ENG1 Team Project
Assessment 2

Deliverable 3:Architecture

Team 13 – Team Unlucky

LILLY BROWN
ROSCOE GILLATT
ADAM JOHNSON
YUMIS ZYUTYU
BRANDON WEST
AYMAN ZAHIR

Software Architecture

Abstract and Concrete architectures were 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

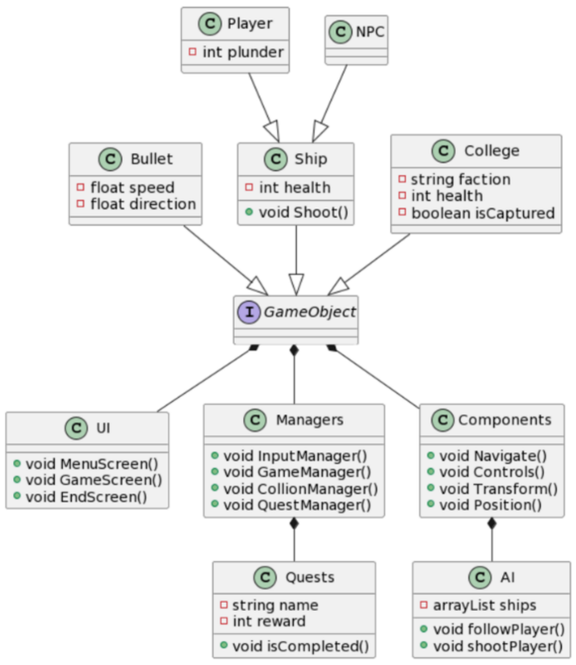

Fig 3.1.1: Diagram of the abstract architecture

Concrete Architecture

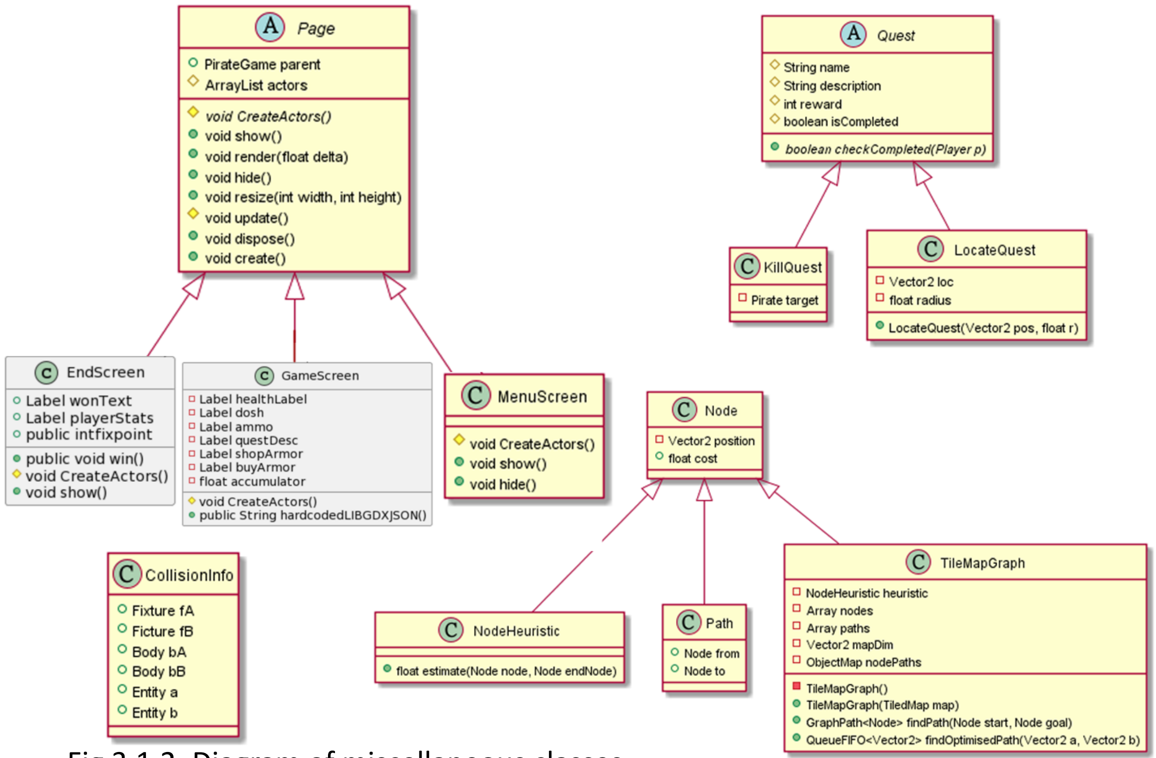

Fig 3.1.2: Diagram of miscellaneous classes

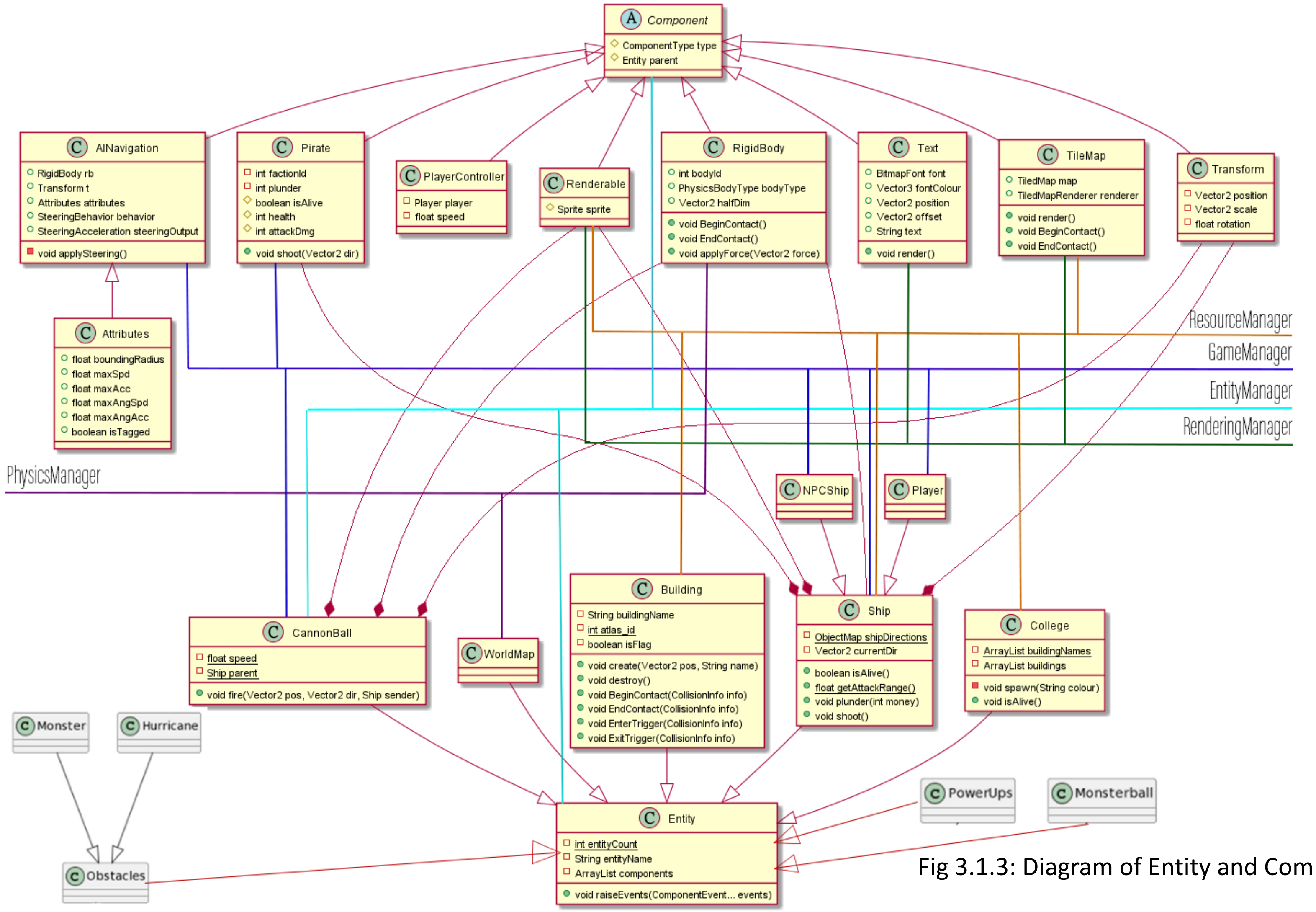

Fig 3.1.3: Diagram of Entity and Component classes

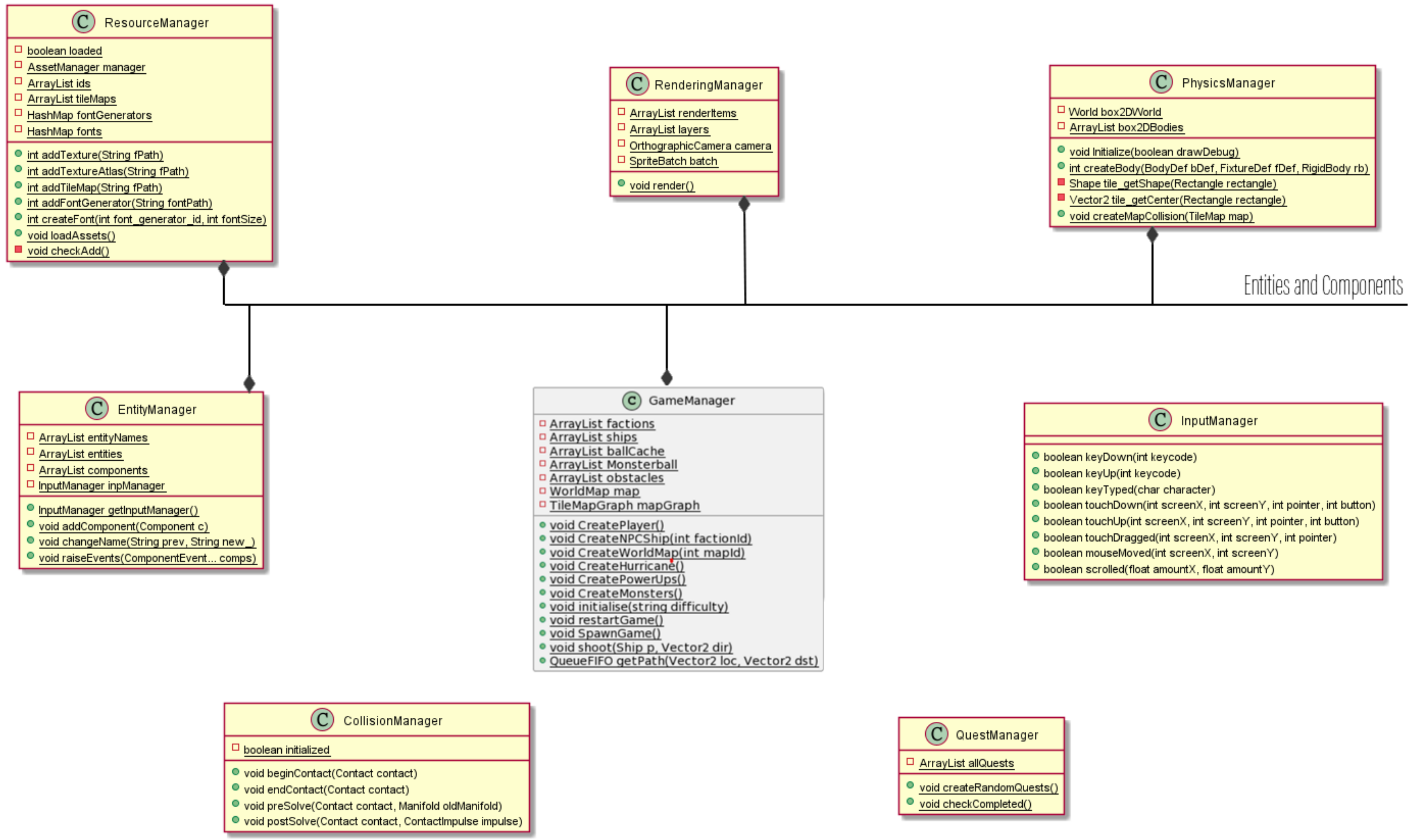
C ResourceManager
boolean loaded
AssetManager manager
ArrayList ids
ArrayList tileMaps
HashMap fontGenerators
HashMap fonts
int addTexture(String fPath)
int addTextureAtlas(String fPath)
int addTileMap(String fPath)
int addFontGenerator(String fontPath)
int createFont(int font_generator_id, int fontSize)
void loadAssets()
void checkAdd()
C RenderingManager
ArrayList renderItems
ArrayList layers
OrthographicCamera camera
SpriteBatch batch
void render()
C PhysicsManager
World box2DWorld
ArrayList box2DBodies
void Initialize(boolean drawDebug)
int createBody(BodyDef bDef, FixtureDef fDef, RigidBody rb)
Shape tile_getShape(Rectangle rectangle)
Vector2 tile_getCenter(Rectangle rectangle)
void createMapCollision(TileMap map)
Entities and Components
C EntityManager
ArrayList entityNames
ArrayList entities
ArrayList components
InputManager inpManager
InputManager getInputManager()
void addComponent(Component c)
void changeName(String prev, String new_)
void raiseEvents(ComponentEvent... comps)
C GameManager
ArrayList factions
ArrayList ships
ArrayList ballCache
ArrayList Monsterball
ArrayList obstacles
WorldMap map
TileMapGraph mapGraph
void CreatePlayer()
void CreateNPCShip(int factionId)
void CreateWorldMap(int mapId)
void CreateHurricane()
void CreatePowerUps()
void CreateMonsters()
void initialise(string difficulty)
void restartGame()
void SpawnGame()
void shoot(Ship p, Vector2 dir)
QueueFIFO getPath(Vector2 loc, Vector2 dst)
C InputManager
boolean keyDown(int keycode)
boolean keyUp(int keycode)
boolean keyTyped(char character)
boolean touchDown(int screenX, int screenY, int pointer, int button)
boolean touchUp(int screenX, int screenY, int pointer, int button)
boolean touchDragged(int screenX, int screenY, int pointer)
boolean mouseMoved(int screenX, int screenY)
boolean scrolled(float amountX, float amountY)
C CollisionManager
boolean initialized
void beginContact(Contact contact)
void endContact(Contact contact)
void preSolve(Contact contact, Manifold oldManifold)
void postSolve(Contact contact, ContactImpulse impulse)
C QuestManager
ArrayList allQuests
void createRandomQuests()
void checkCompleted()

Fig 3.1.4: Diagram of Manager classes

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, unlike what fig. 3.1.1 seems to suggest. Nevertheless, it is useful to see them grouped under intangibles while planning the overall architecture.

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, 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, figs 3.1.3 & 3.1.4 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.

Concrete Architecture Justification:

Architecture construct	Requirement ID	How it is fulfilled
Page	FR_VIEWPORT _SCALING	Page class has a class resize() which takes the width and height of the display or window, thus being able to render the game on displays with different sizes.
InputManager	FR_SHIP_KB_INPUT	InputManager has numerous functions in it which accepts keyboard signals from the user for ship navigation.
Player, Ship, Pirate, GameManager, MenuScreen	FR_PLAYR_FIRE/AMMO FR_MONEY_TRACKING FR_MONEY_UPDATE FR_POINTS_TRACKING FR_POINTS_UPDATE	Within the Ship class there are functions such as shoot() which allows the player to shoot when it's called in GameManager. The Pirate Component has getter and setter functions such as addPlunder() and getPoints() which the extended Player class utilises to track and update ammo, points and plunder.
MenuScreen, QuestManager, GameManager	FR_QUEST_TRACKING FR_QUEST_ RANDOMISE FR_QUEST_OBJECTIVE	In Quest and Quest manager, there are methods called checkCompleted() and createRandomQuests(), which check for quest completion,hence tracking boss unlocking, and also randomly generate quests linked with the college entities.
GameManager	FR_COLLEGE _ENTITY_TRACKING	GameManager class has methods to initialise and spawn all the game Entities within their own ArrayLists; Initialize() and GameSpawn() respectively.
GameManager, AI, Entity Pirate, NPCShip	FR_FRIENDLY_AI FR_FRIENDLY_ INTERACT FR_HOSTILE_AI	The AI package contains multiple classes and functions such as Path class which help AI navigation along with generic functions for shooting and collisions which EnemyState class inherits from Pirate via NPCShip.
GameManager, ExitScreen	FR_GAME_RESET	GameManager contains the restartGame() method which manually resets all the entity attributes such as ship positions and college buildings. It is called by EndScreen when the Restart button is clicked.
GameManager, MenuScreen	FR_GAME_DIFFICULTY	GameManager has the Initialize() method which takes in the file name of the Json file associated with the difficulty settings as an argument and initialises the game with those settings. The difficulty is chosen with buttons added in the MenuScreen.
QuestManager, GameScreen, ExitScreen	FR_GAME_WIN FR_PLAYER_DEFEAT FR_SCENARIO_FAIL	GameScreen's update() method calls the QuestManager's checkCompleted() method to check if all quests are completed then sets screen to the EndScreen() with win() to display winner text. Similarly when player health or ammo depletes it switches to EndScreen() and displays the default loser EndScreen.
GameScreen, GameManager, External files	FR_SAVE_POINT	The GameScreen has the hardcodedLIBGDXJSON() function which returns a JSON string with updated game settings by utilising getters to be written to an existing save file.
AI, Entity, Monster, Obstacle	UR_WEATHER_ ENCOUNTER, UR_OBSTACLE_ ENCOUNTER	The AI package contains the MonsterState() class which implements the monster AI. Both weather and monster functionality are defined under Entity()'s Obstacle and Monster classes respectively