Software Architecture

Assessment 1 Architecture: https://engteam14.github.io/website2/pdfs/Arch1.pdf

Abstract architecture was created with draw.io with basic relationships of the program being shown on the diagram.

Concrete architecture was produced jointly by PlantUML and Adobe Photoshop. 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

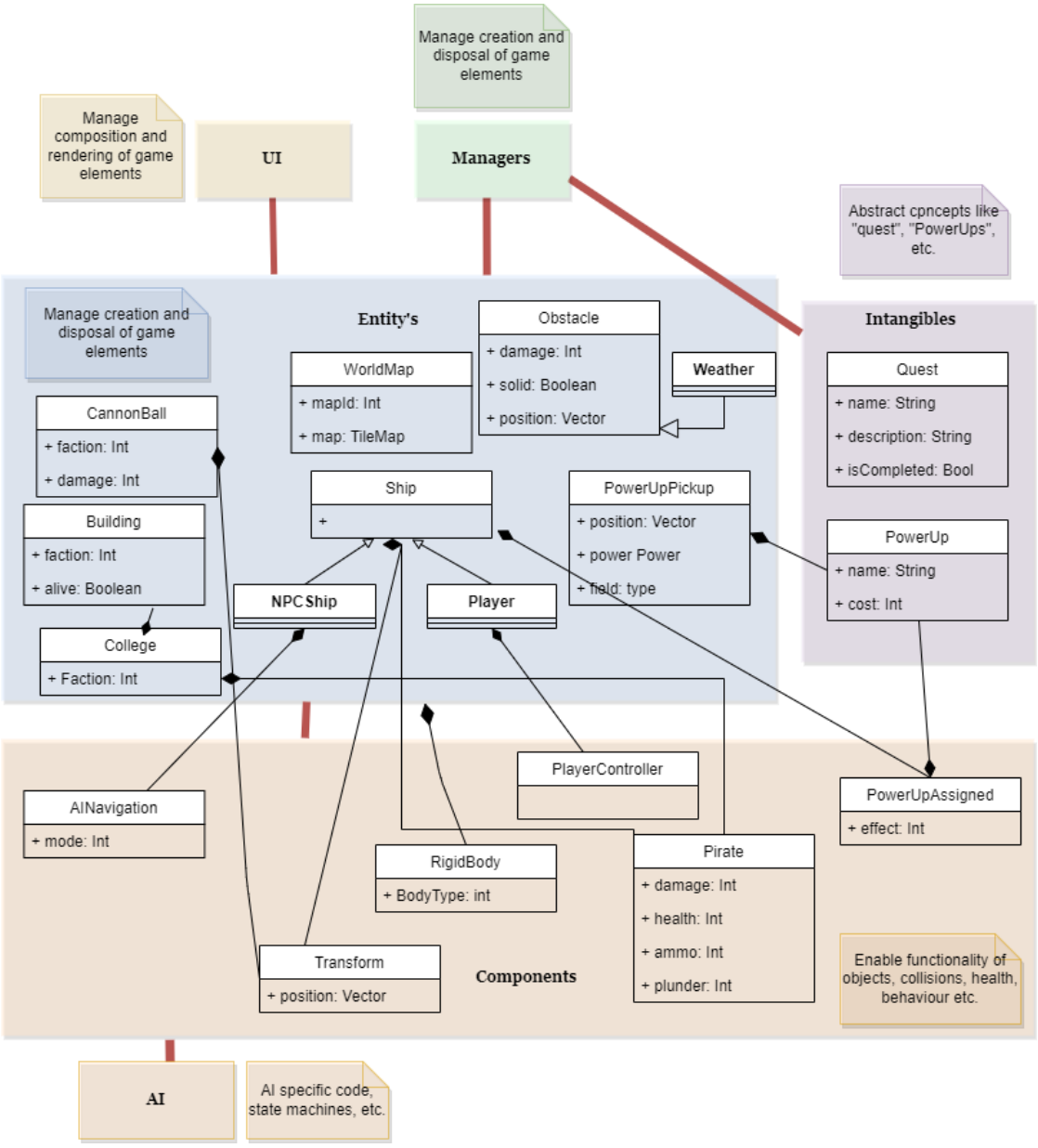

Fig 3.1.1: Diagram of the abstract architecture

Concrete Architecture

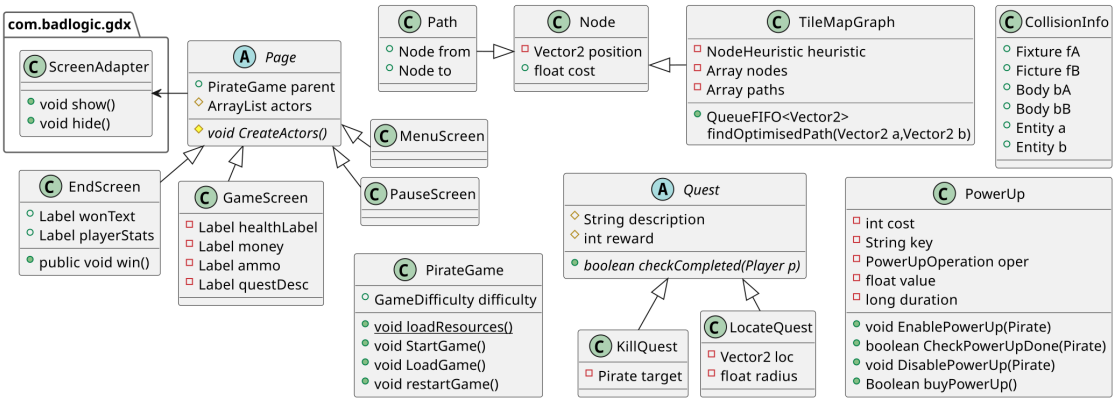

Fig 3.1.2: Diagram of miscellaneous classes

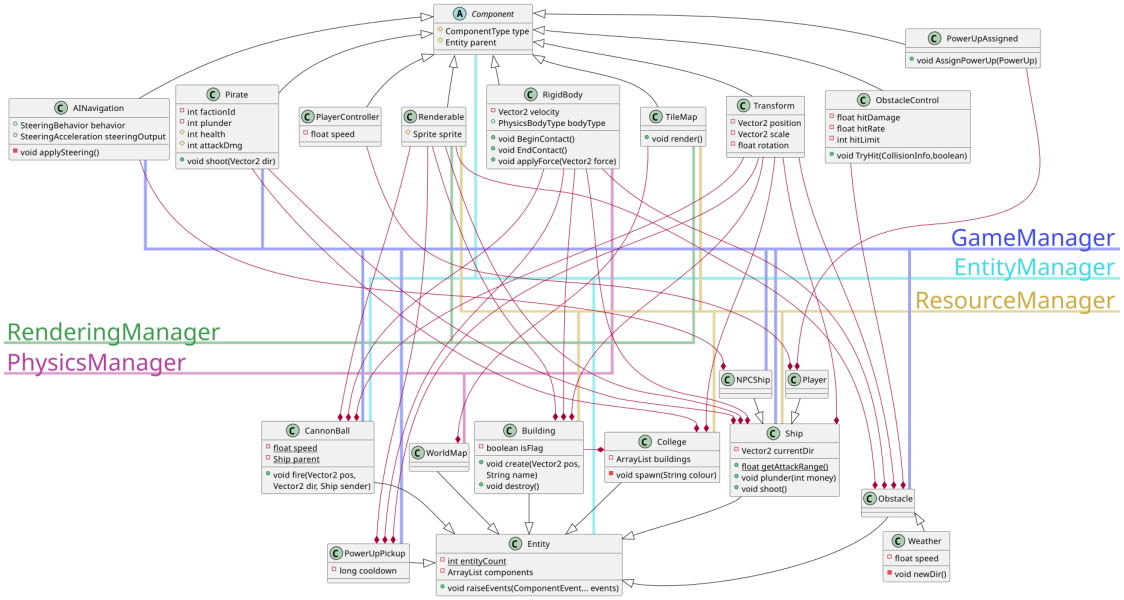

Fig 3.1.3: Diagram of Entity and Component classes

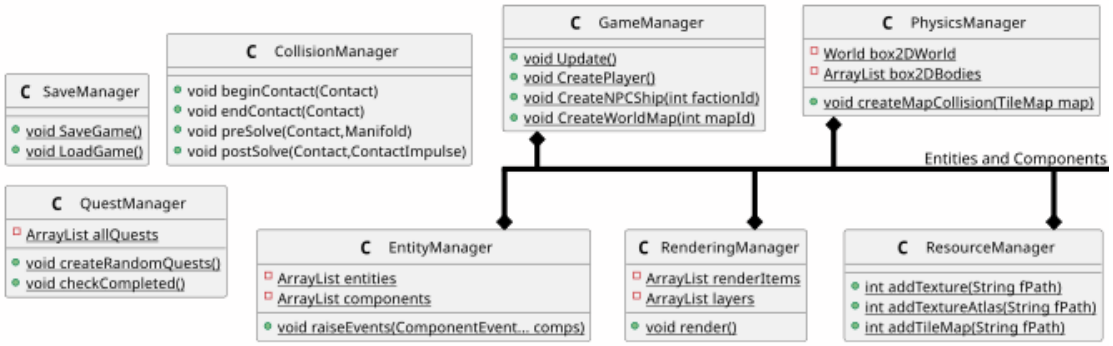

Fig 3.1.4: Diagram of Manager classes

3b)

The abstract architecture is concerned with decomposing the game into smaller logical elements which could be planned and reasoned about separately. Connections drawn between groups of elements signify a logical relationship rather than necessarily representing extension or composition relations (shown by red lines), the diagram also shows some important classes, their attributes, and Composition/Aggregation. Our abstract architecture mainly focuses on the entity-component relationships the game would need as well as some other intangibles and the concept of having managers which are logically linked to the entities. This enabled

us to have a base outline of the different classes needed in order to develop our concrete 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. It is important to note that the concrete architecture is not a thorough representation of the final implementation to ensure it is not overly detailed causing confusion.

`PirateGame` is the main class of the game which contains all the class instances of the game UI screens and also the methods for creating, starting, loading and restarting the game.

Entity Components

- We use an Entity-Component system to manage our game as shown in the abstract diagram. The `Entity` class represents each interactable object within the game world. Each `Entity` contains `Component`s, which are reusable patterns of code present throughout multiple entities. The most basic components are `Renderable` which defines how the entity is displayed and `Transform` which defines the entity's position within the game, which all entities other than `WorldMap` have.
  Renderable works in tandem with the `RenderingManager` which controls the `Camera` and makes sure all entities with attached `Renderable` components are drawn each update which helps satisfy the NFR_RENDER_SMOOTHNESS requirement. All the textures and sprites which are used throughout the game are loaded in and stored within the `ResourceManager`.
- All components inherit the `Component` class which was added into the concrete architecture as a base class for components to handle raising events and holds the `ComponentType` as an enum and its Entity parent, and `Entity` is inherited by all entities and was added for the same reason. This is all handled by the `EntityManager` which tells components to `raiseEvents` and holds the ArrayLists of `entities` and `components`.

Physics

To implement physics we added the `Rigidbody` components and a `PhysicsManager`. The `Rigidbody` component is used to define the collision fixtures of the object and is a component of `CannonBall`, `PowerUpPickup`, `Buidling`, `Ship` and `Obstacle` because they all need to colide/be collidable.
The `PhysicsManager` creates and keeps track of collidable terrain hitboxes. The `PhysicsManager` was not explicitly defined in the abstract as it was assumed that the game engine would handle all physics but due to the nature of LibGdx, the `PhysicsManager` had to handle it.
The collisions themselves are handled by `CollisionManager` which defines and handles the behavious for all different types of collisions. This is so that different collisions can have different effects (i.e some dealing damage, some having a knockback effect etc) in accordance with requirements:
NFR_WORLD_COLLISIONS, NFR_SHIP_COLLISIONS, and NFR_BULLET_COLLISIONS.

Ships

Ships are instantiated by `GameManager`. Two classes inherit the `Ship class`, `NPCShip` and `Player` in both the abstract and concrete architectures. This means that the two different types of ships can have different components due to the different functionality needed but also inherit some attributes and methods from `Ship`.
`Ship` has the `Pirate` component which gives all ships including the player a damage value, health, and plunder as well as a faction (Fulfils UR_EARN_PLUNDER/XP and) but only the NPCShip has the `AINavigation` component to fulfil FR_FRIENDLY_AI and FR_HOSTILE_AI while `Player` has the `PlayerControl` component to fulfil FR_PLAYER_FIRE, UR_FIRE_WEAPONS, UR_BULLET_DODGE and UR_SHIP_CONTROL as it allows the player to move their ship and fire.

Power-Ups

We separated PowerUp functionality into its own class, `PowerUp` in both abstract and concrete due to the two different ways of gaining PowerUps, buying (UR_SPEND_PLUNDER) and picking up (UR_POWER_UP). This class can then be applied to a ship using the `PowerUpApplied` component. `PowerUp`s in the game world are stored under the `PowerUpPickup` entities, which are spawned in by the `GameManager` when the game starts. `PowerUp`s bought in the shop do not need a new class as they are purely data.

Obstacles & Weather

Obstacles are entities that have the `obstacleControl` component as well as the standard components mentioned above. This assigns a `hitDamage` and `hitRate` to each obstacle and deals damage to the player (UR_OBSTACLE ENCOUNTER). `Weather` inherits `Obstacle` and adds functionality for moving around the map. This was designed from the outset in abstract architecture due to the similar functionalities of obstacles and weather.

Cannonballs

The `CannonBall` class represents the cannonballs that the ships and college buildings can fire which can collide with other entities, causing damage. .(UR_FIRE_WEAPONS, UR_HOSTILE_BUILDING_COMBAT & UR_SHIP_COMBAT). The collisions are registered with `CollisionManager` and moved with `PhysicsManager`. This class is defined in the abstract architecture.

Colleges

A `College` has a `Pirate` component which was added to enable shooting and assign loot and health. Each instance of `College` also has `Building` objects attached to it. UR_COMPETING_COLLEGES and UR_HOSTILE_COLLEGE_CAPTURE. `College` also gained the `Transform` Component in the concrete architecture as we decided it would be useful to allow the college to be placed in specific places that could be moved easily if the map changes in development.

UI

In our abstract diagram, we had a general UI group but upon researching LibGdx we found that each UI screen is implemented with a separate class which must inherit `ScreenAdapter` which is a libGdx defined class with a standard method for `update()`, `render()` and more. In our concrete diagram, the different screens actually inherit a class defined by us called `Page` which itself inherits ScreenAdapter. This is so that Page can create actors to be displayed which will be the same process across all screens. The `PauseScreen` and `MenuScreen` screen helps to implement UR_GAME_SAVE and also the `Pause` class also implements FR_GAME_RESET and `EndScreen` displays a different `wonText` depending on whether the player won or lost the game.

Quests

Quests are implemented by having a main `Quest` which is the parent class of the quest types and has the method for checking is completed as well as important attributes. `KillQuest` and `LocateQuest` both inherit the `Quest` class and represent different types of quests. (FR_QUEST_OBJECTIVE). In the abstract, the `Quest` class was a standalone Intangible but due to the slightly different implementation needed for killing vs picking up, we created the child classes. Furthermore, a `QuestManager` was added to handle the creation of

tests and to check through all quests to see if they are completed or not. This fullfills FR_QUEST_RANDOMISE, UR_QUEST_PROGRESS, UR_GAME_WIN and FR_QUEST_TRACKING.

AI

`NPCShip` has the component `AINavigation` which gives the AI ship entities the ability to use the AI algorithm to move around the map in our abstract architecture. This is the same way that it was implemented in our concrete architecture, however instead of having an `Int` representing the `mode` that the ship is in, it has a `SteeringBehavious` and `SteeringAcceleration` which is a type of vector defined by the LibGdx Game Engine which was added in to simplify the implementation so that we could use predefined functions from LibGdx in the movement of our entities.
Once behaviours are set, they are passed on to the `AINavigation` Component attached to the ships which use a path made up of `Node`s to control the entity's movement. These classes all came under the AI group in the abstract architecture as we were hoping the game engine we chose would have a way of dealing with this but unfortunately libGDX does not. The AI classes fulfil the objective NFR_AI_LAG and UR_FRIENDLY_SHIP_ENCOUTNER, FR_FRIENDLY_AI, FR_HOSTILE_AI and UR_HOSTILE_SHIP_ENCOUNTER

Saving

The `SaveManager` class is the class that enables UR_SAVE_GAME. The `SaveGame` method fullfills FR_SAVE_GAME_STATE and the `LoadGame` methods fulfill FR_GAME_LOAD. Again this came under the Managers heading in the abstract diagram but it wasn't until we chose the game engine that we could decide exactly how this would be implemented.