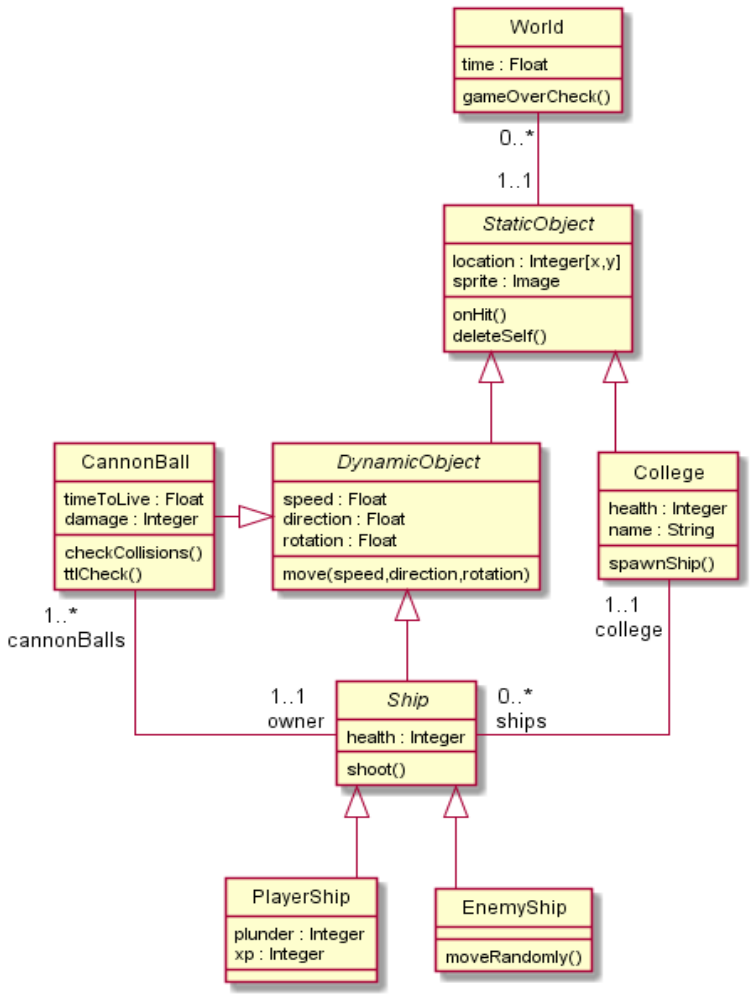
Section 3A - Concrete and abstract architecture

Initial architecture discussion took place shortly before we began implementation, the discussion took place in one of our weekly meetings where we brainstormed the interactions between objects in the game and how these objects related to each other. From this meeting we came up with quick pseudo-sequence and class diagrams on paper as well as brief textual descriptions of interactions between classes.

From there, we decided to use the simple and intuitive modelling language PlantUML to create our Sequence and Class diagrams. Creation of these diagrams were supported by the PlantUML extension in Visual Studio Code, allowing quick preview of the diagrams to rapidly build the diagrams from our initial architecture discussions and paper prototypes. Furthermore, Visual Studio Code provided syntax highlighting and basic predictive suggestions, in conjunction with the documentation and examples on the PlantUML website made it simple to pick up and use PlantUML.

The use of UML helped to support implementation of a game by giving us a starting point for our implementation and a rough outline of the kind of interactions, attributes, and methods of our game objects. Furthermore, producing UML concrete architecture retrospectively produces a valuable overview of our game and structure, which helps to document our code further. This will be useful for the team taking over our project to understand the architecture of our project.

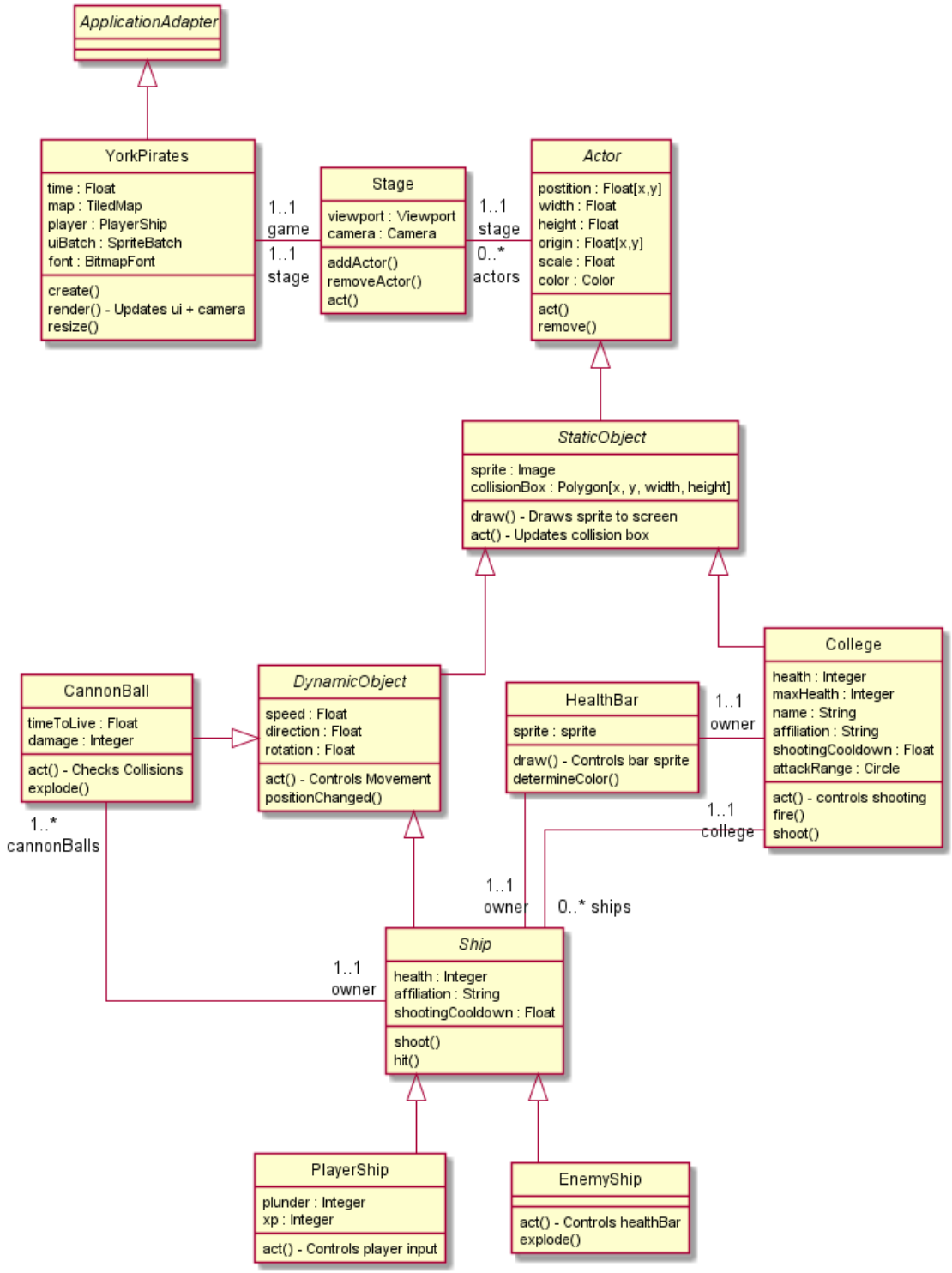


Abstract Architecture

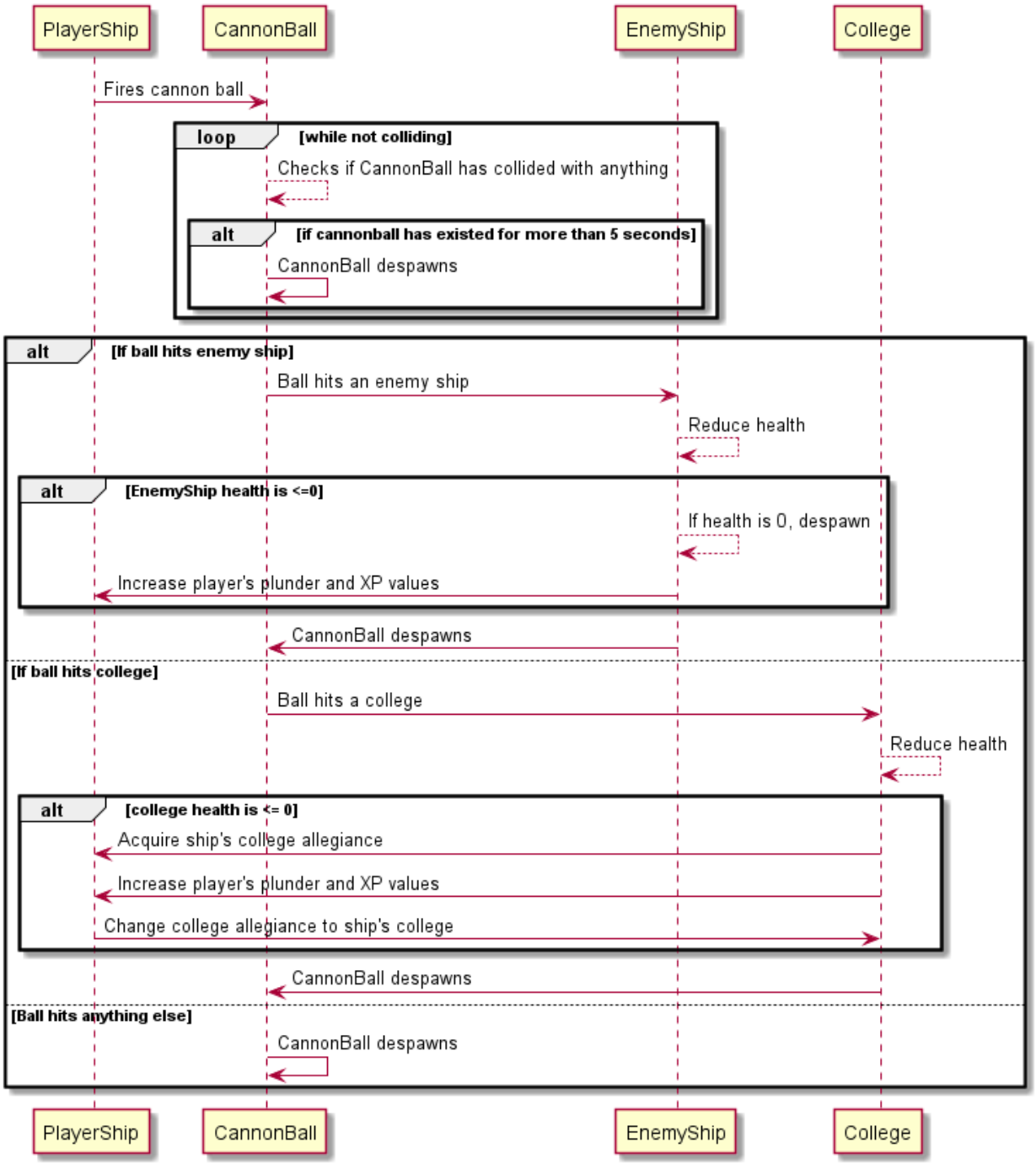
This diagram represents our abstract architecture produced prior to the implementation of our game. Therefore, it has no reference to any methods or classes from LibGDX.

Concrete Implementation

This diagram reflects our concrete implementation decisions, as shown in the diagram it now includes classes from LibGDX that are extended or used within the game such as ApplicationAdapter, Stage and Actor. When compared to the abstract architecture, you can see that we maintained the class structure but added a few extra methods and properties, justification for some of these decisions can be found in part 3B.



Sequence diagram for interactions with the CannonBall class



This diagram represents the behaviour of our CannonBall class in our concrete implementation and how it interacts with the player and enemy objects.

There is an element of abstraction in this diagram as in the actual concrete implementation, there are more checks involved to ensure that the cannonBall has collided with a college. This involves checking if the college is friendly, i.e., they are both of the same affiliation and therefore not damaging them. However, this was omitted from the diagram to reduce the already large number of conditionals included in the diagram.

Section 3B - Justification of concrete and abstract architecture

ApplicationAdapter - Our concrete architecture builds upon classes and constructs from LibGDX. Our game is made runnable through the ApplicationAdapter class which YorkPirates extends. LibGDX supports producing executable .jar files that can be ran on multiple operating systems, thus supporting the requirement NFR_2 “The application should run on all major operating systems”. Furthermore, LibGDX creates the game in a windowed application, combined with the resize() function allows the game to be scaled to the size of a screen. This satisfies the requirement NFR_3 “The game should scale to any screen size”.

Actors and Stage - Our class hierarchy inherits from LibGDX’s Actor class, this gives all objects in the game access to powerful methods such as:

- Act(), a function that performs code defined by the player at a fixed interval, typically every frame.
- Draw(), a function used to draw the Actor into its parent coordinate system.

The two functions make our architecture design and implementation far simpler, as we don’t have to worry about producing complex functions from scratch to draw our sprites on the screen and perform functions every frame. The consequence of this decision ensures that we can meet the timescale of our project as stated in the planning section of 4B, as we are not wasting time “reinventing the wheel” by reproducing efficient code that already exists in LibGDX. Additionally, we utilise the LibGDX stage class to organise our actors, making it easier to control and check groups for collisions.

StaticObject - At the core of our concrete architecture is the StaticObject class, this class contains the root constructor function that is referenced by all other connected abstract and concrete classes. It sets up the objects bounds, collision box, position and initialises the sprite, not only is this a positive decision because it abstracts away complexity for higher order classes but also because it provides a single point of change when an adjustment is required for all objects, e.g., increasing the size of all collision boxes.

Movement / DynamicObject - Our movement system follows the best practice of reducing repetition in our code. This is shown in our concrete and abstract architecture’s movement system. The abstract class DynamicObject has an act() loop that will change the object's position based on its speed and rotation. Any classes that implement this class can call super.act() to process movement and control movement speed based on the object’s desired behaviour. This reduces the size of the codebase by removing the need for multiple movement functions, and reduces the work needed to change movement behaviour. This movement system satisfies the requirements FR_2 - FR_4 related to movement behaviour in the player ship. Additionally, this movement system supports the cannon ball class.

To control player movement, the act() function handles player input every frame, changing movement speed up to a maximum speed and controlling rotation of the ship based on a rotation speed. This satisfies the requirement FR_2 “The ship's movement will be controlled with ”WASD”.

Collisions / CannonBall - As shown in the sequence diagram, the interactions that cannonballs have are much more complex than we anticipated in our abstract architecture, with

cannonballs providing their own hit detection between classes using collision rectangles. However, for classes such as Ship and College, these facades abstracts the underlying complexity by providing a simple method shoot(). This method only requires a set of target coordinates to shoot a cannonball from the center of the ship/college. Which in the case of the College is the coordinates of any ships that don't belong to its own college as stated in requirement FR_14 “Colleges should fire upon unaffiliated ships”. However, in the PlayerShip class, these target coordinates are taken from the user's mouse position, therefore satisfying the requirement FR_7 “projectile should be launched from ship in the direction of the mouse pointer”.

This method also makes it trivial to spawn more cannonballs while ensuring old cannonballs are removed after an amount of time due to the time to live checks in the CannonBall class’s act() method. This satisfies the requirement FR_9 “Projectiles should have a lifespan, disappearing after some time”.

Furthermore, all hitable objects include a hit() method in our concrete architecture, allowing the cannonball class to invoke that method upon collision. Upon being called, those objects reduce their health, check if they need to be destroyed and in the case of objects shot by PlayerShip, collect plunder and XP. This helps to satisfy a large amount of requirements, including FR_10 and FR_11 which relate to gaining plunder and XP from destroying enemies/passively, as well as FR_15 “Colleges should be damaged by player’s weapons and players should take damage when shot by a college” and FR_21 “the player should sink when they have taken too much damage”.

When the hit() method is called, the CannonBall that hits that object will remove itself from the stage, as specified in FR_8 “The projectile will stop when it hits objects in the world”.

HealthBar - The HealthBar class was created to improve code reuse to create health bars above enemies and colleges as well as a health bar to represent the player character in the UI. This class provides a consistent visual style while eliminating the need to code health bars in each class. In the future, this class could easily be adapted to display health bars for other objects, such as obstacles.

TileMap - In the concrete architecture, we opted to use the TileMap object, and the program Tiled to create a tiled map system for our game. This program simplifies organising objects on the map. This satisfies the requirements FR_12 “The world should have a fixed map with colleges spread out evenly” and FR_17 “colleges should be surrounded by allied ships.

Finally, to ensure continuity with naming conventions used in LibGDX our class naming structure follows the PascalCase convention, with all classes named in the format “WordWord” and all methods following camelCase e.g “wordWord”.

Overall, the architecture could be described as “open” in the sense that abstract classes can be modified without altering the functionality of its children and parent inheritances. Also, adding new classes to the architecture should be a smooth process. As a result, we hope that this combined with our documentation makes it easier for the next team to take over our project.