {
        _gameEventListeners = new List<GameEventListener>();
    }

    public void Raise(Component sender, object data)
    {
        foreach (var gameEventListener in _gameEventListeners)
        {
            gameEventListener.OnEventRaised(sender, data);
        }
    }

    public void RegisterListener(GameEventListener listener)
    {
        if (!_gameEventListeners.Contains(listener))
        {
            _gameEventListeners.Add(listener);
        }
    }

    public void UnregisterListener(GameEventListener listener)
    {
        if (_gameEventListeners.Contains(listener))
        {
            _gameEventListeners.Remove(listener);
        }
    }
}
```

### *GameEventListener Class*

The `GameEventListener` class is the Observer interface, defining a method `OnEventRaised()` that will be called when the event is triggered. The Observer interface also allows the observer to register and unregister itself from the subject.

```
public class GameEventListener : MonoBehaviour
{
    [SerializeField] private GameEvent gameEvent;

    public CustomUnityEvent response;

    private void Start()
    {
        gameEvent.RegisterListener(this);
    }

    private void OnDisable()
    {
        gameEvent.UnregisterListener(this);
    }

    public void OnEventRaised(Component sender, object data)
    {
        response.Invoke(sender, data);
    }
}
```

### *CustomUnityEvent Class*

The `CustomUnityEvent` class is a simple extension of the built-in `UnityEvent` class, allowing it to accept two parameters: one of type `Component` and one of type `object`.

```
[Serializable]
public class CustomUnityEvent : UnityEvent<Component, object>{ }
```

## Usage

To use this system, you can create a new `GameEvent` object and a new `GameEventListener` object. Then, you can attach the `GameEventListener` script to a game object and assign the `GameEvent` object to the `GameEventListener`'s `gameEvent` field. The `GameEventListener` will then listen to the `GameEvent` and respond to it by calling the `OnEventRaised()` method.

This system allows for much flexibility and modularity. You can create different `GameEvent` objects, each representing a different event, and different `GameEventListener` objects, each with their own implementation of the `OnEventRaised()` method. This makes it easy to add and remove event listeners at runtime, and it also makes the code easier to maintain and extend.