Implementing our Architecture and Requirements

In the following sections we explain how each requirement is implemented in the code base along with what classes are used from the concrete architecture.

UR_PLATFORM - We coded our project in Java using the LibGdx library. As a result of this our game deploys as a jar file, meaning that it can be run cross platform on any platform with Java. This satisfies FR_CROSS_PLATFORM_WIN, FR_CROSS_PLATFORM_MAC and FR_CROSS_PLATFORM_GNU_LINUX, as all of these platforms support Java. Furthermore, as LibGdx is based on OpenGL this allows us to run our game with stable graphics past 30fps (FR_MIN_FPS), and scale the game to different resolutions (FR_VIEWPORT_SCALING). LibGdx also provides an InputManager, allowing us to detect input easily in our game in order to satisfy FR_MENU_KB_INPUT. Beyond the basic framework of LibGdx, we also coded our project using an Entity-Component system. In this system, all entities inherit from Entity, and call components inherit from Component, this allows shared features such as event calls to be given to all entities and components. An Entity may have any number of Components, all of which provide different functionalities, for example a Transform gives an object a position, rotation and scale within the game world, while a Renderable allows that object to be displayed in the world with a sprite.

UR_GAME_INIT - LibGdx initialises the game from a base class, PirateGame, which extends the LibGdx class Game. This class is able to switch between different UI screens. To initialise the game, we call the initialization methods of relevant managers, as well as loading in important resources. To allow the game to begin from a starting configuration (FR_GAME_RESET), we used a settings file GameSettings.json. This is managed by the GameManager, which loads it in and provides all other classes access to it.

UR_SHIP_CONTROL - The Player has a PlayerController component, which processes input and calculates how to move the Player based on this. It has a method getDirFromWASDInput(), which takes keyboard input and processes it into a direction to be used in movement (FR_SHIP_KB_INPUT). This function makes it easy for both WASD and Arrow Key controls to be used. The main update() method also detects input for shooting, which can either be the space key (FR_SHIP_KB_INPUT) or the left mouse button.

UR_COMPETING_COLLEGES - The Faction class is used as a wrapper to store data for a specific faction in the game. It stores distinguishing values such as the name, colour, location, etc. Faction data is stored under the settings specified in UR_GAME_INIT, and loaded into the Faction class. This allows easy distinction between entities in the game, as Colleges and Ships have a Faction. These entities are all defined in the GameManager's SpawnGame() with a Faction, and then kept track of throughout the game (FR_COLLEGE_ENTITY_TRACKING).

UR_FRIENDLY_SHIP_ENCOUNTER & UR_HOSTILE_SHIP_ENCOUNTER - The NPCShip class is used for all ships that aren't the Player and hold reference to objects such as the stateMachine which determines how they behave, the AINavigation component which controls their movement based on assigned behaviours. Ships that are friendly or haven't discovered a target yet will reside in the WANDER state (FR_FRIENDLY_AI) which will have them travel a certain distance from their College while looking for targets, a behaviour which is defined in the circleOrigin() method. When a target is found, a hostile ship will switch to the PURSUE state (FR_HOSTILE_AI) which will enact the followTarget() method to bring ships in range of the target. Upon reaching the target they will then cease motion and enter the ATTACK state where they will attackShip() until the target either is killed or leaves the attack radius.

- **New feature** - NPCShip.circleOrigin() - A method that creates a new Steering Behaviour for NPCShips in the WANDER state to follow. The behaviour sets the ships on a trajectory around the College to which they are assigned, which they will circle until they come in contact with a target.

UR_FIRE_WEAPONS
The Pirate Component which is attached to Ships and Colleges implements the shooting functionality. There are a limited number of Cannonballs in existence, which are kept track of within the Pirate as ammo (FR_PLAYER_AMMO). NPCShips and Colleges fire when they are in a certain state and/or proximity of their target, whereas the Player can fire upon receiving the LEFT button input, which is read and processed by the PlayerController() (FR_PLAYER_FIRE).

UR_BULLET_DODGE
When cannonballs are fired they travel across the screen (FR_BULLET_TRAVEL) at a speed which is stored in the settings JsonValue referenced in the GameManager. While moving the Player, the path that these are travelling can be avoided, meaning that damage won't be received.

UR_SHIP_COMBAT & HOSTILE_BUILDING_COMBAT
In the game, the player is able to engage in combat with other ships and colleges. A key aspect of this combat system is that both ships and buildings have health. For both Player and other Ships, this health value is managed by a Pirate component (FR_PLAYER_HEALTH), while buildings are designed to lose all health in a single hit. When a Building is hit and destroyed, this is effectively the College it is attached to losing health, as once all a College's buildings are destroyed, the College is captured. When a Ship receives damage, this is also managed by the Pirate component, when damage is to be dealt the Pirate's takeDamage() method may be called, which will decrease the relevant ship's health (FR_PLAYER_DAMAGE, FR_ENEMY_SHIP_DAMAGE). Additionally, the GameScreen will reference the Player Pirate's health value when displaying health onscreen. This is done in the update() method which is called every frame, meaning the value will be updated appropriately (FR_PLAYER_DAMAGE).

UR_GAME_LOSE
If the Player is reduced to 0 health, either as a result of combat or taking damage from an Obstacle, then the Player is killed (FR_PLAYER_DEFEAT). This results in the game reaching a fail state(FR_SCENARIO_FAIL) and the EndScreen() being called.

- **New Feature** - GameDifficulty - An Enum with the values EASY, REGULAR, HARD, used to define the chosen difficulty of the game.

UR_EARN_PLUNDER & UR_SPEND_PLUNDER - All Ships have a Pirate component, including the Player ship. This Pirate component stores and manages the Ship's 'plunder', which can be added to with the addPlunder() method (FR_PLUNDER_TRACKING). When the player completes a Quest, (which can include enemy encounters), they will receive 'plunder' as a potential reward. The QuestManager manages these Quests, and when it detects that they are complete it will call the Player Pirate's addPlunder(), giving the Quest's unique plunder reward value as the amount of 'plunder' to give the player, fulfilling FR_PLUNDER_UPDATE. The player can spend their plunder in the games shop. This is displayed on the bottom right corner of the main game screen and is defined in the main method of the GameScreen class, lines 118-235 and is set up using a Scene2d table. In this section of code, an instance of PowerUp for each power up available in the shop is made, and on click of a button, the PowerUp.buyPowerUp() method is called. This checks that the user has enough plunder to buy it, removes the plunder from the player, and calls AssignPowerUp on the PowerUpAssigned component of the player as shown in the concrete architecture and fulfils FR_PLUNDER_SPEND.

UR_EARN_XP - All Ships have an XP value, referred to within the code for readability as 'points'. This value is managed in the same way as 'plunder', being tracked by the Pirate component (FR_XP_TRACKING), and being assigned for completing Quests by the QuestManager (FR_XP_UPDATE). However in addition to this, 'points' are also awarded over time for playing, as this non-spendable value is intended as a metric to show player progress. In order to achieve this, the Player will assign its Pirate points every 1000ms during the update() method, using the addPoints() method of Pirate (FR_XP_UPDATE).

UR_QUEST_PROGRESS - The player will progress through a series of Quests throughout the duration of the game. To allow this, we made a Quest class along with a QuestManager to manage these Quests. Quest is an abstract class, which LocateQuest and KillQuest inherit from. We used this structure so that we could implement many different types of Quest, all while having them share core attributes and methods necessary for QuestManager to process them. A LocateQuest is associated with locating a specific Entity, in our case we used Chest (FR_QUEST_OBJECTIVE). Alternatively a KillQuest is associated with killing a particular College (FR_QUEST_OBJECTIVE). QuestManager generates all the Quests randomly when the game starts, with the method createRandomQuests() (FR_QUEST_RANDOMISE). QuestManager also tracks what the current Quest is (FR_QUEST_TRACKING), as well as determining when a Quest has been completed, and distributing the appropriate rewards. GameScreen will use QuestManager to get a description attribute of the current Quest, and display it on-screen for the player to see (FR_QUEST_TRACKING). Additionally if the current Quest is a KillQuest, an Indicator will display itself on-screen to point the player towards the location.

UR_OBSTACLE_ENCOUNTER - The player may encounter various obstacles while sailing in our game. We designed obstacles as an entity Obstacle, containing the component ObstacleControl. Using this design allowed us to both better organise code, as well as allowing us to give ObstacleControl to other classes should we want them to exhibit obstacle behaviours. During the GameManager's SpawnGame() method we iterate through the GameSettings.json file to find values for different types of obstacles and locations they could spawn in. This is then used to spawn in Obstacles randomly across the game (FR_OBSTACLE_SHOW). The ObstacleControl component allows Obstacles to deal damage in a variety of unique ways, based on several attributes. The damage attribute defines how much damage they deal to the player on hit, the hitRate attribute determines how often the damage will occur (if it repeats) and the hitLimit attribute determines how many hits may occur before the Obstacle breaks (if it can break). These implement FR_OBSTACLE_HIT.

UR_WEATHER_ENCOUNTER - The player may also encounter bad weather when sailing in the game. This is implemented using an Entity called Weather, which inherits from Obstacle. This allows Weather to use the damage properties of ObstacleComponent to its advantage, while also being unique compared to other Obstacles. Weather will constantly receive velocity in its update() method, which is calculated with its method newDir() which gives it a scalar direction it can multiply by its speed. This allows Weather to move randomly around the lake (FR_WEATHER_SHOW). In addition to new methods, Weather will also specify that its RigidBody component is a trigger, this means that other entities may pass through it, meaning it can pass over the Player while dealing damage, satisfying FR_WEATHER_HIT.

UR_POWER_UP - The player is able to obtain PowerUps through both PowerUpPickups on the map, as well as buying them through the shop. A PowerUp is a modifier which will be applied to a specific Pirate attribute (FR_POWER_UP), such as health, damage, plunderRate, etc. A PowerUp may have a duration, in which case it will only last a certain amount of time, or alternatively it may also be permanent, allowing a pseudo-implementation of upgrades to the Player. To manage applying this to the Player, a PowerUpAssigned component is necessary, which will hold the currently assigned PowerUp, as well as applying and removing values when necessary, and facilitating the swapping/removal of PowerUps when more are gained.

- **New Feature** - PowerUpOperation - An Enum with the values replace, increment, decrement, multiply and divide. Each representing a type of operation a power up could do to its chosen value. Eg a powerup with operation multiply may double the ammo of the player.

UR_DFCTLY_LVL - To implement the difficulty level, the game has a drop down box in the MenuScreen which on pressing play, changes the string in the drop down menu to an enum GameDifficulty and assigns it to the difficulty attribute in PirateGame. This means that it can be passed into GameManager.Initialise(difficulty) where GameManager.changeDifficulty(difficulty) is called before the entities of the game are created . This method opens a JSON file called settings.json and loads it into a JsonValue. The part of the file loaded depends on the difficulty level as this is where the values for different starting health, ammo, ship speed and enemy damage are stored and change depending on which difficulty is chosen. The entities then use these values as their initial values.

UR_GAME_SAVE - To implement the saving of the game state, as shown in the concrete architecture, the manager class SaveManager was added.. To allow the user to save their current progress in the game (FR_SAVE_GAME_STATE), the SaveManager.SaveGame() method is called upon clicking the 'Save+Exit' button in the pause screen.
The SaveGame() method systematically saves the state of each College and Ship in the game as well as the selected difficulty and the currentQuest..:

- It first loads the preferences file using Gdx.app.getPreferences("pirate/GameSave_game_1"). (creates a new one if already there or overwrites the values in it if it already exists)
- The first thing saved to the prefs file is the Difficulty.
- Then it iterates over the list of all ships in the game, where the player is the first in the list, adding the coordinates of the ship, the health, ammo, points, plunder and the faction it is assigned to.
- Then it fetches the list of all colleges with GameManager.getColleges(). Again it iterates over the ArrayList but this time only saves whether or not the College is alive.
- Lastly it saves the preferences file using prefs.flush()

To implement loading the game, (FR_GAME_LOAD), the menu screen calls the PirateGame.LoadGame() method on pressing 'resume ' which calls SaveManager.LoadGame() which gets the difficulty for the game first from the preferences file and tells the game manager to change it. It then calls SaveManager.SpawnGame()' which loads the list of ships and iterates through it, changing the values assigned to each ship in accordance with those loaded from the .prefs file. If the value is not saved in the file then it keeps the standard value. It then goes

through the list of colleges, calling .killThisCollege() on each college that is dead in the saved game.

Significant Changes to previous code

To implement some of the requirements we had to make the following changes to the code from the previous team:

PR #29, Initial Tests - In order to create tests for our implementation, we required the game to be run headlessly. In order to achieve this, we had to refactor code relating to rendering. We removed the tryInit() method from RenderingManager, so as to cause it to only Initialize() when we choose, as well as modifying addItem() to prevent this change from causing it to error. The game only renders if we call RenderingManager.Initialize(), so this change allowed us to call it in PirateGame when running the game usually, but not call it in tests to run it headlessly (2164768). We additionally had to modify GameManager.SpawnGame(), to allow it to be ran without triggering CreateWorldMap(), as creating a WorldMap will crash if RenderingManager.Initialize() has not been run (63790f0).

PR #51, Improvements - In order to implement UR_EARN_XP, many aspects of the code had to be refactored to include 'points'. Attributes, in addition to getters and setters, had to be added to Pirate. GameScreen had to be updated to display the value. Player had to be updated to award a 'point' every second in the update() event. Many Quests had to be given a reward value, with which QuestManager could use to supply 'points' upon completion. (60f5144)

PR #55, Adds Difficulties to game - To complete requirement UR_DFCLTY_LVL, we had to reorder the way that the game spawns. Previously upon loading the game, an instance of every screen, including GameScreen was made, which triggers GameManager.spawnGame() which loads the settings file and creates all entities. To make it so the player can choose a difficulty before this method is run, the load order had to be changed so that a new method StartGame() is triggered once the play button is pressed which creates the GameScreen and initialises the game.
(bc85009)

PR #57, PowerUp System - Each PowerUp (UR_POWER_UP) in the game modifies a different attribute of the Pirate component of a Ship, using string values stored in a JSON file to specify the attribute. Due to this, the relevant attributes of the Pirate component, had to be refactored from variables into values within a HashMap, to be accessed by the string value (e6e2ee9).

PR #57, PowerUp System & PR #65, Obstacles / Weather - When implementing both the PowerUp system (UR_POWER_UP) and the Obstacles system (UR_OBSTACLE_ENCOUNTER), GameManager.SpawnGame() had to be amended to also spawn in Obstacles and PowerUps. Additionally GameSettings.json had to be modified to include the values used by SpawnGame() when initialising the Obstacles. (f05d6e7), (5a97834)

PR #72 - We deleted any unused classes and commented out any unused methods. This was because these were not needed and removing them improves mainly organisation as well as efficiency.