Section 5: Architecture/Component design

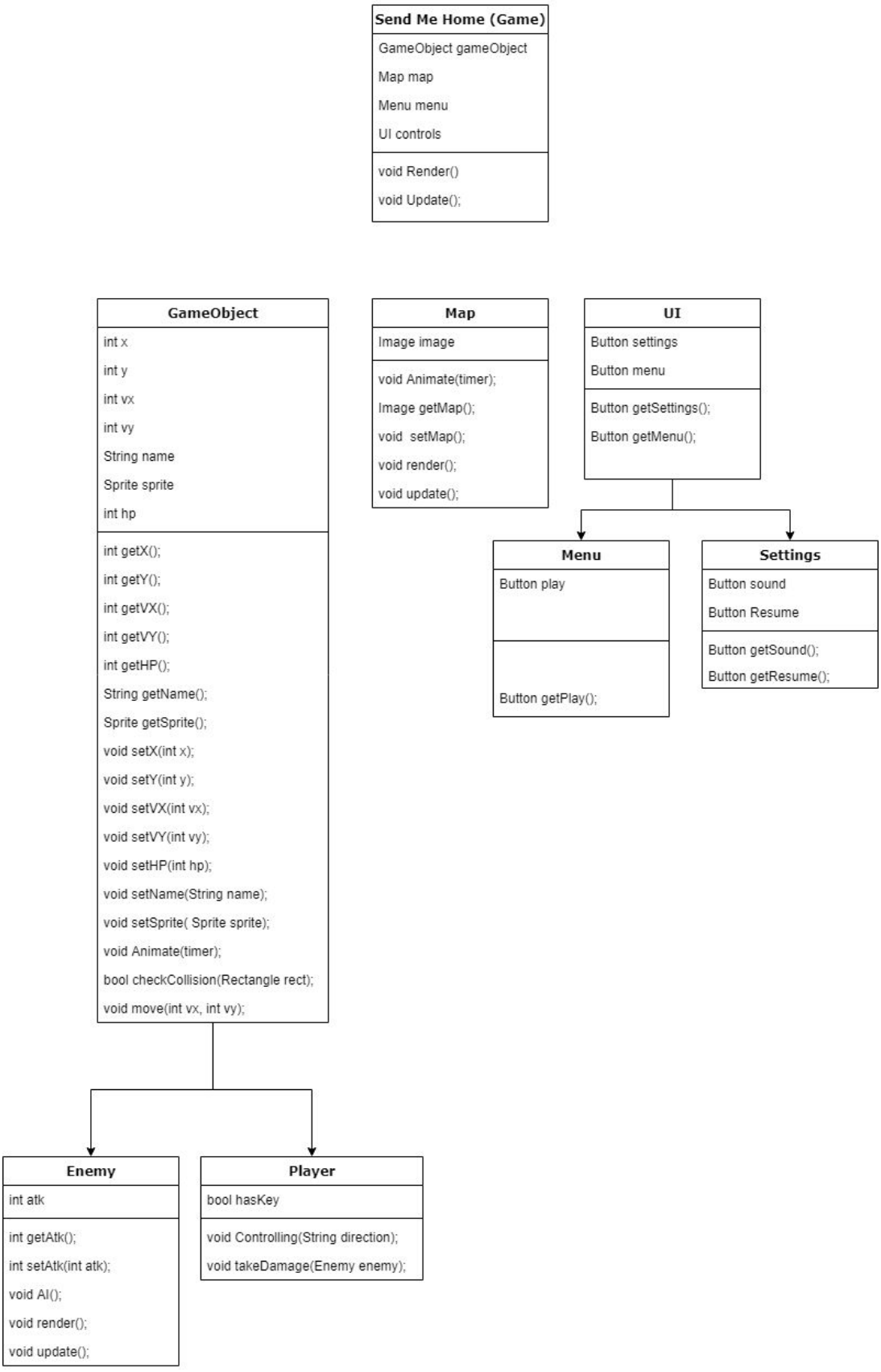
Architectural Style

Object-Oriented architecture focuses on splitting up responsibilities of our application into individual reusable objects. Using classes we can easily define objects such as enemies and since we have split our application into reusable objects it makes it easy to create more enemies. Object oriented architecture will also allow us to easily make objects such as buttons to use for starting the game, pausing the game, changing settings, controls to play the game, etc.

5 architectural design considerations

Our design addresses the five architectural design considerations Because our game is relatively simple, and is just a platformer, the design is economic. Using Object-Oriented architecture allows our game to have visibility because it is a well known architecture type. Most people will be able to guess what the code design would look like just based on the name. Our project will be split up into classes which will allow our project to have spacing. The system will be consistent and balanced due to our classes. Our game is event driven. The player will give input via the menu/ controls and the game will respond by playing the game, or moving and updating the position of objects, that is what gives our project emergence.

Class Description Graphic



Component Design Principles

The Open-Closed Principle:

We have a class called “Player" which inherits from our “GameObject”, We can create functions and make adjustments to “Player” without having to adjust or make any modifications of “GameObject”.

The Liskov Substitution Principle:

Our “Enemy” and “Player” classes both inherit from “GameObject” which means that in place of “GameObject”, we will be able to use either “Enemy” or “Player” as a substitute without causing any issues.

Dependency Inversion Principle:

Since we are using an Object-Oriented architecture, this will allow us to apply abstraction via inheritance. This way we won’t have any concrete components until we actually need to create the player character or make the play button for the menu.

The Interface Segregation Principle:

Each user will have their own game data. This will allow the user to download the game and have their settings and progress saved specific to the individual.

The Release Reuse Equivalency Principle:

We will label all classes which are reusable. It can be helpful when we need to manage and control as newer versions evolve. We are also using object-oriented programming which will allow us to use certain objects (classes), such as the gameobject class and enemy class, multiple times.

The Common Closure Principle:

The classes will be based in groups that have a main class and subclasses that will change together due to inheritance and related items. For example, if our animation function changes in the gameobject class then it’s going to change for the enemy and player class as well.

The Common Reuse Principle:

Only put the classes which can be reused together in a package together. When some classes in this package change, the release number of the package will change. All classes which invoke this package will need to change and be tested. The classes outside of this package shouldn’t need changing because they are not reused together. For example, if something changes in the UI class, then it shouldn’t need to be changed in the player class.

Section 6: Interface design

We will have a menu which will allow the player to start the game. We will keep a consistent color scheme throughout the menu, settings, and gameplay. Our buttons will be simple to use and include graphics that say exactly what the button does. This will allow the user a consistent and easy to use experience. The objective of the game will be simple and we will provide instructions for the player about movement and controls therefore placing the user in control, reducing their memory load, and making a consistent interface.