Mega-Eater

Technical Design Document (TDD)

Most of the document is described by the following TDD example:
https://dlorenzolaguno17.github.io/TDD/

ToC

1. Core sections

1.1 GDD Overview

Mega-Eater is developed for Gram Lergrav – a paleontological museum in Jutland. The goal is to present the museum's collection in an engaging way.

In the game, you take control of a megalodon roaming the seas of Jutland 10 million years ago. Your objective is to catch as much prey as possible within a limited amount of time. Each successful hunt earns you points and provides a little information about the different sea creatures. When the timer runs out, you'll be presented with a loading screen with a fun fact about one of the sea creatures in the game. Finally, you can enter your name on the leaderboard to showcase your score.

1.2 Scope

Initially, the game will be kept simple with a controllable protagonist and animated prey. The game will also feature a soundtrack as well as audio and visual effects. Additional features may be added at a later stage.

1.3 Technical Goal

The game must be playable directly in a web browser on an iPad without the need for extra installations or third-party software. This also means the game must be accessible via Safari, as it's the standard software on an iPad. Therefore, the game should be optimized for touch controls rather than mouse and keyboard.

The game must be able to run stably at a minimum of 60 FPS (frames per second) on standard iPad hardware. The system must be able to handle at least 20 simultaneous objects, such as fish, sharks, and environmental objects, without significant performance loss.

1.4 Technical Risks

A significant risk is that the game will not perform optimally in a browser, as it's more difficult to optimize. Additionally, there's a risk that the game may work differently, for better or worse, across various browsers. Some effects and advanced graphics modules built into many engines have difficulty functioning optimally via a browser, so we'll be somewhat limited in our choice of effects and animation modules.

2. Code Style Guidelines

2.1 Naming Rules

2.1.1 Constants

In C#, constants are named with **UPPERCASE_LETTERS**, with words separated by underscores. The `const` keyword is used for constant values, while `readonly` can be used for values that are set once and not changed afterward.

```
const string MY_CONSTANT = "constant value";
public static readonly string MY_GLOBAL_CONSTANT = "another constant value";
```

2.1.2 Non-constant variables

Non-constant variables should be written in **camelCase**. This makes it easier to distinguish them from constants, classes, and interfaces.

```
string myVariable = "variable value";
```

2.1.3 Public Properties

For classes and global properties, we must use **PascalCase** as it helps differentiate between constant and non-constant variables.

```
public string MyGlobalVariable = "global variable value";
```

2.1.4 Unused variables

_ is used to indicate variables that are syntactically required but will not be used. This is often seen in **lambda expressions** or **event handlers** where there's no need to access the parameter. It makes the code more readable and clearly signals that the value is being ignored.

```
public void PrintValues(Dictionary<int, string> map)
{
    foreach (var (_, value) in map)
    {
        Console.WriteLine(value);
    }
}
```

2.2 Variables

2.2.1 Protected, Private and Public

When working in c# and unity, it's important to know when and how to use the right access modifier.

Public:
Used for whenever a property needs to be callable outside of a class, for example whenever separate scripts or user interfaces needs to read their direct value such as "health"

Private:

Used for whenever a property shouldn't be called outside of its class example, a speed property we don't want other scripts to change.

Protected:

Used primarily for inheritance. A protected variable or method cannot be accessed by everyone, but it *can* be accessed within the class itself and by its subclasses. This is especially useful when creating base classes with common functionality that should be shared with child classes but not exposed publicly.

When working with access modifiers in classes, it's a good idea to start off by making everything private and only open the properties/methods necessary.

2.2.2 Enums vs Booleans

When working with a bunch of Booleans at the same time such as isWalking and isRunning it's much preferred to use Enums instead, as that makes the code cleaner and the entire codebase just overall more readable.

BAD

```
bool isWalking = false;
bool isRunning = false;
```

GOOD

```
public enum MovementState
{
    Unknown = 0,
    Walking = 1,
    Running = 2
}

MovementState movementState = MovementState.Unknown;
```

2.3 Loops

2.3.1 Which loops to use

When working with c# and unity in general there's a few rules whenever it comes to which loops you should choose.

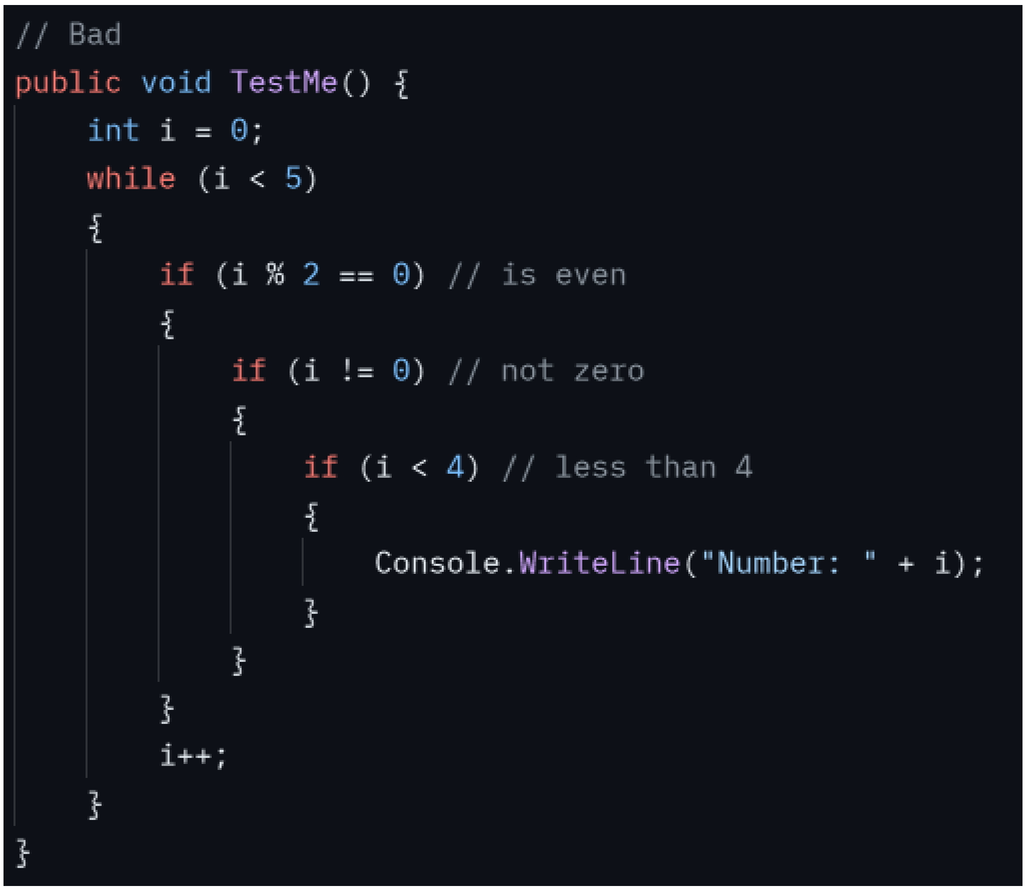
For loops are generally preferred since it gives you full control of indexing and it's a lot safer when changing/modifying elements during iteration.

Foreach can be used for readability whenever you only need to read data and not change elements during iteration.

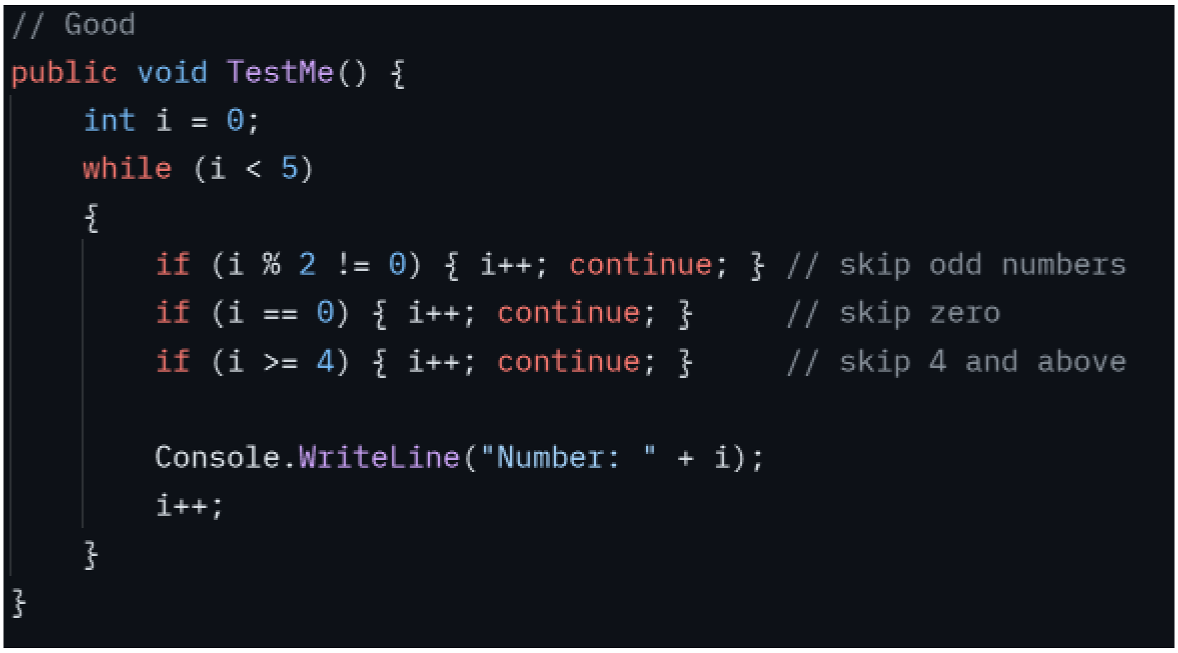
Rest of the loops like **while, do-while** etc. Can be used whenever necessary.

2.3.2 Continue

Whenever we need to skip over an iteration during looping, we should use the continue keyword. This allows us to clearly express that certain cases are not relevant, without adding multiple layers of nested if statements. By doing so, the main logic of the loop stays easy to read and maintain, which improves overall code clarity and reduces the chance of mistakes.



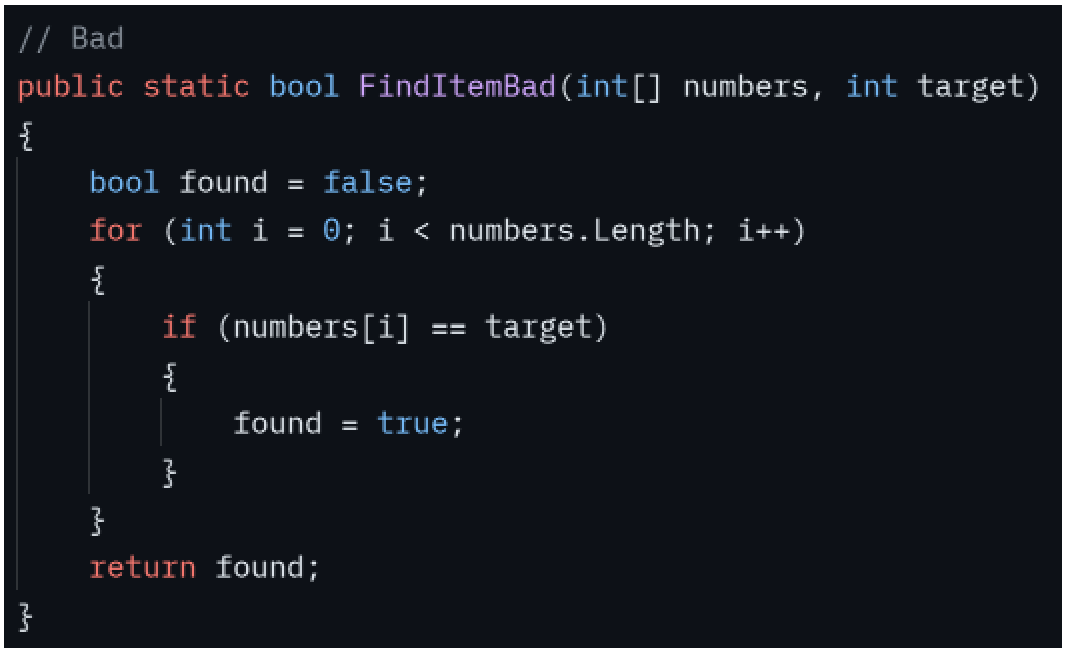
```
// Bad
public void TestMe() {
    int i = 0;
    while (i < 5)
    {
        if (i % 2 == 0) // is even
        {
            if (i != 0) // not zero
            {
                if (i < 4) // less than 4
                {
                    Console.WriteLine("Number: " + i);
                }
            }
        }
        i++;
    }
}
```



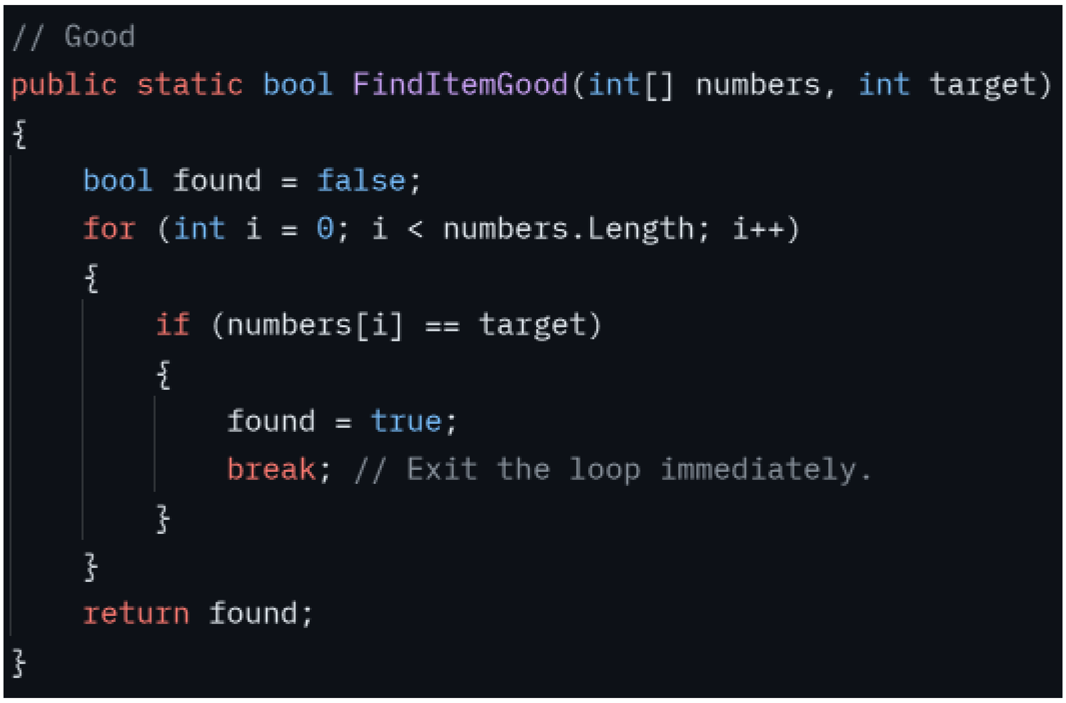
```
// Good
public void TestMe() {
    int i = 0;
    while (i < 5)
    {
        if (i % 2 != 0) { i++; continue; } // skip odd numbers
        if (i == 0) { i++; continue; }      // skip zero
        if (i >= 4) { i++; continue; }      // skip 4 and above

        Console.WriteLine("Number: " + i);
        i++;
    }
}
```

2.3.3 Break

Use break to provide an early, clean exit from a loop. A well-placed break can significantly improve both the performance and readability of our code. By exiting as soon as the desired condition is met, you prevent unnecessary iterations, reducing computational overhead and making the code's intent clearer to anyone who reads it.



```
// Bad
public static bool FindItemBad(int[] numbers, int target)
{
    bool found = false;
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] == target)
        {
            found = true;
        }
    }
    return found;
}
```

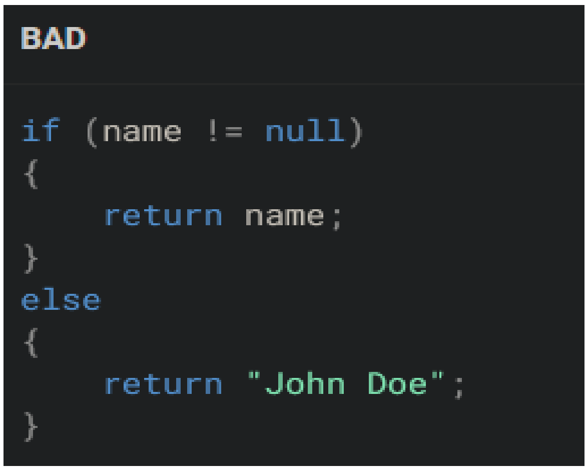


```
// Good
public static bool FindItemGood(int[] numbers, int target)
{
    bool found = false;
    for (int i = 0; i < numbers.Length; i++)
    {
        if (numbers[i] == target)
        {
            found = true;
            break; // Exit the loop immediately.
        }
    }
    return found;
}
```

2.6 Conditionals

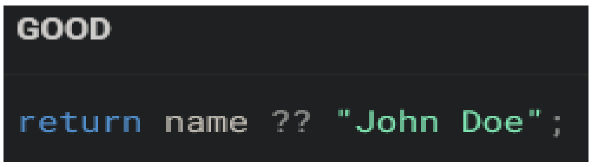
2.6.1 Ternary operators

When writing "in-variable conditionals," we prefer **ternary operators** because they make the code more readable and easier to maintain.



BAD

```
if (name != null)
{
    return name;
}
else
{
    return "John Doe";
}
```

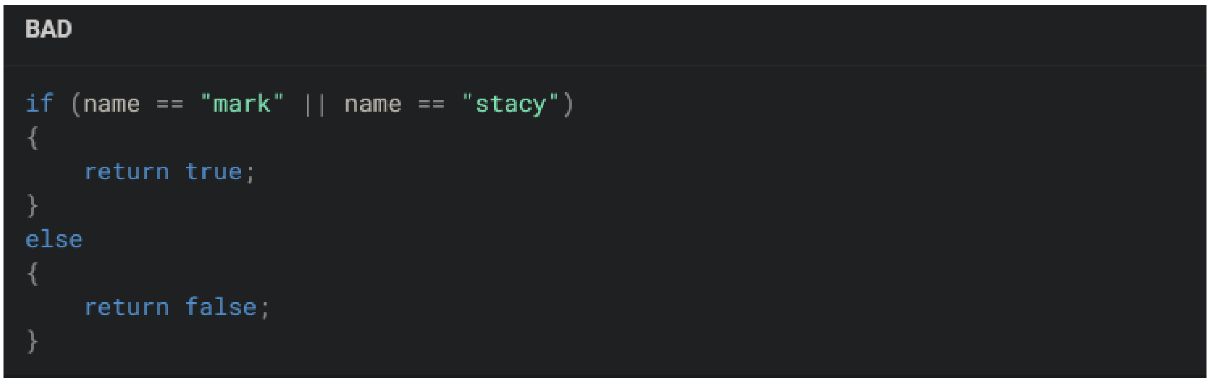


GOOD

```
return name ?? "John Doe";
```

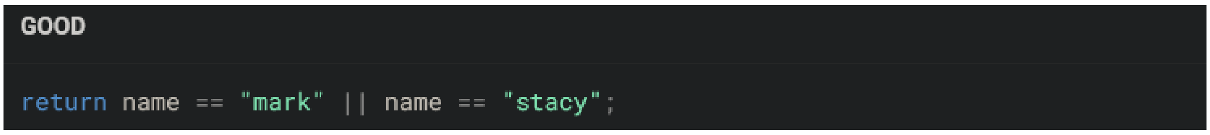
2.6.2 Dont write "if true return true"

When returning or assigning a value, avoid using "else" blocks. This degrades the code and makes it harder to read and maintain.



BAD

```
if (name == "mark" || name == "stacy")
{
    return true;
}
else
{
    return false;
}
```



GOOD

```
return name == "mark" || name == "stacy";
```

2.6.3 Prefer positive boolean expressions

This maintains code readability by always ensuring that we check for **positive Boolean expressions** first.

BAD

```
if (!isHappy)
{
    return "sad";
}
else
{
    return "happy";
}
```

GOOD

```
if (isHappy)
{
    return "happy";
}
else
{
    return "sad";
}
```

2.7 Classes & Interfaces

The class and struct types in C# have distinct uses based on their underlying behavior.

Use a class for most general-purpose types, especially for large, complex objects where an object's unique identity is important. Since a class is a reference type, when you pass it around, you are only copying a memory address, which is very efficient for large objects. Classes also support inheritance, which is a key feature for building a rich object hierarchy.

Use a struct for small, lightweight objects that are primarily used to hold data, such as a coordinate or a color. A struct is a value type, which means the entire object is copied when you pass it or assign it to a new variable. This is efficient for small types but can be a performance issue for larger ones. A general guideline is to use a struct if its instance size is 16 bytes or less and it's designed to be immutable.

2.8 Semantic Commit Messages

Vi bruger semantiske commit-beskeder for at holde vores commit-historik ren, organiseret og let at forstå. Dette hjælper os med hurtigt at identificere, hvad hvert commit gør, spore fejl og generere meningsfulde release notes.

2.8.1 Commit Message Format

A semantic commit message follows this format:

<type>(<scope>): <description>

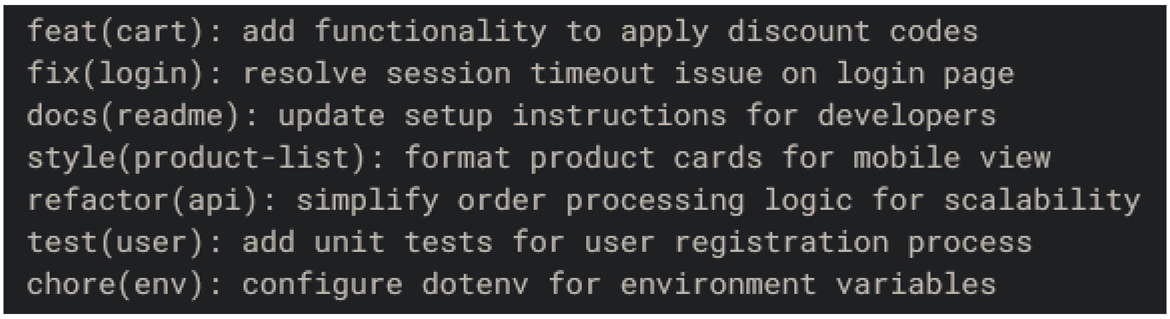
Type: The type of change being made (e.g., feat, fix, docs, style, refactor, test, chore).

Scope (optional): The scope of the change (e.g., variable name, file, module, feature)

Description: A short description of the change.

2.8.2 Examples

Here are some examples of semantic commit messages and usage.



```
feat(cart): add functionality to apply discount codes
fix(login): resolve session timeout issue on login page
docs(readme): update setup instructions for developers
style(product-list): format product cards for mobile view
refactor(api): simplify order processing logic for scalability
test(user): add unit tests for user registration process
chore(env): configure dotenv for environment variables
```

3. External Libraries

3.1 UML

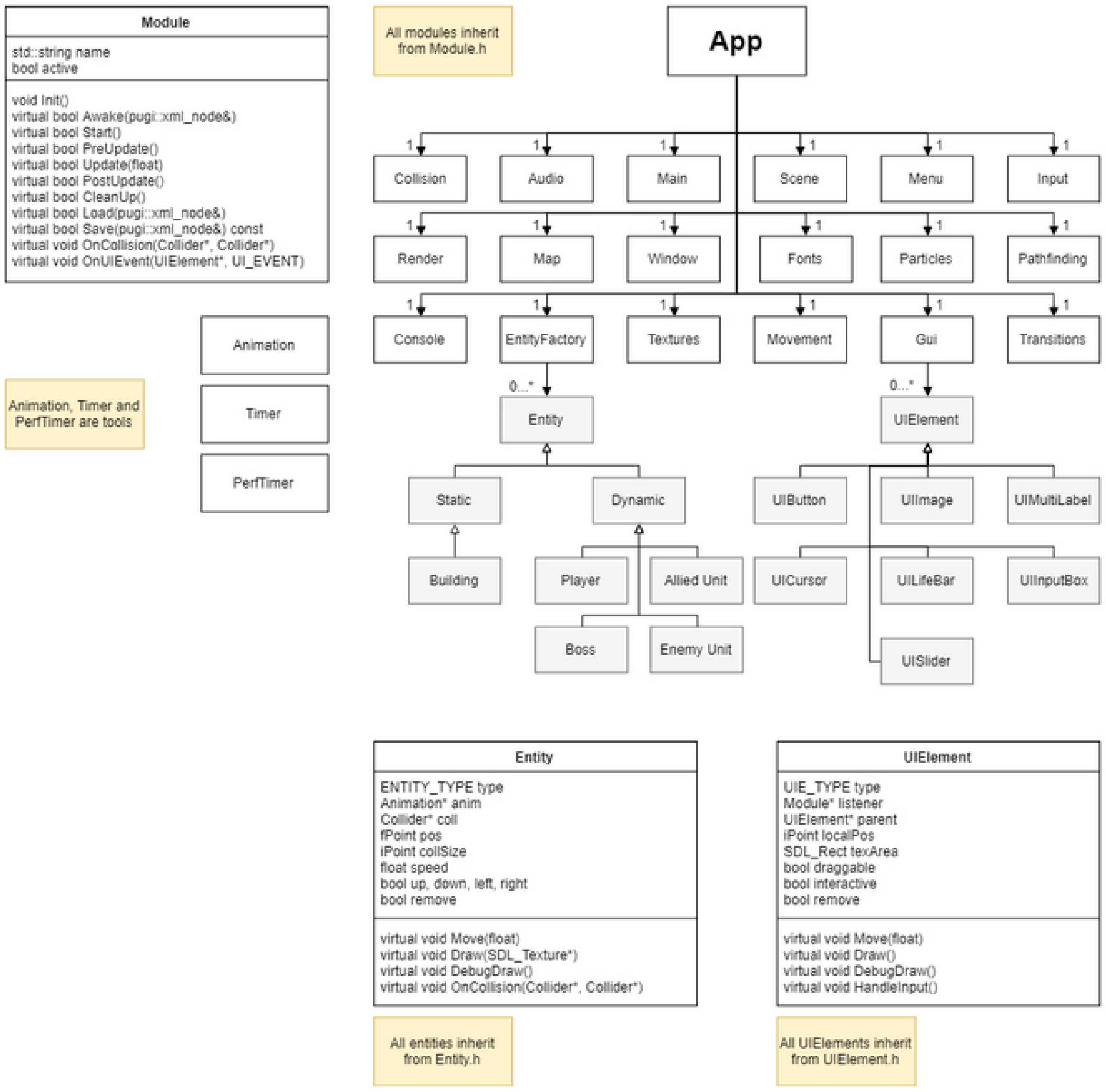
We'll use UML (Unified Modeling Language) to visually represent the design of our game. This is a crucial step in our Technical Design Document (TDD) as it provides a standardized way to model the system's structure and behavior while we develop the game.

3.1.1 Why UML?

UML diagrams will help us clarify and communicate complex ideas about the game's architecture. Instead of just describing how the game works in text, a visual model can reveal potential issues, dependencies, and relationships between different parts of the system. This makes it easier for the team to understand the overall design and ensures everyone is on the same page. By creating these diagrams early on, we can identify design flaws or inefficiencies, leading to a more robust and scalable final product.

Below is an example of how a UML diagram over a game could look like.

DevCrumb's general UML for their game [Warcraft II: The Stolen Artifacts](#).



4.2 Branching

Branches are essential for a smooth and collaborative development process. By following a clear set of rules, we can ensure that branching remains a powerful advantage rather than a source of confusion.

4.2.1 When to create a branch

A new branch should be created for one of the following reasons: Bug-fix, feature and releases. Each branch must be precise and specific to a single task, to avoid scope creeping.

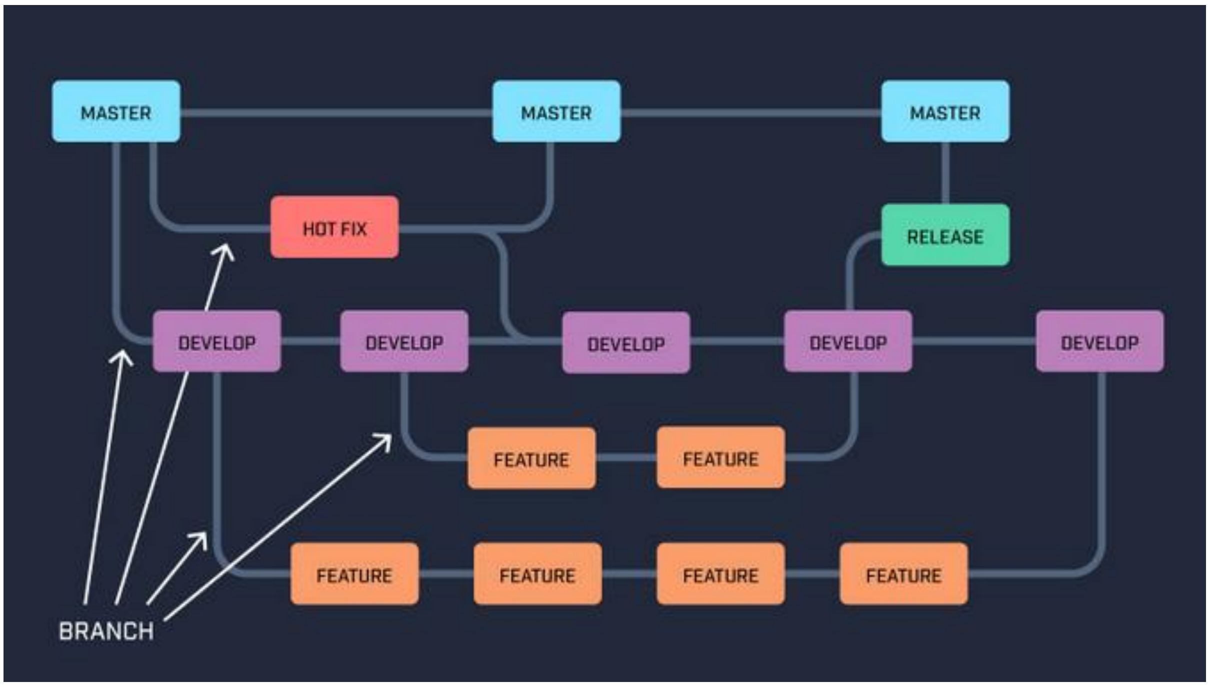
4.2.2 Which branch to use

Master: This branch is for the live and most up to date version of the game, NOTHING that is not 100% finished should ever make it to this branch post-game launch.

Develop: This is the main development branch. Here is where all features and bug fixes get merged and when deemed ready moved over to master branch.

Feature: Whenever you start working on a new feature (such as creating a character, making an enemy etc.) you create a new branch based of the develop branch, this way we preserve separation between finished and unfinished/unstable pieces of code.

Hotfix: If there's a serious/minor bug that needs addressing ASAP from the live game this branch can be created from Master and merged directly post testing



5. Building Stones

5.1 Choice of Engine

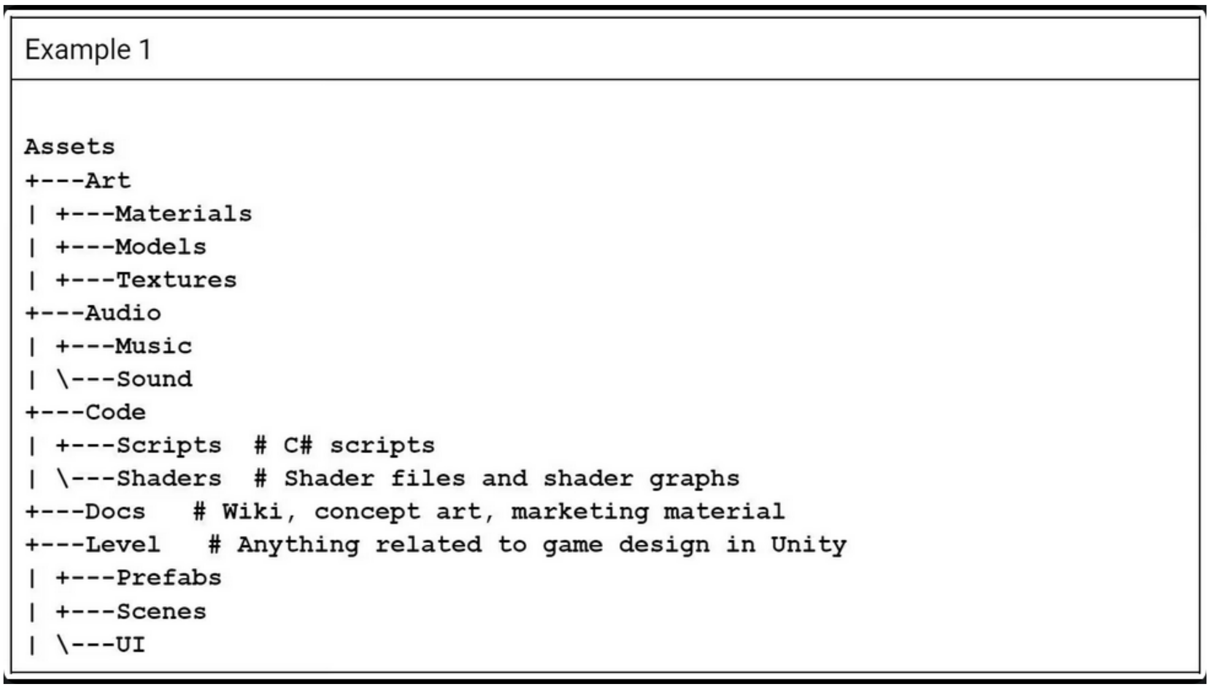
After reviewing our Product Owner requirement for the game, we chose Unity for our game engine.

This was primarily driven by two key factors, firstly Unity native support for C# as its primary scripting language which is a perfect match for our current skill sets and our matching coding standards.

Secondly, Unity's robust ability to build games that run direct in web browser was crucial enabling us to meet the PO requirements while providing a powerful framework to manage the performance.

5.2 Project Structure

For the project structure, we will be following the standard Unity games project structure as seen below.



Example 1

```
Assets
+---Art
| +---Materials
| +---Models
| +---Textures
+---Audio
| +---Music
| \---Sound
+---Code
| +---Scripts  # C# scripts
| \---Shaders  # Shader files and shader graphs
+---Docs   # Wiki, concept art, marketing material
+---Level   # Anything related to game design in Unity
| +---Prefabs
| +---Scenes
| \---UI
```

5.3 Test Driven Development

Testing in game development in general can be challenging, but certain features should be tested to ensure reliability and performance. We will focus on testing logic heavy components that can be isolated from our game engine MonoBehavior dependencies.

This can be but not limited to score calculations, fish spawning algorithms, etc.