# Salt N Sepr

**Group Members:**
Alberto Lee
Archie Godfrey
Jacqueline Eilertsen
Jake Schoonbrood
Joshua Levett
Matteo Barberis

## Architecture Report

To plan out the architecture we will have to use an abstracted view of the architecture as it is highly improbable that new classes/functions/variables will not be added later in development, it is far easier to add to the architecture structure than rewriting the core in the middle of the development process. Thus we will use a conceptual model to give an architecture framework to work off of. Not only will this model let us understand the system, it will also give everyone in the team a better understanding therefore allowing for more efficient conveyance of system details and general system development, therefore a quicker development time and a reduced risk of errors.

Use of an object-oriented architecture at face value seems the most logical because we will have many objects in the game such as “Firetruck”, “Alien”, “Fortress”, which should allow for the comprehensive creation of an object tree for our game. After some research into architecture types we found that object-oriented architectures can cause problems when they are expanded [1].

Inheritance hierarchies can become inflexible creating complications  when additions are made to our object tree. When extra features are added “exceptions” can start to appear, such as new objects being unable to fit into the existing hierarchy [2]. This means more code is moved to the main game object which can make the code long and convoluted and therefore more prone to bugs.

The inheritance hierarchies can become convoluted and lose logical sense (i.e. an object that damages might be called a “Alien”, but if we then wanted fortresses to damage then would make “Fortress” a child of Alien. Is a Fortress part of the alien class because it damages?).

To counter the problems with an object-oriented approach we can make use of components. Components act as the “properties” of an object. For example, with reference to the problem before, we can have a component which damages called “Damage”. Now we can create two individual classes “Alien” and “Fortress” and attach the Damage component to both, this way we can avoid elaborate object trees. Using a base inheritance hierarchy with components allow for less confusion allowing for better system understanding, this will also allow a third-party to have an easier time implementing new systems without needing an in depth view of the architecture.
We will favour composition over inheritance, therefore our objects will be made mostly out of components so as to avoid the creation of complex inheritance trees. The popular game engine “Unity” also makes use of components for object creation [3] which shows our approach is tried and tested.

We thought about using the architecture principle of MVC (Model-View-Controller) which is often used as a base guideline for the creation of software. After some research we found that MVC is often not used in the creation of games as the three sections that make up the principle are very closely related in game development [4]. Referring to the KISS principle, we decided not to use MVC or similar ideas due to the small scale of Kroy.

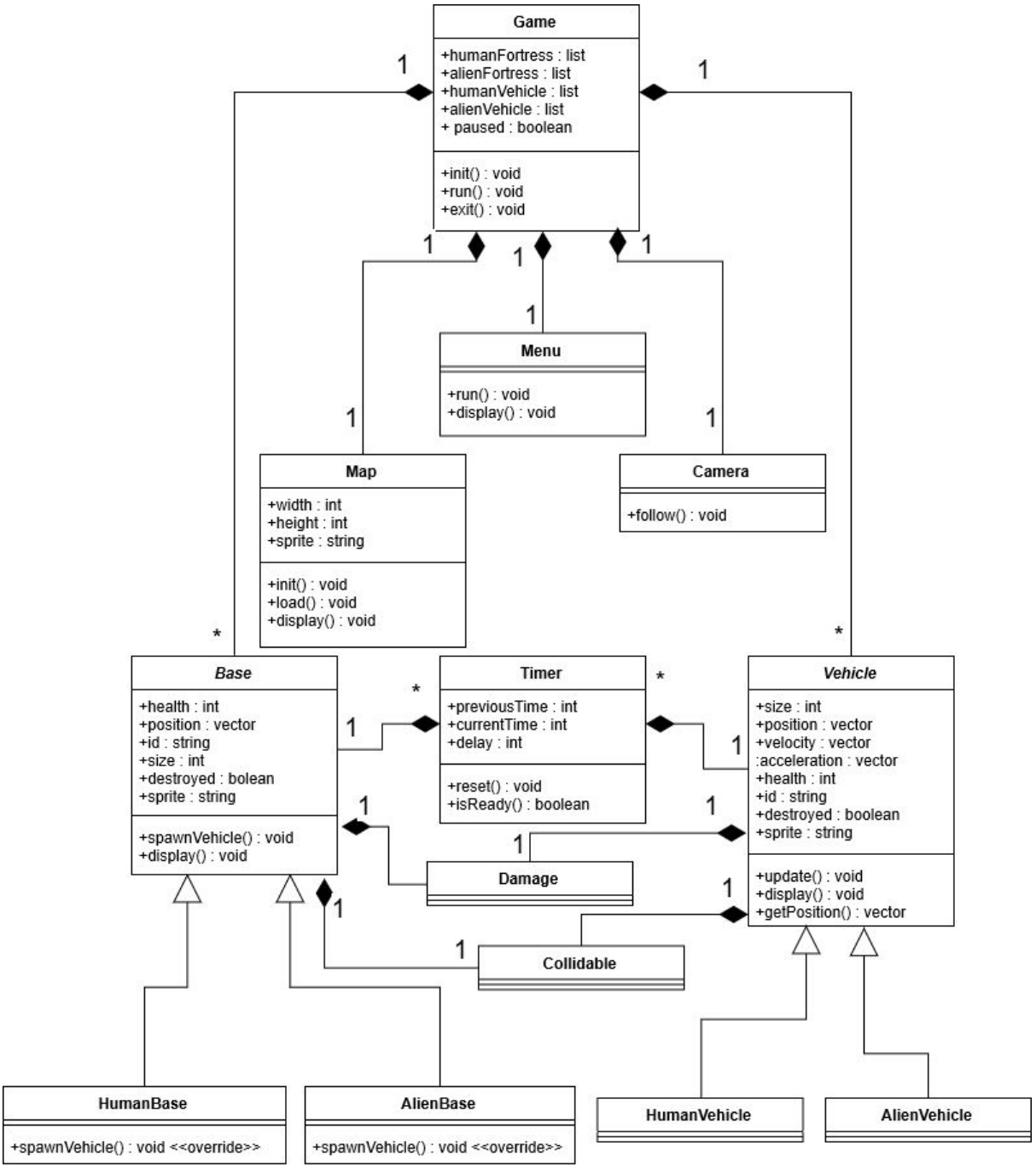


*Figure 1: UML class diagram*

UML (Unified Modelling Language) is a standardized modelling language that is often used to model software systems. We have used UML because it allows us to visualize and specify what the architecture of our system will look like in a standardized way without having to do any implementation. Moreover, the use of UML will make the development process more efficient.

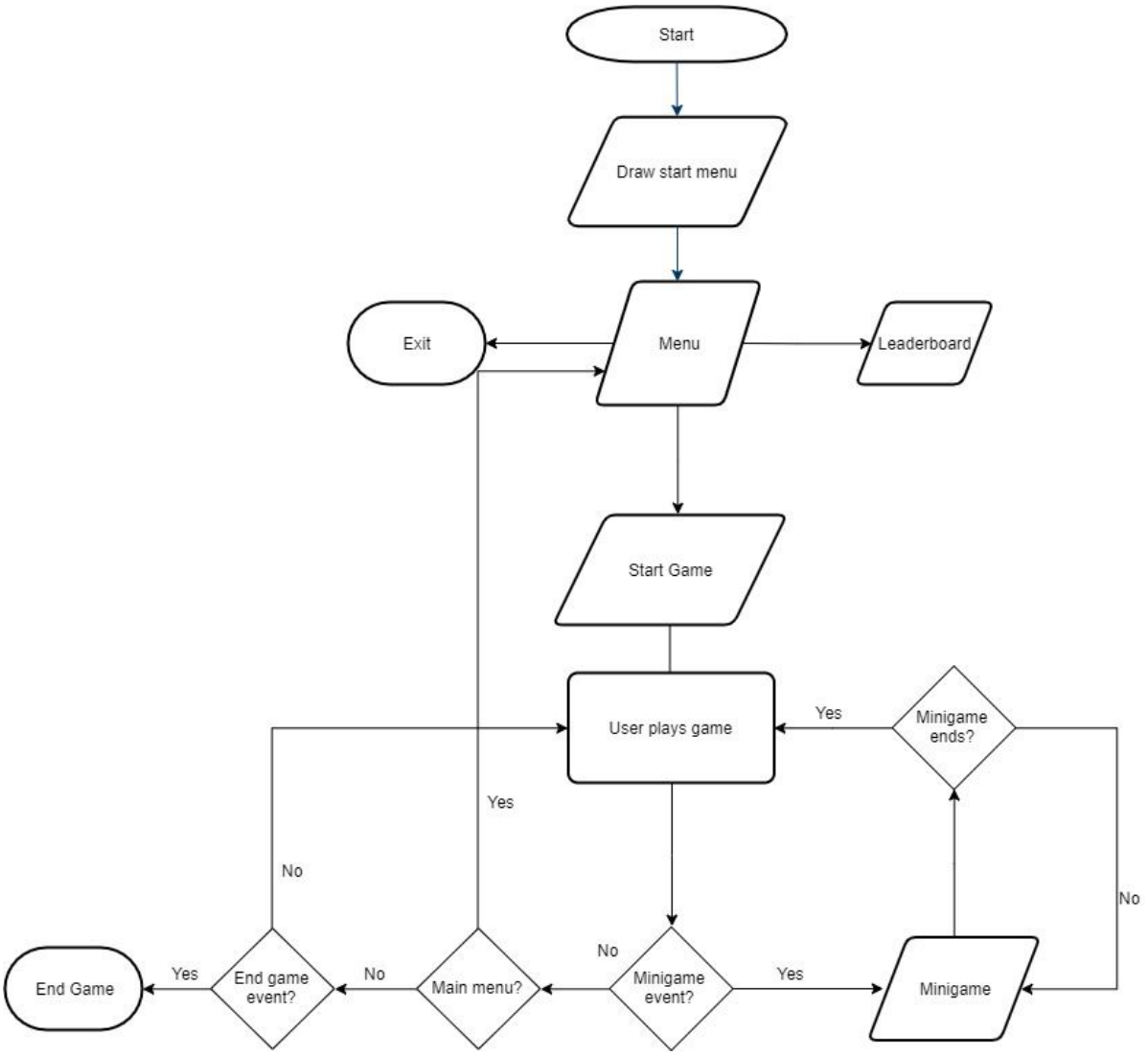


*Figure 2: Gameplay flowchart*

Figure 2 shows the abstract gameplay flow that we intend to use in the game. It describes how the user will interact with elements when the game is started.
The minigame is a part of the user requirements thus must be included,  it will be triggered by a special event during the game (shown in Figure 2) as a way for the player to make more in-game money. The specifics of how the minigame will be activated or displayed still need to be decided.

We used the tool draw.io to create the UML diagram (Figure 1) and the flowchart (Figure 2) [5]. This tool is useful for the creation of a wide range of diagrams and proved useful for the development of the architecture.

**UML Classes Justification**

**Game**

The Game class is where the main game operations will be executed.
The reason why instances of objects such as Bases and Vehicles are stored into lists and not arrays is because the player, or the enemy, could spawn an indefinite number of vehicles or bases depending on how long the game lasts, which is impossible to know in advance. We could use a dictionary to store each type of instance to make the program more efficient and avoid too many iterations, but as the game will be relatively small, the changes in performance will go unnoticed.

**Menu**

As stated in the user requirements, the Menu class will display the menu, leaderboard and minigame. This menu will be an overlay on the actual game.
While it would be possible to put this in the game object, it is more useful to be able to call menu functions from this class to use/create a menu when needed.

**Map**

The Map class will handle all operations involving the creation of the map.
There will be multiple different functions needed for the loading/creation of the map, meaning it makes sense for this to be handled in its own object.

**Camera**

The Camera class will hold operations handling the view of the game. It will handle resizing of the view as well as the translation of the view across the game world.
It is needed because there will have to be some way of controlling the view so that it can follow the player character between the four different fire trucks that were specified as a user requirement.

***Base***

The abstract Base class represents the fortress objects. It will handle the base fortress functions and variables.

**Timer**

The Timer class holds methods for creating timers. They can be used as components in other classes to determine, for example, how often a vehicle can shoot or when a fortress is ready to spawn a new vehicle.

***Vehicle***

The abstract Vehicle class represents the vehicle objects.
This class is needed for both the human and alien vehicle types. It contains basic variables that will be needed by both. As specified in the requirements a destroy() method is included so that alien and human vehicles can be destroyed.
The class will use components such as “collidable” which will give it collision physics

**HumanBase**

This class extends the functionality of the Base abstract class, with the only significant difference of overriding the “SpawnVehicle” method, as human fortresses will need to spawn humans vehicles rather than alien ones. It could possibly have other features (methods and variables) specific to humans that have been left out of the diagram as the requirements might change in the future.

**AlienBase**
AlienBase extends the functionality of the Base abstract class, with the only significant difference of overriding the “SpawnVehicle” method, as human fortresses will need to spawn alien vehicles rather than human ones. It could possibly have other features (methods and variables) specific to aliens that have been left out of the diagram as the requirements might change in the future.

**HumanVehicle**
HumanVehicle is the child of the Vehicle class, it acts the same as the Base class but with features specific to a human vehicle.

**AlienVehicle**
This class is the child of the Vehicle class, it acts the same as the Base class but with features specific to an alien vehicle, for example the ability to move in groups.

**Justification of overall structure**
When we got together to design the structure of the game, we asked ourselves what the basic building blocks of the game would be, and the first two things that came up were the Base and Vehicle class, as the game is about fortresses which product moving objects to destroy each other. As the Bases for both Humans and Aliens would realistically have the same behaviour, we decided to make a main abstract class for the base and one for the vehicle so that humans and aliens could inherit from them, so that we could limit code duplication. At this point we realised that the Base would need a way to determine when it’s the right time to spawn a new vehicle, as it wouldn’t be realistic to spawn a vehicle at every single frame of the game. So we decided to assign to it a Timer component that would handle that decision.
We wanted to limit the use of global variables so we decided to encapsulate the bases and vehicles into a game object, also to favour abstraction. The Game object also has a camera, a map and a menu components as from our research we found out they are the basic building blocks of almost every game.
We still had the problem of deciding where the collision detection logic and damage detection would be. To increase code reusability and limit duplication of collision and damage logic in both the Base and Vehicle objects, we decided to create a Collision and Damage objects that will handle that based on the arguments they get.

**References**

[1] M. Jordan, “Entities, components and systems,” *Medium*, 20-Nov-2018. [Online]. Available: https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d. [Accessed: 02-Nov-2019].

[2] R. Wenderlich, “Introduction to Component Based Architecture in Games,” *raywenderlich.com*, 2013. [Online]. Available: https://www.raywenderlich.com/2806-introduction-to-component-based-architecture-in-games. [Accessed: 02-Nov-2019].

[3] Unity Technologies, “Unity - Manual: Creating and Using Scripts,” *Unity3d.com*, 2019. [Online]. Available: https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html##targetText=Unity%20allows%20you%20to%20create,the%20C%23%20programming%20language%20natively. [Accessed: 02-Nov-2019].

[4] “Why are MVC & TDD not employed more in game architecture?,” *Game Development Stack Exchange*, Sep-2010. [Online]. Available: https://gamedev.stackexchange.com/questions/3426/why-are-mvc-tdd-not-employed-more-in-game-architecture. [Accessed: 02-Nov-2019].

[5] "Flowchart Maker & Online Diagram Software", *Draw.io*, 2019. [Online]. Available: https://www.draw.io/. [Accessed: 08- Nov- 2019]