Architecture

Stakeholders: Richard Paige, University of York Communications Office
Team: Barney Morgan, Cameron Smith, Harry Berge, Jake Phillips, Matthew Wilkie, Rob Weddell

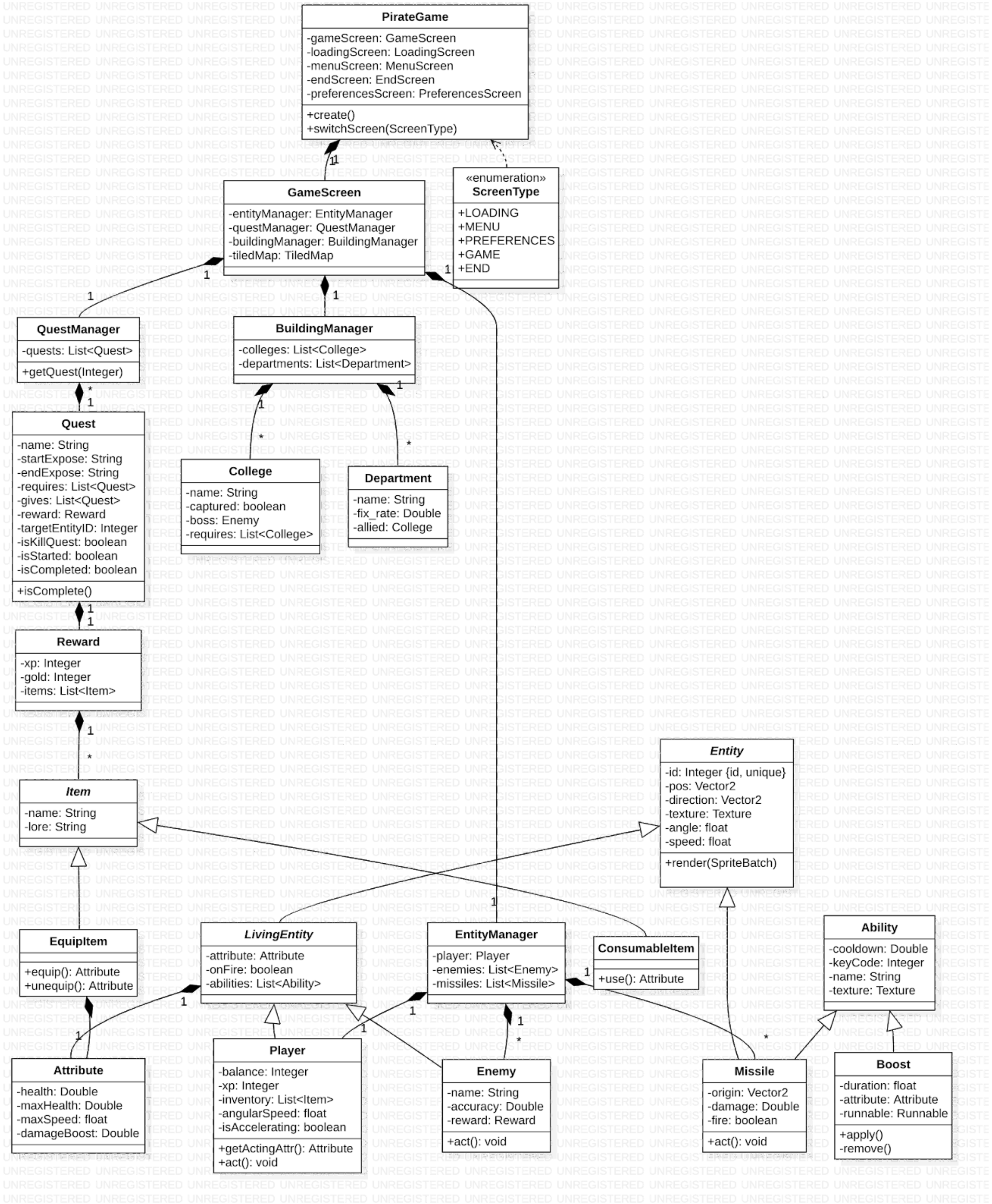
Proposed System Architecture

In order to plan the proposed system architecture outlined by the requirements [1], StarUML [2] was used to create a 2.x UML diagram [3]. This tool was chosen for its ability to provide high detail diagrams without much prior experience which was an important factor given that no team member had prior experience creating UMLs. However, not all aspects of the proposed architecture were suited to being displayed in the UML format. Consequently, flow charts were created with LucidChart [4] to represent the sequences of actions and game operations.

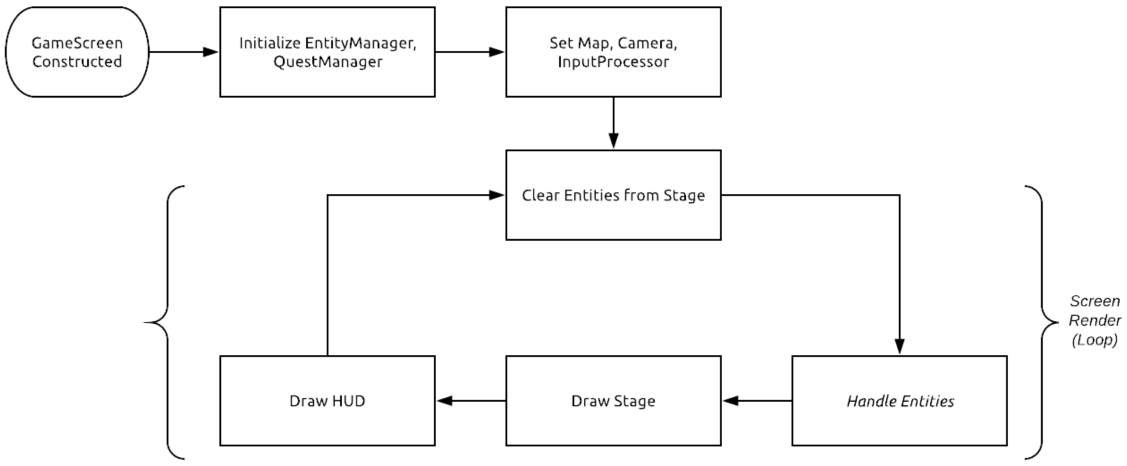
Subsequent to eliciting requirements, it was decided that thorough research was required in order to establish an en effective architecture. The research conducted involved evaluating frameworks, choosing the best solution for the projects needs and how it works. It was decided that LibGDX [5], an open-source development framework based on OpenGL [6], would be suitable for the project given its vast range of features. The main diagram [7] below specifies how the main features of the game will be implemented. Within this diagram getters, setters, and basic arithmetic operators have been omitted for brevity, however, the intention to use Project Lombok [8], a Java library that uses annotations to automatically generate repetitive structures such as getters and setters.

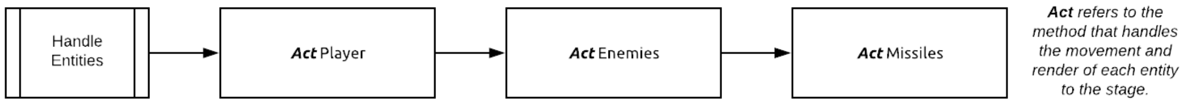
Each box, as seen below, represents a class which contains its variables and functions. Class, function or variable names in italics means that it is static. Additionally, there are many arrows used below to show relationships between classes. The dashed arrow depicts a dependency on the source class by the target (seen below, PirateGame relies on ScreenType). The white-filled arrow shows a class generalisation, that one class extends another: this can be seen where the Entity class is extended by Missle and LivingEntity classes. Finally, the black diamond shows composition, where one class is references and instances of it are stored in another. This diamond also has a symbol associated with the arrow denoting the multiplicity of the relationship: this symbol is either a number or '*' meaning '1 - n', for example, a single instance of the EntityManager class may store any number (* or n) of Enemy or Missle objects.

Java naming convention was used to create sensible names for variables and functions. The main class, PirateGame, will extend the LibGDX class 'Game' which is responsible for the initializing the game. LibGDX also defines 'Screen', a class that can be extended to allow custom pages. The class 'GameScreen' is responsible for the main portion of the game including loading the map, processing user input and continual render of game objects. As we intend to have multiple 'moving' objects in the game, there is an abstract 'Entity' class which extends LibGDX's 'Actor' and defines attributes and functions shared between these objects. The 'Entity' class is extended two further classes, 'Missile' and 'LivingEntity'. The 'Missile' class will be used to create instances of projectiles that will be fired on the map. The 'LivingEntity' class is itself an abstract class and is extended by two further classes 'Player' and 'Enemy' who will have access to their instance of the added 'LivingAttribute' class which holds information such as the health, speed and cooldown of the entity. The 'Missile', 'Player' and 'Enemy' classes will both override the method act()' as defined in LibGDX's 'Actor' class, this is called during the stage render (in GameScreen) to set the new position of the entity based on its current position and direction.


PirateGame
-gameScreen: GameScreen
-loadingScreen: LoadingScreen
-menuScreen: MenuScreen
-endScreen: EndScreen
-preferencesScreen: PreferencesScreen
+create()
+switchScreen(ScreenType)
«enumeration» ScreenType
+LOADING
+MENU
+PREFERENCES
+GAME
+END
GameScreen
-entityManager: EntityManager
-questManager: QuestManager
-buildingManager: BuildingManager
-tiledMap: TiledMap
QuestManager
-quests: List<Quest>
+getQuest(Integer)
BuildingManager
-colleges: List<College>
-departments: List<Department>
Quest
-name: String
-startExpose: String
-endExpose: String
-requires: List<Quest>
-gives: List<Quest>
-reward: Reward
-targetEntityID: Integer
-isKillQuest: boolean
-isStarted: boolean
-isCompleted: boolean
+isComplete()
College
-name: String
-captured: boolean
-boss: Enemy
-requires: List<College>
Department
-name: String
-fix_rate: Double
-allied: College
Reward
-xp: Integer
-gold: Integer
-items: List<Item>
Item
-name: String
-lore: String
Entity
-id: Integer {id, unique}
-pos: Vector2
-direction: Vector2
-texture: Texture
-angle: float
-speed: float
+render(SpriteBatch)
EquipItem
+equip(): Attribute
+unequip(): Attribute
LivingEntity
-attribute: Attribute
-onFire: boolean
-abilities: List<Ability>
EntityManager
-player: Player
-enemies: List<Enemy>
-missiles: List<Missile>
ConsumableItem
+use(): Attribute
Ability
-cooldown: Double
-keyCode: Integer
-name: String
-texture: Texture
Attribute
-health: Double
-maxHealth: Double
-maxSpeed: float
-damageBoost: Double
Player
-balance: Integer
-xp: Integer
-inventory: List<Item>
-angularSpeed: float
-isAccelerating: boolean
+getActingAttr(): Attribute
+act(): void
Enemy
-name: String
-accuracy: Double
-reward: Reward
+act(): void
Missile
-origin: Vector2
-damage: Double
-fire: boolean
+act(): void
Boost
-duration: float
-attribute: Attribute
-runnable: Runnable
+apply()
-remove()
1
1
1
1
1
1
1
*
1
1
*
*
1
1
1
*
1
1
1
1
1
1
1
1
*
*
UNREGISTERED

Below is a flowchart detailing the regular action (creation and render) of the GameScreen. As the game will have an event-driven architecture, there are few instances of decisions (polling) required.


GameScreen Constructed
Initialize EntityManager, QuestManager
Set Map, Camera, InputProcessor
Clear Entities from Stage
Handle Entities
Draw Stage
Draw HUD
Screen Render (Loop)


Handle Entities
Act Player
Act Enemies
Act Missiles
Act refers to the method that handles the movement and render of each entity to the stage.

The 'act' function is present in all entity classes and is an abstract method first implemented in LibGDX's 'Actor' class. This function takes a float representing the time delta between the last stage render and the current. This variable will be used to compute the distance that the entity has moved and as a result, its new position. Then follows test for collisions with other entities and map elements.

The LibGDX environment can be installed in your project by an executable jar file. This installation also includes the Gradle build tool [9] which is responsible for building and deploying the project. Gradle is an industry standard tool, this means it will be easy to run continuous integration for a later assessment with a tool such as TravisCI [10].

The flowchart [11] shows how the game will operate from the perspective of the player. As previously mentioned, the framework used is event-driven. However, the system is perhaps better understood when seen as a series of decisions. While this is not representative of the implementation, it helps to illustrate aspects of the game such as the quests and storyline.

System Architecture Justification

As is the nature of the preliminary stages of any project, a lot of research has been conducted into how the elicited requirements will actually be implemented. This research includes: setting up LibGDX, understanding the lifecycle of a game instance, and how to alter the look and feel.

Map

The first requirement (req 2.1) specifies that the game map will be recognisable as the University of York campus, thus a map will be designed using Tiled [12], a 2D level editor. Tiled (.tmx) maps can be compiled along with the project assets and used at runtime - the map file name is a final variable declared in the GameScreen class, seen in the UML diagram [7].

Controls

The method of controlling the player's ship forms another requirement (req 2.2): validated in a survey issued to the cohort [13], it was found that most customers preferred using a mix of mouse and keyboard (WASD). As a result, it was decided to use the mouse to aim the weapons and the keyboard (WASD) to move the boat. This will be controlled by the InputProcessor class which determines which buttons are pressed on the keyboard and also the location of the mouse (relative to the player). The movement will not be strictly tile-based as ships will have momentum and angular velocity so that they travel naturally and not necessarily in straight lines. This will aid with realism and allow the project to meet another requirement (req 2.6), whereby there is an element of skill to the game: the combat will be easier for those who are better practised.

Gameplay Objective

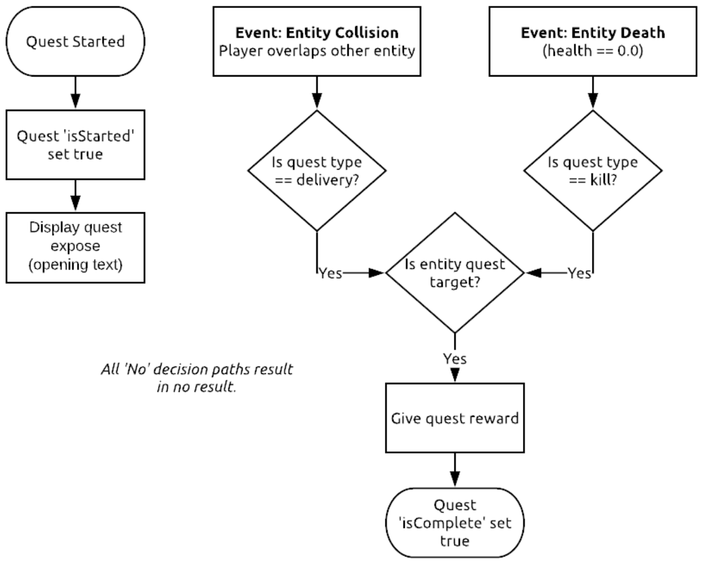
One of the more challenging requirements to implement will be the questing system (req 2.10 & 2.12). With this in mind, the classes have been designed to abstract this function into reusable and understandable segments so that each quest is not manually defined. When initialized, the QuestManager class creates and loads each quest into the quest list: these quests can then be retrieved at a later time. The Quest class itself holds data such as the type of quest (kill or deliver) and the reward to be given upon completion. See the flowchart [14] below for an example of the quest logic. An example quest might be to defeat a college boss and capture the college: in addition to the rewards given by the quest and dropped as loot by the boss, the player will gain access to allied departments.

Minigame

The requirements were clear that there should also be a minigame so that the player can have a break from the main portion of the game (req 2.14). After conducting a survey [11], it was found that there was no clear preference for whether this minigame had an effect on the player's progress of the main game. It was therefore decided to construct a plan around one of the preliminary ideas for the minigame which fits the pirate theme - gambling. Following this, it was proposed that a gambling mini-game would be available at the Departments across the map. This feature will only be possible when the player has defeated it's allied college.

Loot and Levelling

A player class, instantiated by the EntityManager, was then created which will hold details about the player's progress including XP and gold. This will allow the game to meet the requirement that specifies spending of earned riches (req 2.11 & 2.13). Gold can be spent at allied departments to repair the player's ship or game (as mentioned above). Additionally, to allow a unified solution to rewarding the player by completing quests and defeating enemies, it is planned to have a Reward class that can store the various types of reward in one object (XP, gold, items). Quests may manually specify the contents of a reward whereas when defeating enemies, rewards can also be randomly generated which will reduce the potentially repetitive nature of the game. See flowchart below [14] for basic quest implementation: please note, however, that the checks for quest completion will not be done per quest, but instead as a loop through each started quest when an event occurs.


Quest Started
Quest 'isStarted' set true
Display quest expose (opening text)
Event: Entity Collision
Player overlaps other entity
Event: Entity Death
(health == 0.0)
Is quest type == delivery?
Is quest type == kill?
Yes
Yes
Is entity quest target?
Yes
All 'No' decision paths result in no result.
Give quest reward
Quest 'isComplete' set true

References

[1] Element of SEPRise!, Requirements [Online]. Available: https://sepr4.github.io/web/submission/assessment1/Req1.pdf [Accessed 5th Nov. 2018].
[2] StarUML Website [Online]. Available: http://staruml.io/ [Accessed 19 Oct. 2018].
[3] UML Website [Online]. Available: http://www.uml.org/ [Accessed 12 Oct. 2018].
[4] LucidCharts Website [Online]. Available: https://www.lucidchart.com [Accessed 20 Oct. 2018].
[5] LibGDX Website [Online]. Available: https://libgdx.info/ [Accessed 5 Oct. 2018].
[6] OpenGL Website [Online]. Available: https://www.opengl.org/ [Accessed 5 Oct. 2018].
[7] Element of SEPRise!, Main UML Diagram [Online]. Available: https://sepr4.github.io/web/submission/assessment1/uml/Main.png [Accessed 5 Oct. 2018].
[8] Project Lombok [Online]. Available: https://projectlombok.org/ [Accessed 12 Oct. 2018].
[9] Gradle Website [Online]. Available: https://gradle.org/ [Accessed 12 Oct. 2018].
[10] TravisCI Website [Online]. Available: https://travis-ci.org/ [Accessed 3 Nov. 2018].
[11] Element of SEPRise!, Player-perspective Flowchart [Online]. Available: https://sepr4.github.io/web/submission/assessment1/flowchart/PlayerPers.jpeg [Accessed 19 Oct. 2018]
[12] Tiled Map Editor [Online]. Available: https://www.mapeditor.org/ [Accessed 19 Oct. 2018].
[13] Element of SEPRise!, Requirements Validation Survey [Online]. Available: https://sepr4.github.io/web/submission/assessment1/requirements/UserSurveyResults.pdf [Accessed 23 Oct. 2018].
[14] Element of SEPRise!, Quest Flowchart [Online]. Available: https://sepr4.github.io/web/submission/assessment1/flowchart/Quest.png [Accessed 12 Oct. 2018].