3. Architecture

Team 10 | YorKorsairs

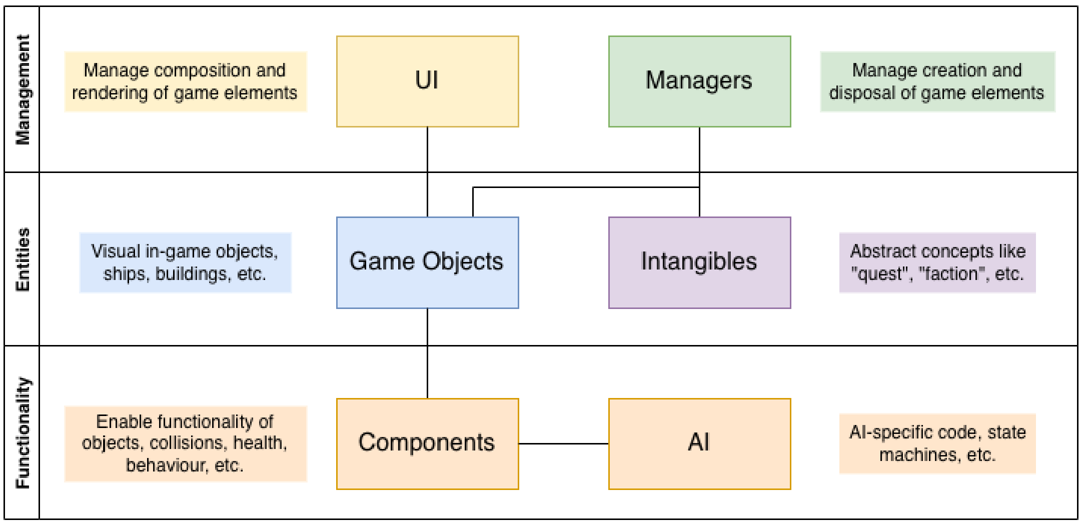
Abdullah
Tom Burniston
Omer Gelli
Adam Kirby-Stewart
Sam Mabbott
Benjamin Stevenson

A. Representation

Software Architecture

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 for efficient representation of object information.

Abstract Architecture



Management
Manage composition and rendering of game elements
UI
Managers
Manage creation and disposal of game elements
Entities
Visual in-game objects, ships, buildings, etc.
Game Objects
Intangibles
Abstract concepts like "quest", "faction", etc.
Functionality
Enable functionality of objects, collisions, health, behaviour, etc.
Components
AI
AI-specific code, state machines, etc.

Fig 3.1.1: Diagram of the abstract architecture

Concrete Architecture

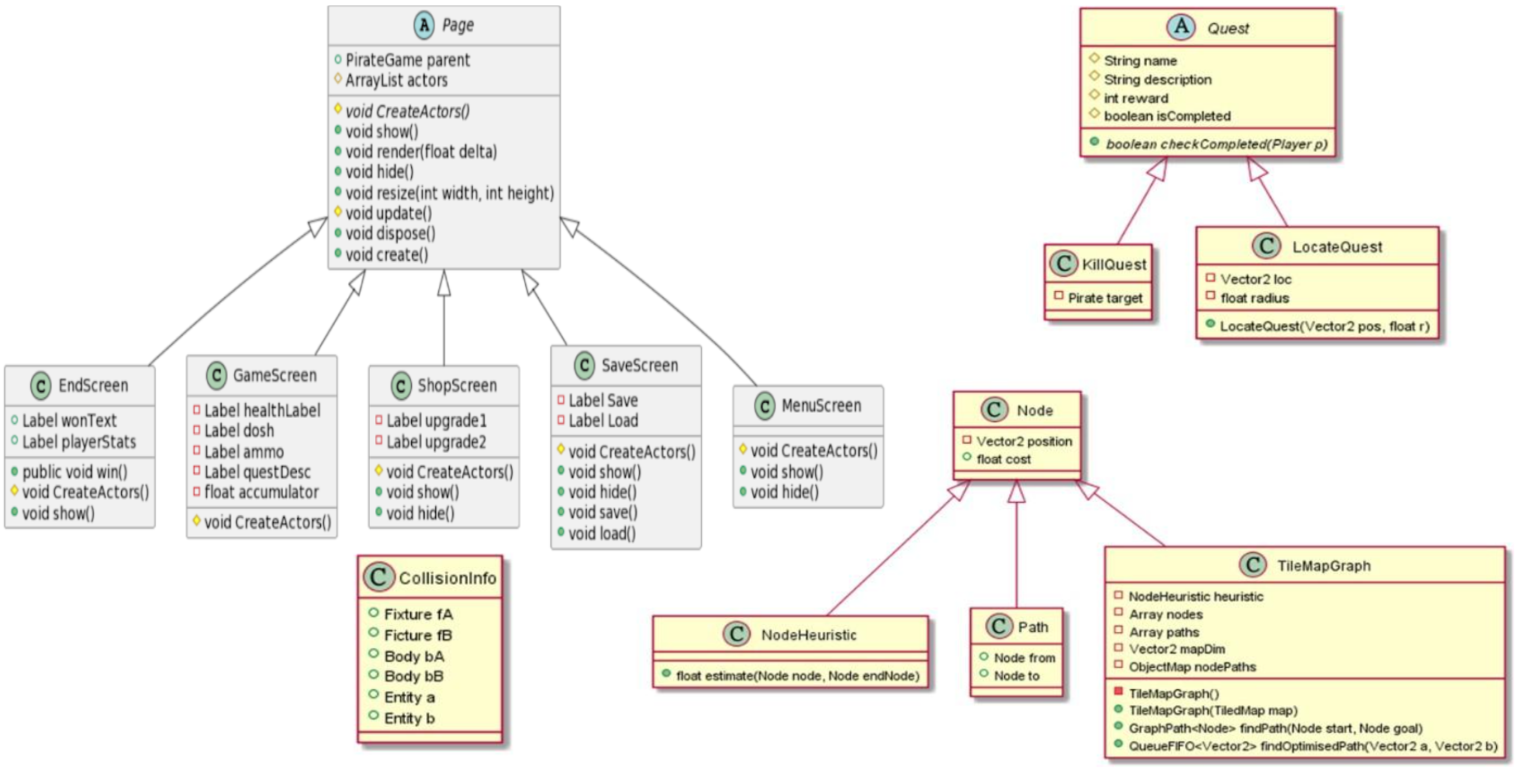


Fig 3.1.2: Diagram of miscellaneous classes

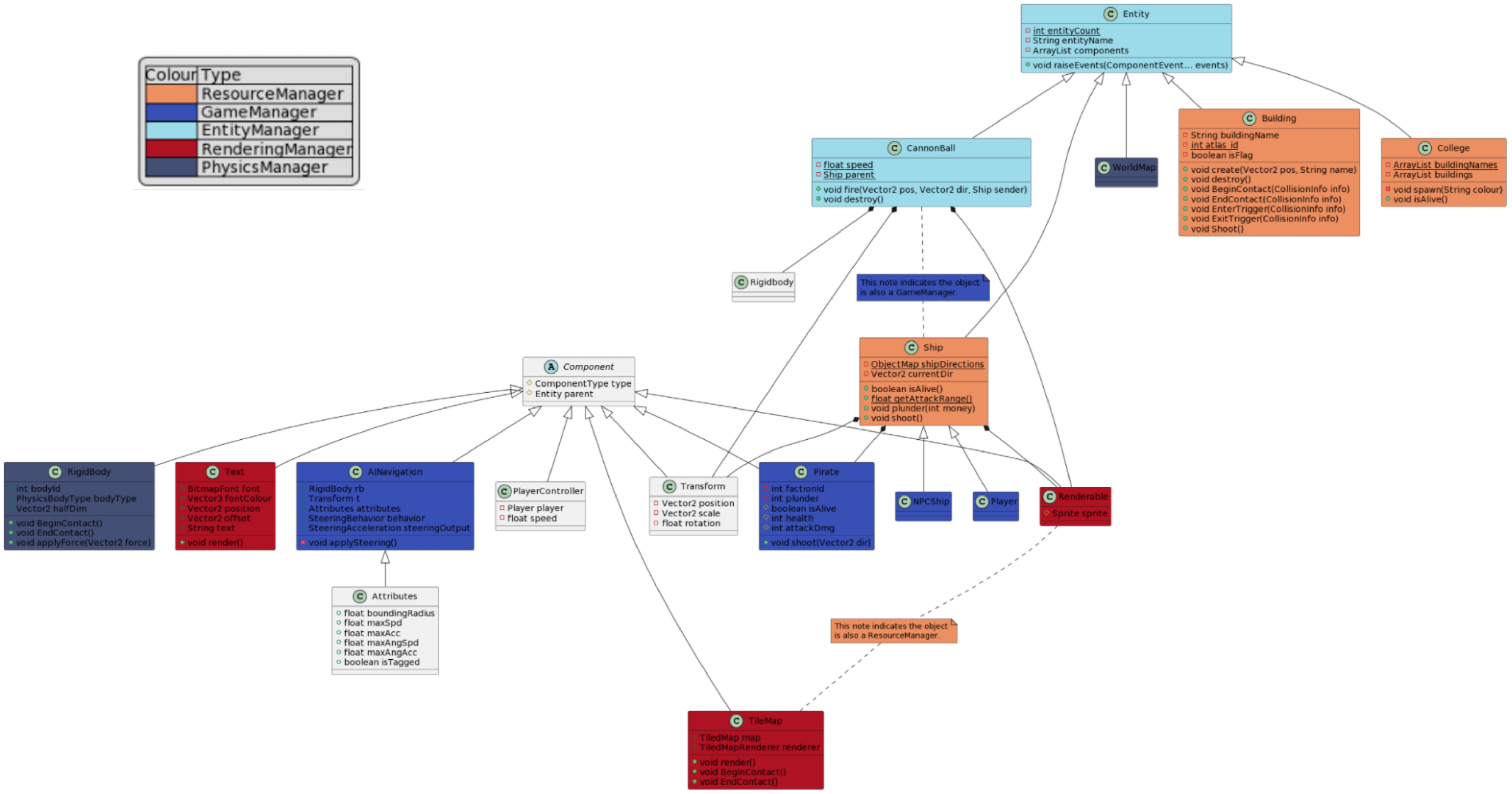


Fig 3.1.3: Diagram of Entity and Component classes

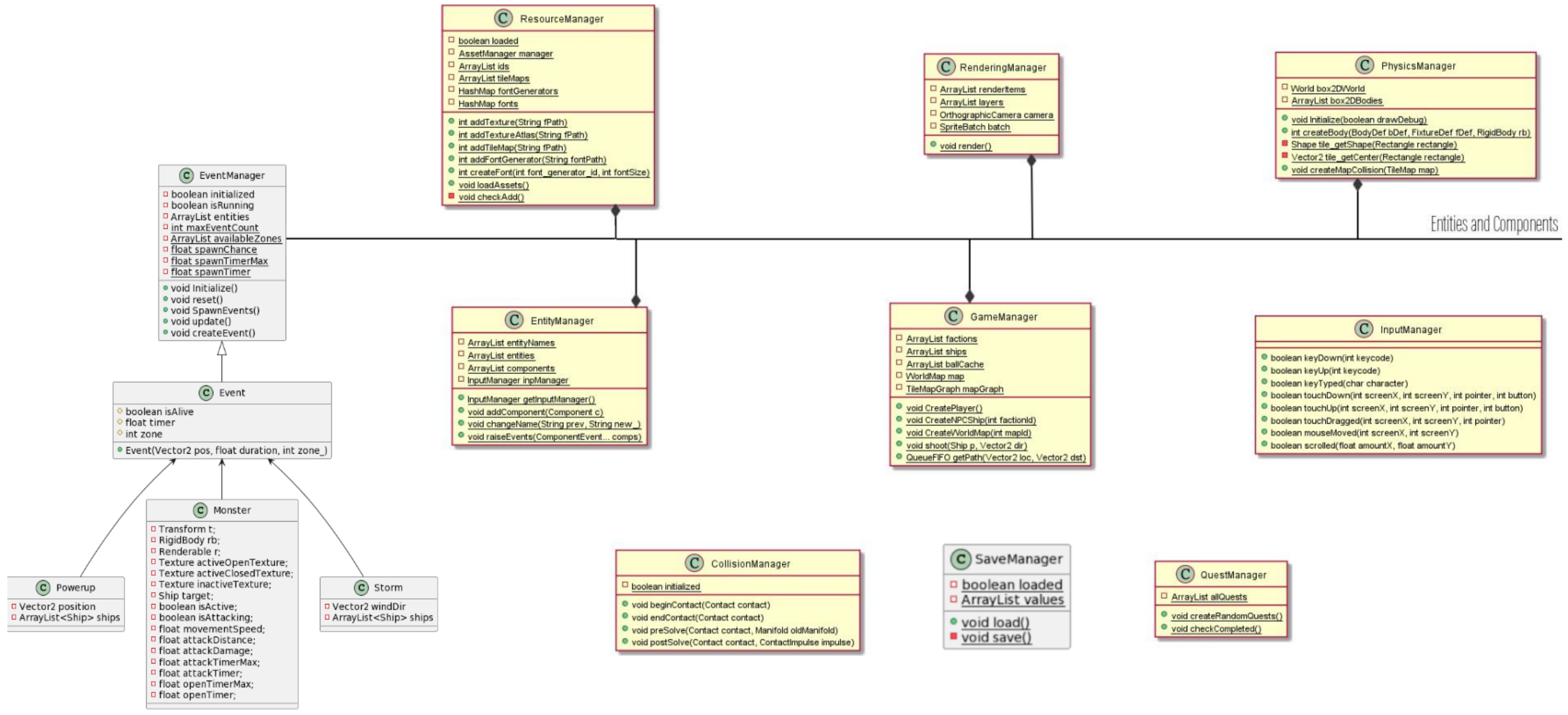

ResourceManager
boolean loaded
AssetManager manager
ArrayList ids
ArrayList tileMaps
HashMap fontGenerators
HashMap fonts
int addTexture(String fPath)
int addTextureAtlas(String fPath)
int addTileMap(String fPath)
int addFontGenerator(String fontPath)
int createFont(int font_generator_id, int fontSize)
void loadAssets()
void checkAdd()
RenderingManager
ArrayList renderItems
ArrayList layers
OrthographicCamera camera
SpriteBatch batch
void render()
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
EventManager
boolean initialized
boolean isRunning
ArrayList entities
int maxEventCount
ArrayList availableZones
float spawnChance
float spawnTimerMax
float spawnTimer
void Initialize()
void reset()
void SpawnEvents()
void update()
void createEvent()
Event
boolean isAlive
float timer
int zone
Event(Vector2 pos, float duration, int zone_)
Powerup
Vector2 position
ArrayList<Ship> ships
Monster
Transform t;
RigidBody rb;
Renderable r;
Texture activeOpenTexture;
Texture activeClosedTexture;
Texture inactiveTexture;
Ship target;
boolean isActive;
boolean isAttacking;
float movementSpeed;
float attackDistance;
float attackDamage;
float attackTimerMax;
float attackTimer;
float openTimerMax;
float openTimer;
Storm
Vector2 windDir
ArrayList<Ship> ships
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)
GameManager
ArrayList factions
ArrayList ships
ArrayList ballCache
WorldMap map
TileMapGraph mapGraph
void CreatePlayer()
void CreateNPCShip(int factionId)
void CreateWorldMap(int mapId)
void shoot(Ship p, Vector2 dir)
QueueFIFO getPath(Vector2 loc, Vector2 dst)
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)
CollisionManager
boolean initialized
void beginContact(Contact contact)
void endContact(Contact contact)
void preSolve(Contact contact, Manifold oldManifold)
void postSolve(Contact contact, ContactImpulse impulse)
SaveManager
boolean loaded
ArrayList values
void load()
void save()
QuestManager
ArrayList allQuests
void createRandomQuests()
void checkCompleted()

Fig 3.1.4: Diagram
of Manager classes

B. Justification and relations to requirements

1. 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.
2. 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.
3. 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.
4. 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.
5. 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. Having the relationships between entities & components and their respective managers depicted in a shorthand form means the information presented is encompassing yet succinct and is nevertheless clear and informative.
6. 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. Because of this specific Javadocs references were made which detail this.

Some key relations to requirements:

- *FR_SHIP_KB_INPUT -* By referring on the InputManager in fig 3.1.4, there are functions in it which accepts keyboard signals from the user for ship navigation
- *FR_VIEWPORT_SCALING - This function has now been implemented correctly. The previous team did indeed make reference to and tried to implement this, it did not work but now does.* By referring to the Page class on fig 3.1.2, there is 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.
- *FR_PLAYER_FIRE -* Referring to the Ship class in fig 3.1.3, there is a function called shoot which is called the same function in the GameManager in fig 3.1.4, which allows users to fire weapons.

- *FR_BULLET_TRAVEL* - Referring to CanonBall class in fig 3.1.3, the method fire() takes the starting position, direction and the sender ship, thus it shows the travel of the munitions sent from ships.
- *FR_QUEST_TRACKING / FR_QUEST_RANDOMISE* - In the Quest class of fig 3.1.2 and Quest manager of fig 3.1.4, there are methods called checkCompleted() and createRandomQuests(), which showed the game can track on player's quest completions and also randomise quest objectives.
- *NFR_WORLD_COLLISIONS* - There are multiple classes and methods which are responsible for world collisions. In PhysicsManager(fig 3.1.4), there is a method createMapCollision() which is responsible for creating zones which can be collided into. In both Building and TileMap classes (fig 3.1.3), there are methods called BeginContact() and EndContact() which process the collision of entities in the game. And data of two entities colliding will be stored in class Collisioninfo of fig 3.1.2.
- *UR_PLATFORM and FR_VIEWPORT_SCALING* - the Game and Update classes the requirement to allow for multiple viewers and the ability to play the game on multiple screens has led to the implementation of architecture which renders and resizes the UI for the best experience.

Some new key justification for architecture and improvements:
Due to the assessment's updated requirements, multiple screens are needed in game and as such a system for swapping between and handling multiple screen states for menu / game etc. will be necessary; the game can be handled by individual classes formed from the Page superclass. Variables are encapsulated by being made private, and providing public getter / setter functions when necessary. This means that they cannot be changed or accessed without the class's knowledge, allowing for maintenance of state.
Another feature we added to the game was that of health for both the colleges and the player. This was done by adding private attributes to the Player and College class. Adding this to the previous team's project will mean there will be visual feedback for players on the health of the ship and enemies as well as passed through the game class to check for changes to the Objectives or GameState such as the health of a college or the player reaching 0. The updated GameState or objectives will be shown to the player via updated visuals or splash screens.
Another major architectural point concerns collisions (managed by the Collisions Manager). This is a function that checks a collision box against everything that can be hit and damaged. This would also mean that there would need to be an hittable interface, which Player and College can implement that would check the difference between player friendly and enemy objects. This part is fundamental and many other classes rely on it and can be easily identifiable as an important part of the game and resultantly this architectural formatting concerning this class is justified.
The use of managers was kept when the assessment was handed over. In addition a new architectural class has been created, that of the EventManager and associated child classes. This was necessary for the requirements of assessment 2 namely the pickup, weather and monster obstacles. As justification, this architectural point was a requirement and kept in keeping with the style of the rest of the existing code as well as, looking more in depth the use of superclass means passing of functions and values is efficient, manageable and identible.
AI for the colleges & ships was expanded to include the primary logic for detecting the distance to the player as well as the direction to the player as well as an improved fighting, distancing and following.