Architecture

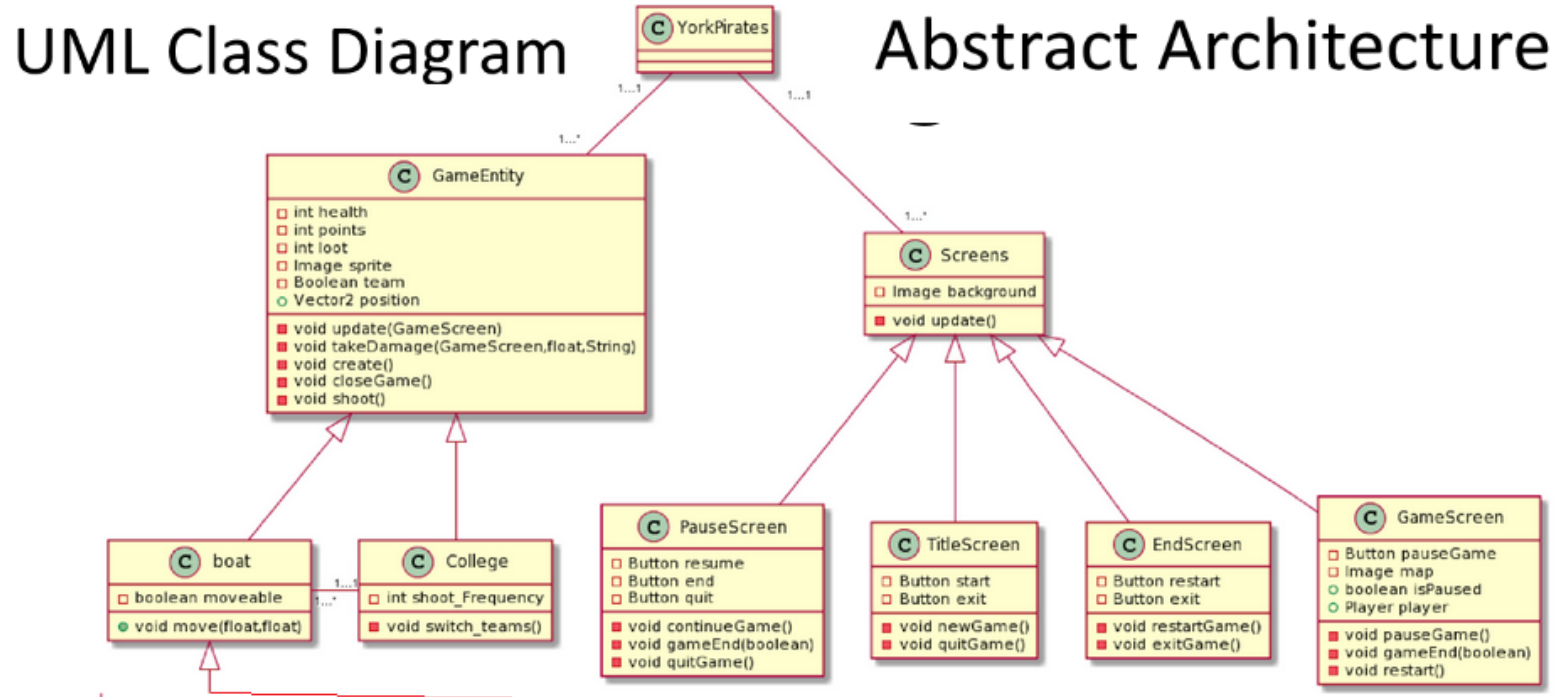
Assessment 1:
Team 14: Bass2

Katie Maison
Saud Kidwai
Jacob Poulton
Cody Spinks
Felix Rizzo French
Joachim Jones

Assessment 2:
Team 6: Team siKz

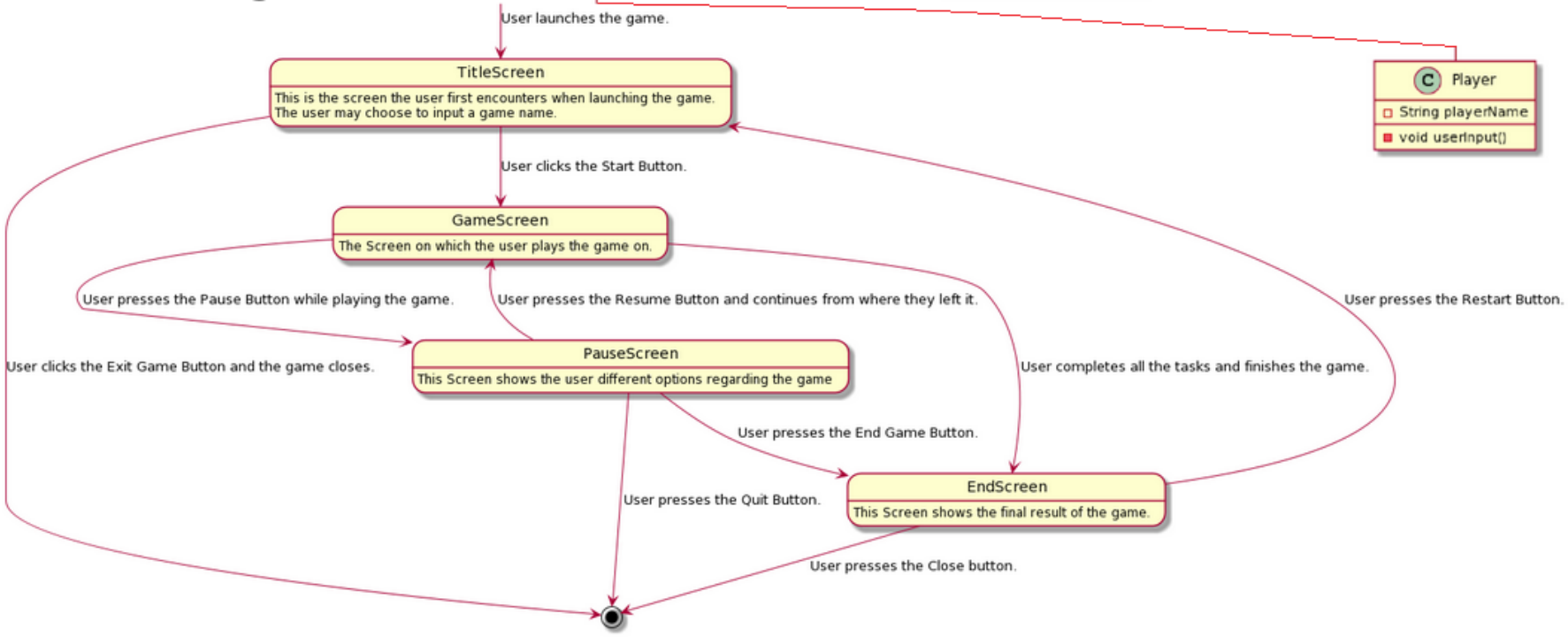
Ryan Bulman
Frederick Clarke
Jack Ellis
Yuhao Hu
Tom Nicholson
James Pursglove

UML Class Diagram

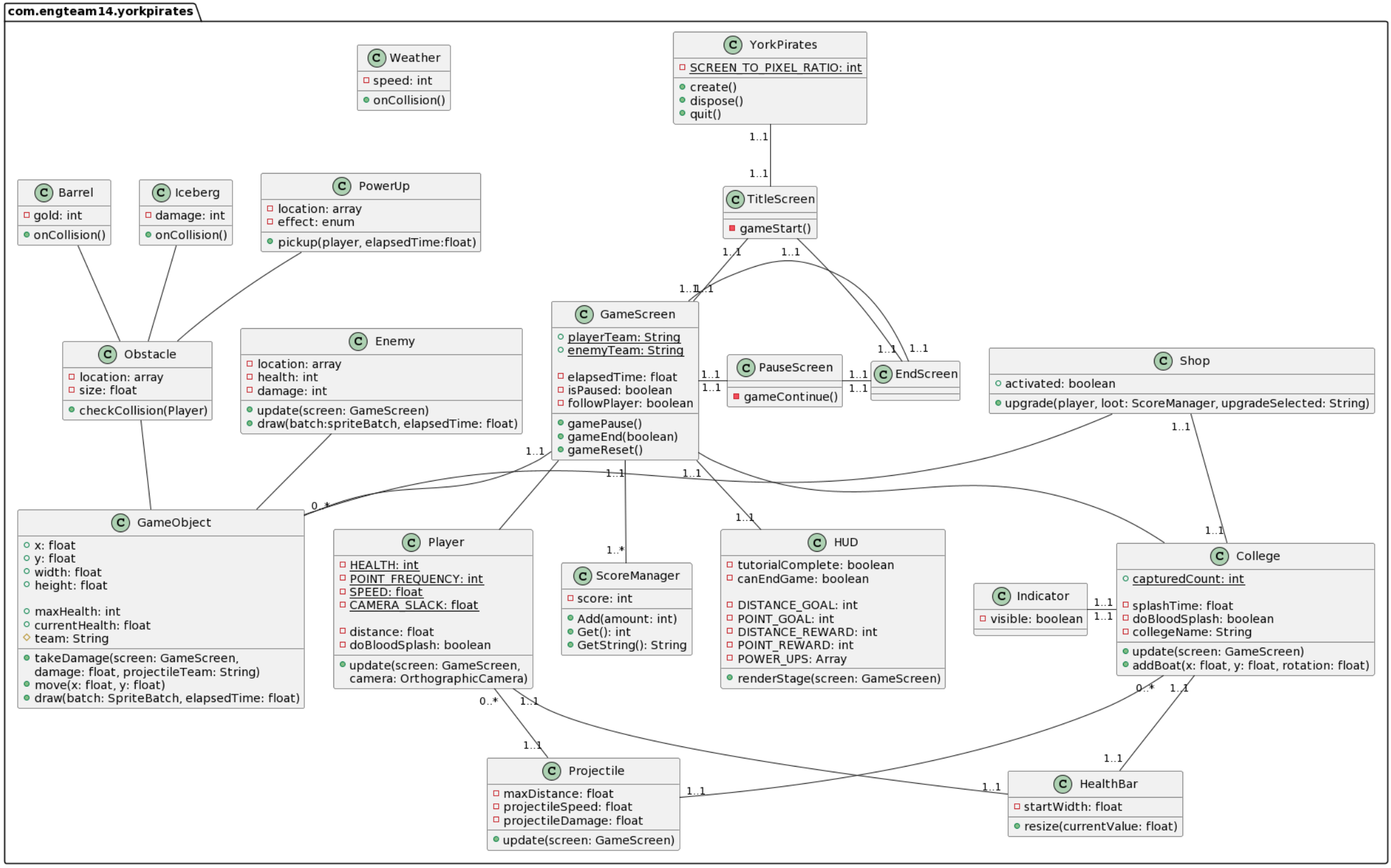


Abstract Architecture

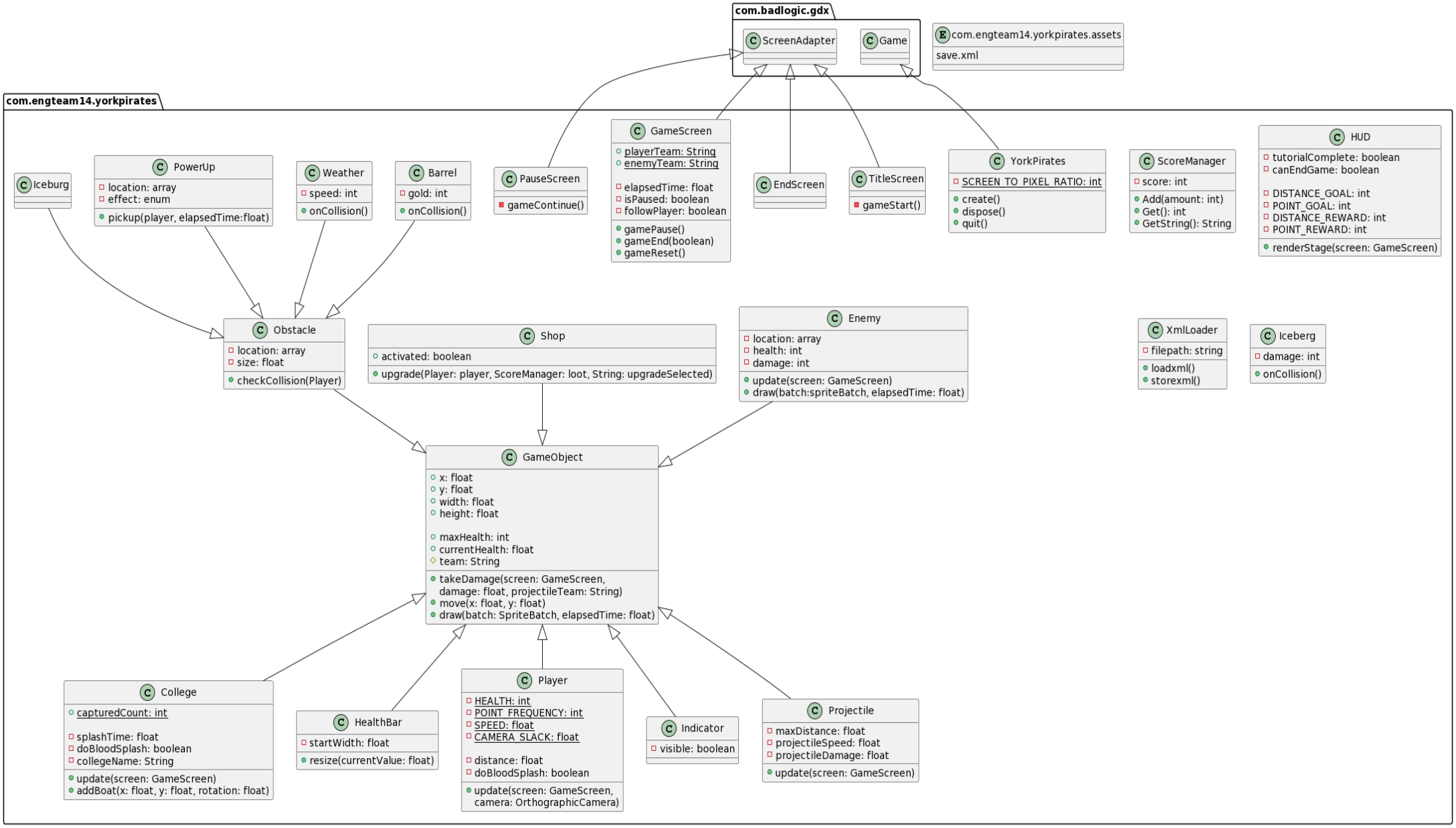
State Diagram

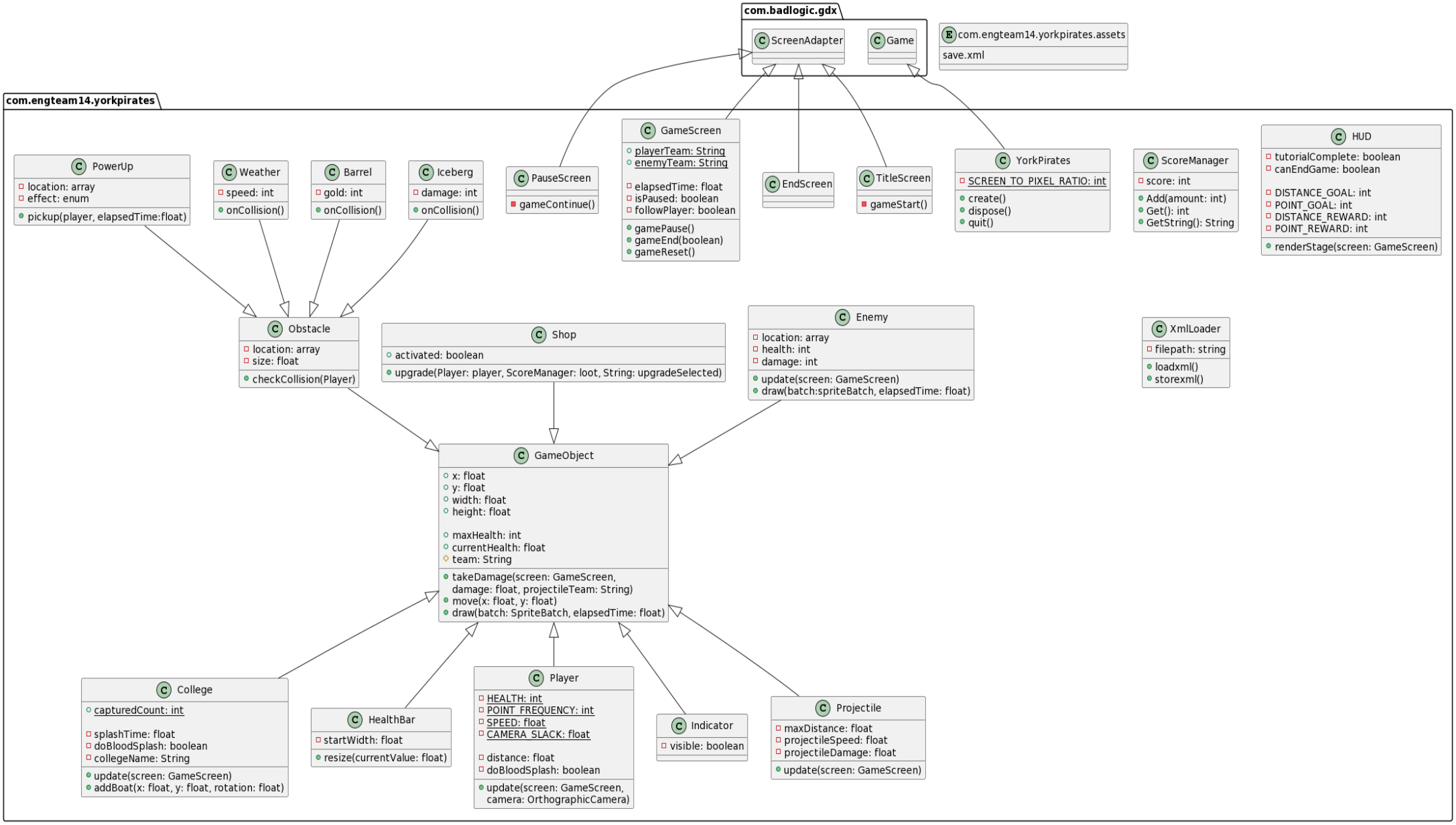


[2] https://engteam14.github.io/media/State_Diagram_4.png, [1] https://engteam14.github.io/media/Abstract%20Architecture.png



[3] https://github.com/TomNicho/yorkpirates/blob/master/abstract.png


com.badlogic.gdx
ScreenAdapter
Game
com.engteam14.yorkpirates.assets
save.xml
com.engteam14.yorkpirates
Iceburg
PowerUp
location: array
effect: enum
pickup(player, elapsedTime:float)
Weather
speed: int
onCollision()
Barrel
gold: int
onCollision()
PauseScreen
gameContinue()
GameScreen
playerTeam: String
enemyTeam: String
elapsedTime: float
isPaused: boolean
followPlayer: boolean
gamePause()
gameEnd(boolean)
gameReset()
EndScreen
TitleScreen
gameStart()
YorkPirates
SCREEN_TO_PIXEL_RATIO: int
create()
dispose()
quit()
ScoreManager
score: int
Add(amount: int)
Get(): int
GetString(): String
HUD
tutorialComplete: boolean
canEndGame: boolean
DISTANCE_GOAL: int
POINT_GOAL: int
DISTANCE_REWARD: int
POINT_REWARD: int
renderStage(screen: GameScreen)
Obstacle
location: array
size: float
checkCollision(Player)
Shop
activated: boolean
upgrade(Player: player, ScoreManager: loot, String: upgradeSelected)
Enemy
location: array
health: int
damage: int
update(screen: GameScreen)
draw(batch:spriteBatch, elapsedTime: float)
XmlLoader
filepath: string
loadxml()
storexml()
Iceberg
damage: int
onCollision()
GameObject
x: float
y: float
width: float
height: float
maxHealth: int
currentHealth: float
team: String
takeDamage(screen: GameScreen, damage: float, projectileTeam: String)
move(x: float, y: float)
draw(batch: SpriteBatch, elapsedTime: float)
College
capturedCount: int
splashTime: float
doBloodSplash: boolean
collegeName: String
update(screen: GameScreen)
addBoat(x: float, y: float, rotation: float)
HealthBar
startWidth: float
resize(currentValue: float)
Player
HEALTH: int
POINT_FREQUENCY: int
SPEED: float
CAMERA_SLACK: float
distance: float
doBloodSplash: boolean
update(screen: GameScreen, camera: OrthographicCamera)
Indicator
visible: boolean
Projectile
maxDistance: float
projectileSpeed: float
projectileDamage: float
update(screen: GameScreen)



[4] https://github.com/TomNicho/yorkpirates/blob/master/inheritance.png

We used the classes part of PlantUML to create class diagrams for the concrete and abstract structure of the project to show inheritance, in addition to state diagrams for the sequence of events that would occur throughout the course of our application's use using 'state' in PlantUML .We used intellij for syntax highlighting and for rendering the images. We simplified and removed getters, setters and some utility and instance attributes.

3.(b) Justification of Abstract Architecture

For our **abstract** architecture, we have focused on how we could structure it so that adding more entities and screens would be simple in future, and how we could reduce the code that would be duplicated in our code base. To do this, we made two main classes, **GameEntity** and **Screens** which both have the method **update()** which is to perform calculations before each entity/screen is rendered.

- **YorkPirates -** The main class of the game
- **GameEntity**
  - The class that every object within the game scene is an instance of. Implements health, rendering, teams, and shooting projectiles. As these are features all objects use, having a base class implement them is important.
- **Boat** and **player**
  - **Boat** inherits all attributes and methods of **GameEntity** and **Player** inherits both. **Boat** additionally has a boolean attribute of **'movable'** which decides whether the boat can move or not. This is because for this stage, the enemy boats can't move but the friendly one can. However in future some Enemy boats may become movable but some could stay docked.
- **Colleges**
  - Inherits all attributes and methods of **GameEntity** but also has a method called **shootFrequency** to set how often it shoots on its own and a **switchTeams** method for switching the images and turning off shooting/being shot by, the player, when the player captures it.
- **PauseScreen, TitleScreen** and **EndScreen**
  - Each has a different set of buttons needed for its screen and are child classes of screens to use its render method and all are child classes of screen to use the same background and update() method for calculations before rendering.
- **Game Screen**
  - When the game is restarted, a new instance of this is created so that the game doesn't have to be fully restarted and also has the methods for restarting, pausing and ending the game but is also a child of the screen.

Concrete Architecture

We started our implementation by creating the classes seen in the abstract architecture. While doing this, we came to find other more efficient, in-built features of LibGDX, such as **ScreenAdapter** or **TiledMap**, which provided better solutions than the ones in the abstract architecture. Additionally to this, as we developed more of the game we found ourselves needing new classes as well as to change old ones. The bullet points following discuss this.

- We have focused on the same features as abstract however have added some aspects for the ease of adding extra functionality in future. One of the outcomes of this is the addition of **TiledMap**s to the game. This allows us to rapidly draft prototype and final levels for the game using the software Tiled, which greatly improves development times, furthermore the **TiledMap** allows for the implementation of a co-ordinate based collision system which is largely more efficient than the previous **Rectangle** based one we used.
- **YorkPirates**
  - Due to the structure of LibGDX, we had to make a main Game class. This matches what we planned in our abstract architecture to an extent but screens are actually child classes of **ScreenAdapter** and YorkPirates instantiates TitleScreen and then switches between the others.
- **TitleScreen, EndScreen, PauseScreen**
  - These classes are extensions of the ScreenAdapter class and render their screens with Buttons and overlays on the paused instance of GameScreen. (note: the attributes for these classes are omitted for clarity). These inheriting ScreenAdapter is different to the abstract architecture as we were not fully familiar with the structure of libGDX. These classes fulfil the requirements: UR/FR.START_SCRN and due to TitleScreen, UR.SCRN_NAME / FR.START.NAME due to the ability to add a name on titleScreen, UR.RESTART_GAME due to the pause menu, FR.START.START and FR.START.EXIT due to the

TitleScreen, FR.KILL_SCRN due to the EndScreen class and FR.GAME_SOUND due to the mute button on the PauseScreen.

- **GameScreen**
  - This class is the main gameplay environment, containing and rendering all instances of the objects within the game, which meets requirement UR.SEE_POS. Furthermore it has the methods for pausing the game with gamePause(), ending the game with gameEnd() and restarting the game with gameReset(). We put those methods in this class because every other class that needs these has access to an instance of this class.
- **HUD**
  - We did not have this class in the abstract architecture but we added it for readability to avoid clutter in the main GameScreen class.
- **GameObject**
  - Every object in the game is an instance of **GameObject** where ones with seperate functionality are a child class of **GameObject**. This is so that common attributes and methods such as currentHealth, takeDamage and position within the world (x, y) are shared among all objects.
- **ScoreManager**
  - **ScoreManager** was created to lay the groundwork for future possible implementations of a more complex loot and points system. It also encapsulates the values, which in the case of points makes it easier to update the points value from the **Player** when they move() or the loot value from the **College** when it is defeated, meeting UR.COLLECT_POINTS and UR.COLLECT_LOOT.
- **College**
  - College is a child class of GameObject with the further features that it has **Projectile**s and a **HealthBar** and **Indicator**. This is in a separate class as it shoots automatically rather than through user input like Player.
- **Player**
  - In the abstract architecture, **Player** was a child of **Boat** because **Boat** allowed movement. However we decided to put the movement method into GameObject because **Projectile**, **HealthBar** and **Indicator,** also needed to be able to move and so therefore we could use the move() method for all of these, as well as in future, moveable enemy boats. This ensures we still meet the requirement UR.UPDATE_POS.
- **HealthBar**
  - **HealthBar** was not in our abstract, however we realised the **HealthBar** was needed for both the **Player** and the **College** and so to save us from code repetition we made **HealthBar** into its own class. This will also make implementing enemy boats in the future easier.
- **Projectile**
  - In our abstract architecture, shooting was implemented as part of **GameObject**, however as we now have more objects in the game and not all of them shoot. Having all objects do this would be inefficient so we moved it into its own class, which **Player** and **College** both use, allowing UR.ATK_CLG to be met.
- **Indicator**
  - In our abstract implementation we did not have a method which allows the user to see where they are relative to the colleges (UR.CLG_POS). This is why we added **Indicator**s, these draw arrows showing the player which direction each college is, fulfilling the requirement UR.CLG_POS.
- **Barrel, Iceburg, PowerUp, Obstacle**
  - We created an abstract obstacle class that the new obstacles could inherit from. We did this because we had a lot of new features to implement that would need to have collision with the player, such as barrels, icebergs and power ups (UR.POWER_UP, UR.AVOID.OBS). We then created classes that inherit from Obstacle with their own behaviour.
- **XMLload, save.xml**
  - In the existing implementation there was no file to save or load object data from. We created save.xml as a file to save data to and XMLload as a class containing functions that would save and load that data. (UR.SAVE_LOAD)
- **ShopUI, Shop**
  - We had been given a new requirement to implement a shop. To do this we created a single class Shop. This was related 1:1 to a college and had a boolean activated that was toggled on when the related college was captured. This class handled the upgrades, while the UI was managed in HUD and user input was taken from GameScreen.(UR.SPEND_LOOT).

Bibliography

[1] “York Pirates! Abstract Architecture Class Diagram” *York Pirates!* https://engteam14.github.io/media/Abstract%20Architecture.png.
[2] “York Pirates! State Diagram” *York Pirates!* https://engteam14.github.io/media/State_Diagram_4.png.
[3] “York Pirates! Concrete Architecture Class Diagram” *York Pirates!* https://github.com/TomNicho/yorkpirates/blob/master/abstract.png
[4] “York Pirates! Concrete Architecture Class Diagram (Inheritance)” *York Pirates!* https://engteam14.github.io/media/inheritence.png.