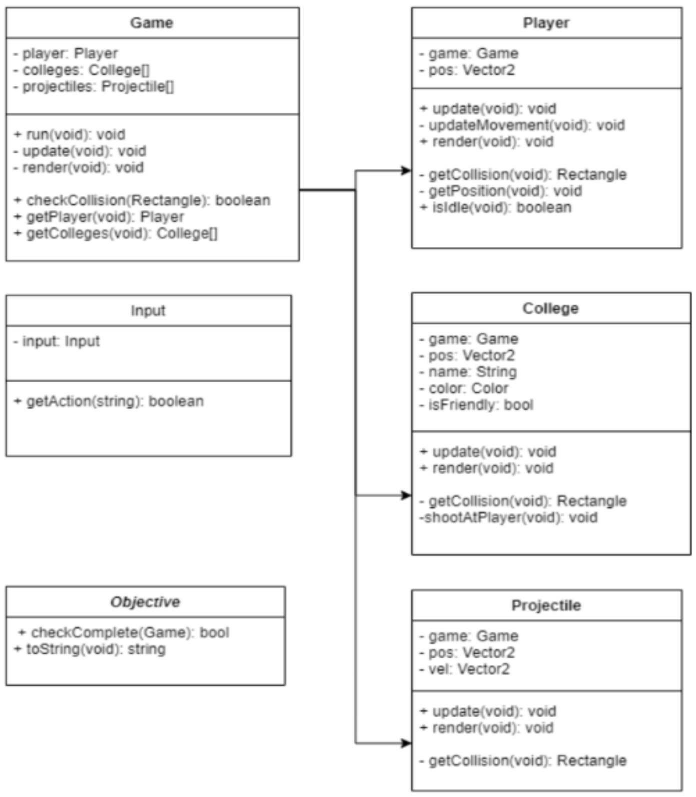
3. Architecture

Team 5 | Team Pending

Thom Robinson
Iris Scoffin
Ayman Elsayed
Annice Headman
Izaac Threlfall
Yihong Zhao

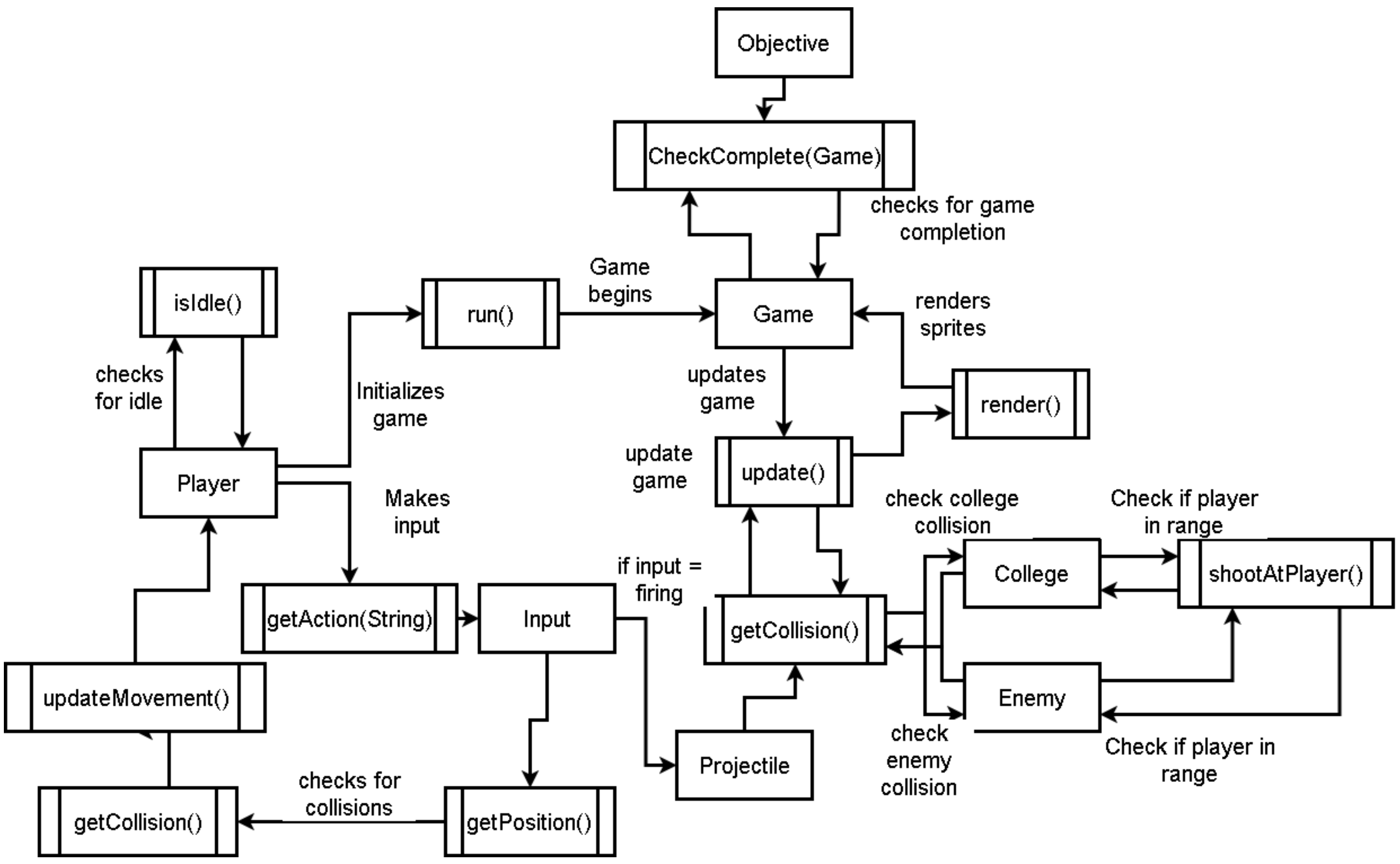
A. Representation

The initial abstract UML class diagram of our game:



The abstract data-flow diagram for our game: (See higher resolution version on our website)

Our initial abstract representation contains 6 main classes, these are the classes we have deemed as



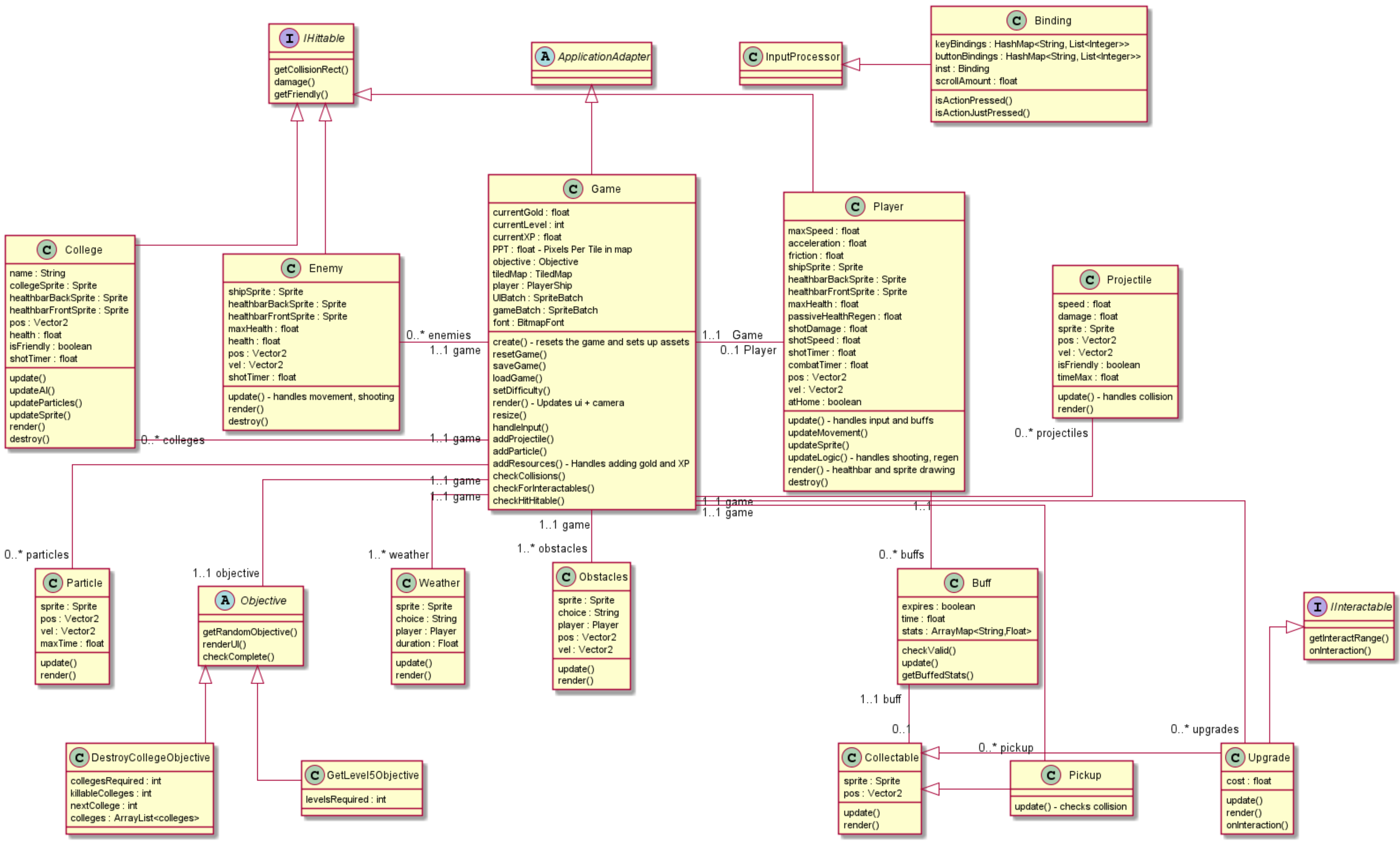
core components which we will need to meet the objectives. We have populated these classes with basic methods which we believe each class will need.

The 6 basic classes are:

- Game - The game class will be the main class for the game. It will keep references to a single Player class instance, and the multiple College instances. Within the Update and Render functions, it will call each instance's update and render functions. During construction, the Game class will load the tilemap, as well as extract the collision objects from it.
- Player - The player class will handle the players ship within the game. The player class will keep a reference to the Game, to allow it to access it's public functions. One of the main functions that it will access, is the checkCollision() function. This allows the player to check whether it is currently overlapping with the terrain and react accordingly.
- College - The college class will handle all the colleges within the game. The class will keep reference to the game and will use its public functions. Like the player class it will make use of the render and update functions as well as have a shotAtPlayer() function to allow for combat.
- Objective - The objective class will keep track and check for completion of the game and individual objectives. This class will have functions to check the game state, and text which indicates the requirements.
- Projectile - The projectile class will create and manage all projectiles within the game. Projectile will also have its own Update() and Render() functions, which will make sure it continues on moving in the direction it is created with. It will also contain a reference to Game, which allows it to use the checkCollision() to be able to detect when it hits a Player or a Wall.
- Input - The input class will handle the keyboard and mouse inputs from the player and, through mapping, ensure the correct action is taken within the game. This will include moving the player, firing projectiles and resetting the game.

The data flow diagram above also gives a basic overview of how the classes interact with each other using the functions mentioned.

We created our abstract class diagram, and abstract Data-flow diagrams on diagrams.net (formerly draw.io) flowchart maker. The concrete class diagram was initially created using Visual Paradigm, but later redesigned in PlantUML using VSCode, due to Visual Paradigm being a pay-for-use tool the new team did not have access to. Both the concrete and abstract class diagrams use UML as it was deemed appropriate for this project because of its tools which it provides for representing object oriented design.



Updated Concrete Architecture UML Diagram (See higher resolution version on our website)

B. Justification

Game Class - Most central game logic is handled within the Game() class, which holds references to all game objects. This made checking collisions, changing difficulties, altering attributes and saving to JSON simple. Furthermore, important variables in game, such as arrays of game objects are encapsulated by being made private, and providing public getter / setter functions. Meaning they cannot be changed without authorisation.

Health - Health and maxHealth are private attributes of the Player, Enemy and College class. They will be used to give visual feedback on the health of the ship and enemies as well as passed through the game class to check for changes to the Objectives or GameState such as the health of a college or the player reaching 0 (F_Player_Death, F_Enemy_Death and F_College_Death). The updated GameState or objectives will be shown to the player via updated visuals or splash screens (F_End).

Health regeneration is done in the update() function of Player, regenerating health passively while in the open ocean but regenerating significantly faster while 'docked' at a friendly college (F_Regen and F_Repair), to do this the isFriendly method and a proximity check determines if the ship is "docked" at a friendly college. However, when playing in the "Impossible" difficulty, regen is removed to increase difficulty. XP is also accumulated passively as the player sails around the map via the Game classes xpGain method. (F_XP).

Collisions and IHittable - checkCollision() is a function in Game that checks a Rectangle against everything that can be hit and damaged. To support this the IHittable interface was created which Player, Enemy and College implement. This interface contains functions such as damage(), getFriendly() and getCollisionRect() which are used in collision detection and damage between friendly and non-friendly objects. (F_Collision)

The benefit of an interface over using a default implementation of the function is that the player, enemies and college can react differently - the Player will update the Game class when it is destroyed and the game state will change, meanwhile the College and Enemy will destroy when it runs out of health. The visual update of destruction and health bars can be handled in other functions either in render() or update(). The College will need to update the Game class however, to allow for keeping track of the objectives such as the college becoming captured. This should also provide the player with XP as well as gold (F_XP).

Combat with colleges - Done using College's IHittable interface and an "isFriendly" attribute to check if a bullet hits a non-friendly college. The college also makes use of the same attribute to distinguish between enemy and friendly ships. The college can also check if the player is within range and will call the projectile class to fire (using Game's addProjectile()) at the player, damaging them upon collision when appropriate (F_College_Attack). Finally, when a college is destroyed the player is given gold and XP. (F_Rewards)

Enemies - The Enemy class adds enemy ships, initially, this class had limited functionality, including randomised movement and checking for collisions with the map. In part 2 we added movement towards the player when the ship is in range and combat between enemies and the player. As a result, Enemy now implements IHittable to enable bullet collisions (using checkHittableHit() in Game), damage and rewards (F_Ship_Attack / Death, F_XP).

Objectives - We decided to implement Objective as an abstract class. Other Objective classes such as DestroyCollegeObjective and GetLevel5Objective inherit from this abstract class. The Game class retains a reference to an Objective class. This meets requirements F_OBJ_Comp and F_OBJ_Track by keeping track of the player's progress. By having objectives that are multi-step and are tracked throughout it increases the game time to help achieve NF_Timing while not drastically increasing the difficulty level (NF_Difficulty).

Stats and Help - We also added on screen advice and stats for the player so they can easily tell how much gold, health and XP they have as well as how much health the enemies and colleges have. Also, tutorial hints such as game controls or mission objectives are displayed. This is done via the tutorial game state and UIBatch in the Game class and will display objective progress as well as Player attributes on the UI. (NF_Help and NF_Stats).

Input - The class Bindings, inherits from LibGDX's InputProcessor and handles all keystrokes from the player giving us the ability to change the keys for different actions within the game. This class allows us to meet F_Restart_Capable, F_WASD, F_WASD_Diagonal, F_Attack, F_Restart_End, F_Close and NF_Keys as the class allows for inputs to be mapped to the necessary actions for each requirement.

The game is fully single player (NF_Players) and does not have the capability to access any kind of network (NF_Network_P, NF_Network_I).

Particles and visuals - We also added a Particles class which creates particles within the game, it has its own render and update functions to do this.(F_Particles) This combined with the colourful artstyle were used to make the game more appealing visually and to attract the target audience. (NF_Audience). The simplicity of the artstyle makes the static map easier to understand for new players to navigate. (NF_Map). The colleges, enemy ships, loot and map have distinct colours and shapes, there's also no overlap between red and green usage throughout the game, this allows colour blind people to play with less difficulty (NF_Colour). The game also does not include any flashing images or lights to help reduce the risk of causing a seizure while playing (NF_Seizure).

Obstacles and Weather - The classes Weather and Obstacles were added in order to satisfy requirement F_Obstacles. Both of these classes use a "choice" string on instantiation, determining the kind of obstacle or weather to be created and changing the sprite and update() effect to match this. However, the two classes differ as Obstacle uses a collision rectangle and the checkCollisions() function in the Game class to determine collisions and affect the player. Whereas, in Weather, its sprite takes up the whole screen and just provides an effect to the player at all times, for a given duration.

Buffs and Upgrades - The Buff class provides a specific buff to the player for a specified amount of time based on a given stat provided when instantiated. This class provides the functionality to give the player a buff, these buffs are then associated with objects that inherit from the Collectable class. In particular, Pickup and Upgrade classes inherit from Collectable. However, Pickup creates a static object on the map that upon collision with the player, applies the given buff to the player for the time (F_Powers). Whereas, the Upgrade class only provides the buff when the player interacts with the object with the "E" button to spend a displayed amount of gold, this interaction is provided by implementing the methods from the IInteractable interface, satisfying the requirement to spend plunder (F_Plunder).

References