Software Architecture

- The abstract architecture was created with draw.io, with the basic relationships of the program being shown on the diagram
- The concrete architecture was produced with PlantUML
- The classes were separated into different categories with the connections within the category shown on the diagram
- The inter-category connections are later added through Adobe Photoshop with lines colour coded for easier understanding

Abstract Architecture

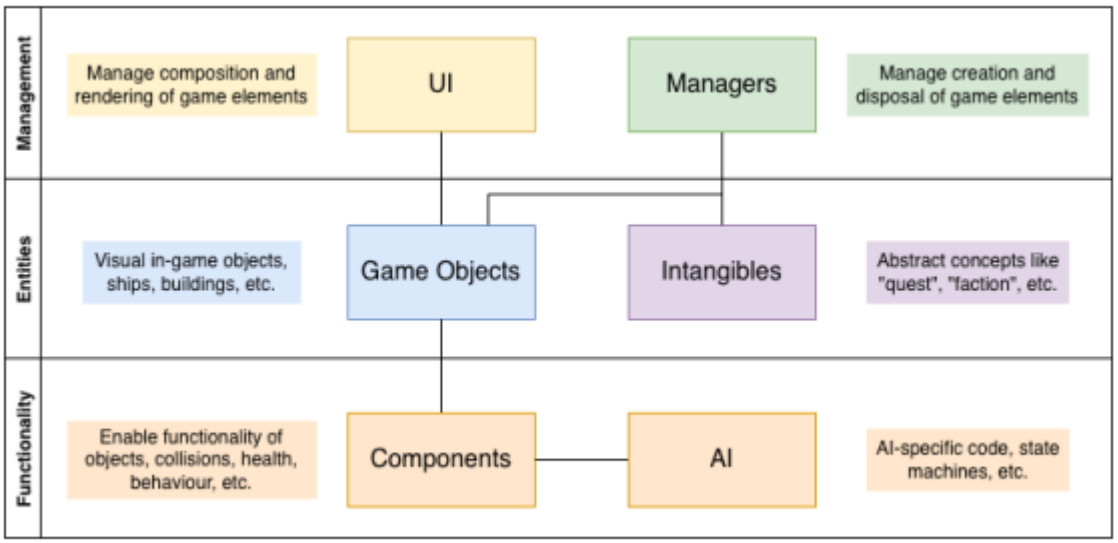


Fig 3.1.1: Diagram of the abstract architecture

Concrete Architecture

Points System (Assessment 1)

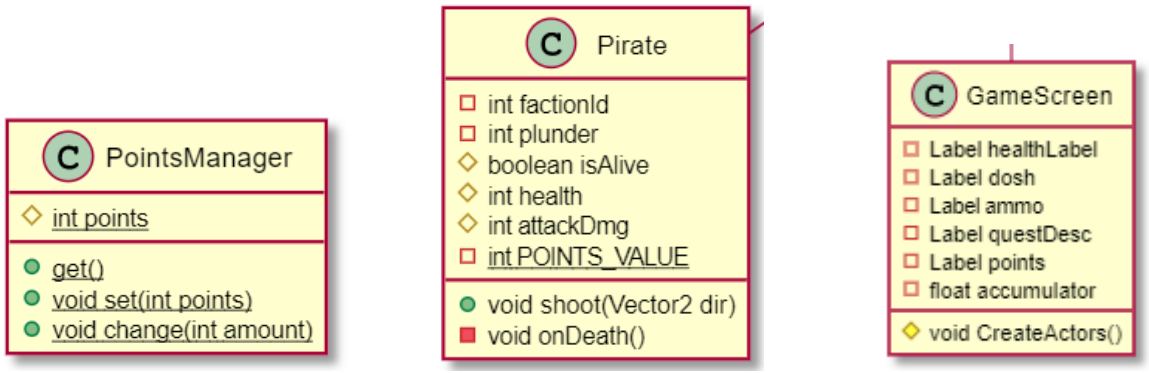


Figure 1.

- A new class ‘PointsManager’ was created in order to change the amount of points the player earns after performing specific tasks
- For example, the ‘Pirate’ class now has a variable ‘POINTS_VALUE’, which is the points awarded for killing an enemy pirate

Ship Combat

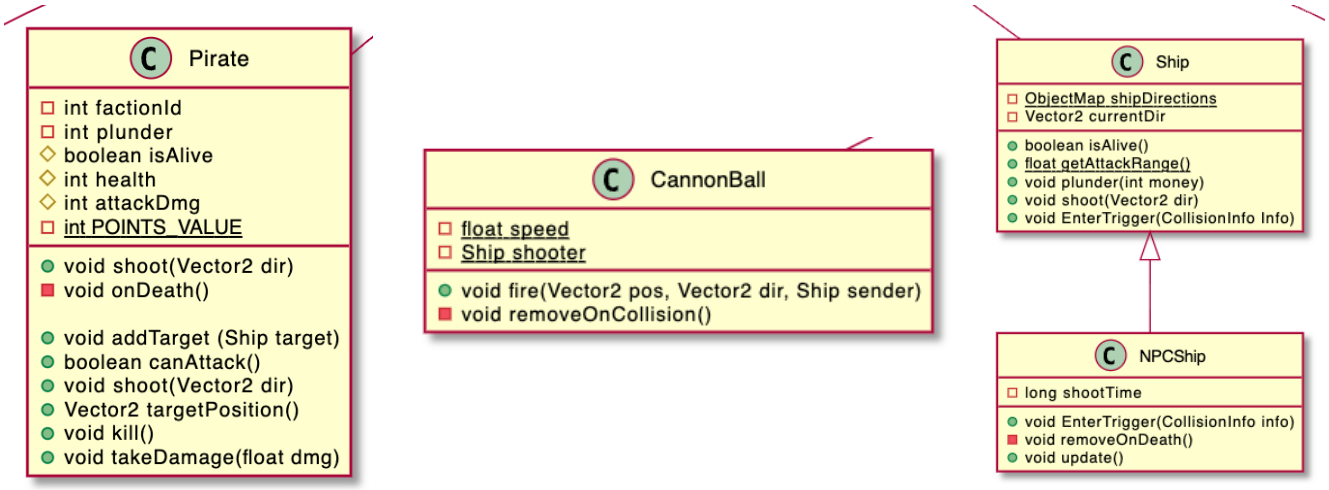
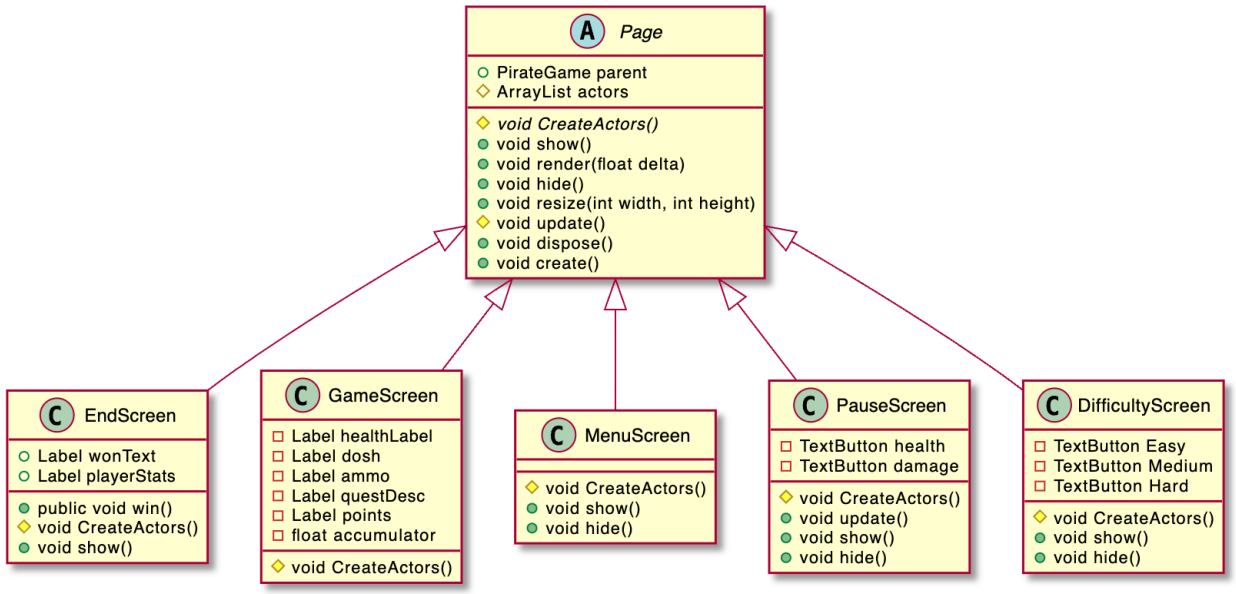


Figure 2.

- New methods have been added to ‘Pirate’, such as ‘takeDamage’ which is called when any enemy ship detects a cannonball fired from the player

Difficulty Selection



- Two new classes ‘PauseScreen’ and ‘DifficultyScreen’ have been added to Page
- The pause screen also controls the powerups, as shown in figure 4
- The difficulty screen is located on the main menu, and pressing a TextButton will call a method to alter the game settings, such as starting health and cannonballs

Figure 3.

Power Ups (Plunder System)

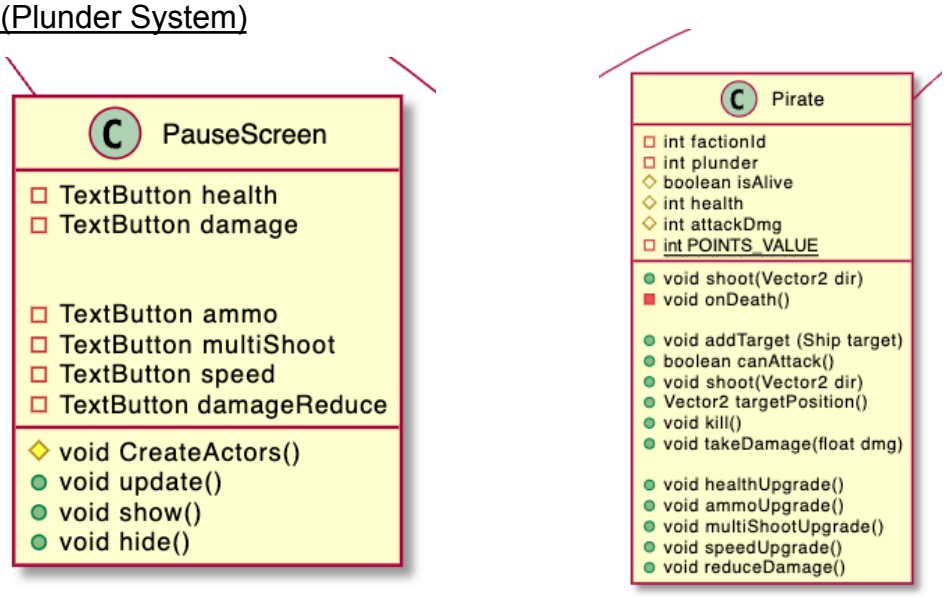


Figure 4.

- The TextButtons seen above are all new powerups, which when pressed on will deduce plunder from the player and call a method to activate a powerup, such as healthUpgrade()

Save State

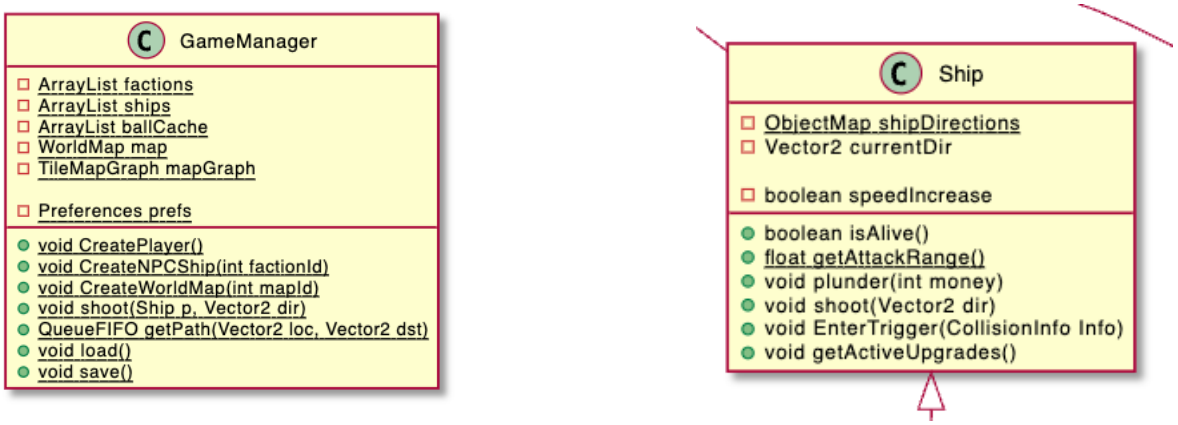


Figure 5.

- Two new methods load() and save() have been implemented to handle saving the player's stats to a file, and then loading them back in when starting the game

As mentioned in the change report, the complete architecture diagrams can be found here:

- https://drive.google.com/drive/folders/1Ux-RdH39QXHEAT0aGJmCgRMJGW-bPU9V?usp=sharing

The abstract architecture is concerned with segmenting the large, monolithic task of building the game into separate logical elements which could be planned and reasoned about separately. Connections drawn between elements signify a logical relationship rather than necessarily representing extension or composition relations such as those featured in the UML diagram detailing the concrete architecture. For example, factions/colleges ended up implemented as components and managed implicitly. Nevertheless, it is useful to see them grouped under intangibles while planning the overall architecture.

The concrete architecture builds on the abstract in two main ways, by capturing additional implementation details, and by reflecting the contribution of the game engine to enabling game functionality.

Additional specifics of the game's implementation are provided by means of detailing the class structure of the code, annotating the classes with their significant functionality in the form of methods and variables, and drawing the relationships between the classes on the diagram.

The structure of the concrete architecture is informed by that of the game engine. For example, we move from the UI element of the abstract architecture to a separate **Page** class and its subclasses responsible for rendering and composition of UI widgets, and the **Renderable** component and **RenderingManager** class for the rendering of in-game objects such as ships and buildings: this is due to how the game engine implements the rendering of different game aspects. In this way, the concrete architecture provides significantly more detail at a lower conceptual level than the abstract.

It should be noted that significant discretion had to be exercised regarding the level of detail captured in concrete architecture: it was neither feasible nor desirable to capture the full level of detail of the code's implementation. In the interest of using the concrete architecture as a higher-level abstraction used for reasoning about and planning the implementation, only significant functionality was captured and boilerplate methods and variables have been omitted.

Furthermore, we had to deviate from the UML standard to depict certain relationships without making the diagrams too large to display on A4 paper. Hence, some figures have the relationships between entities & components and their respective managers depicted in a shorthand form that we hope is nevertheless clear and informative.

Another point of note regarding the architecture and implementation is that during the process of implementation, certain approaches were selected that were not obvious during the architecture planning stage. For example, update methods called by the game loop were leveraged to provide certain functionality, like monitoring for game over conditions within the **GameScreen** class. These approaches were not foreplanned and are hard to document within a UML class diagram. Hence, a better reference to them would be perusing the rendered Javadocs associated with the game.

Relations To The Requirements

- Below are the requirements that needed significant changes made in the architecture, namely, the assessment 2 ones

UR_SHIP_COMBAT

- By referring to figure 2, there are numerous functions and variables that control the player and enemy ship's health, cannonballs, movement etc
- For example, both have a shoot() function that gets the position of the other entity and fires a cannonball
- Ship and NPCShip both have methods for detecting collisions, which allows them to interact with cannonballs to take damage, and allows NPCShips to detect and target the player ship.

- If an NPCShip collides with a ship that is not a member of its faction, it will add that ship to a list of targets that it follows and fires at.
- Also, the class ‘Pirate’ has a ‘POINTS_VALUE’ variable which is used to award the player with points if they defeat an enemy
    - UR_EARN_POINTS + UR_EARN_XP

UR_OBSTACLE_ENCOUNTER

- Obstacles were done through the tileset on the application ‘Tiled’
- Due to this, no changes to any class diagrams were needed as it involves no code

UR_SPEND_MONEY

- As shown in figure 4, a new UI class extending Page, PauseScreen, was added to enable the player to spend plunder
- This class interacts with the existing Pirate class to handle the player’s plunder value.
- Pirate is responsible for checking how much plunder the player has and decreasing the plunder value when they buy an upgrade

UR_POWERUPS

- By pressing Z during gameplay, the player is able to access the PauseScreen menu and spend their plunder on various powerups
- PauseScreen has various buttons which call methods in the Pirate class when clicked. Handling all five of these methods in Pirate made the code more manageable than having the buttons call methods in various different classes.
- Most of the powerup methods act on fields already a part of Pirate, with the exception of speedUpgrade(), which changes the speed variable in PlayerController

UR_DIFFICULTIES

- As shown in figure 3, a new UI class extending Page, DifficultyScreen, was added to enable the player to select a difficulty at the start of the game
- A new button was added to the existing MenuScreen class, which sets the screen to the DifficultyScreen when clicked
- Each of the buttons in DifficultyScreen interacts with the Pirate class to set the player’s starting stats based on the difficulty selected, so that they start with more health and ammo on the easy difficulty and less on the hard difficulty

FR_SAVE_STATE

- Saving and loading is handled by new methods in the GameManager class.
- The save method is accessed via a button on the PauseScreen menu and if a save file exists then the load method can be accessed via a button on the main MenuScreen
- The save data is stored locally using libgdx ‘preferences’

UR_EARN_POINTS

- A new PointsManager class was created to keep track of the player’s points. It uses a private int to internally store the points, and provides getter and setter methods for modifying the points.
- These methods are static, so that they can be called from anywhere in the code without requiring a reference to a PointsManager instance. This fits the concept of a points system, in which you may earn points for a variety of different actions, i.e. from a variety of positions in the codebase.
- For example, the Pirate class has been updated so that it calls the PointsManager.change() method when it dies, rewarding the player with a constant number of points when a pirate dies.