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 WPS Office. The classes were separated into different categories with the connections within the category shown on the diagram. The PlantUML Diagrams were established by running code in Visual Studio Code with a PlantUML Extension. The inter-category connections are later added through PDF editing in WPS Office with lines colour coded for easier understanding.

Abstract Architecture

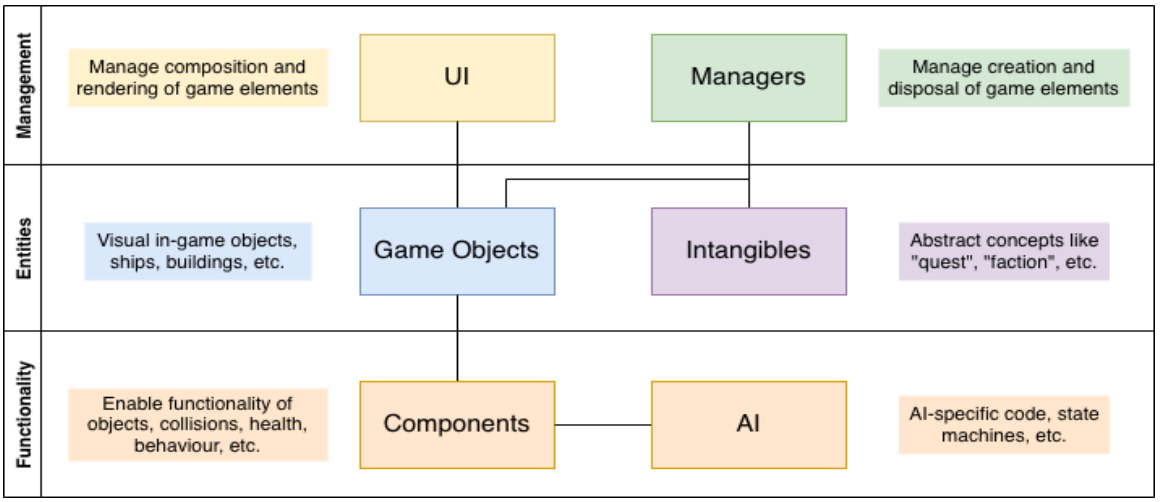

Fig 3.1.1: Diagram of the abstract architecture

Concrete Architecture

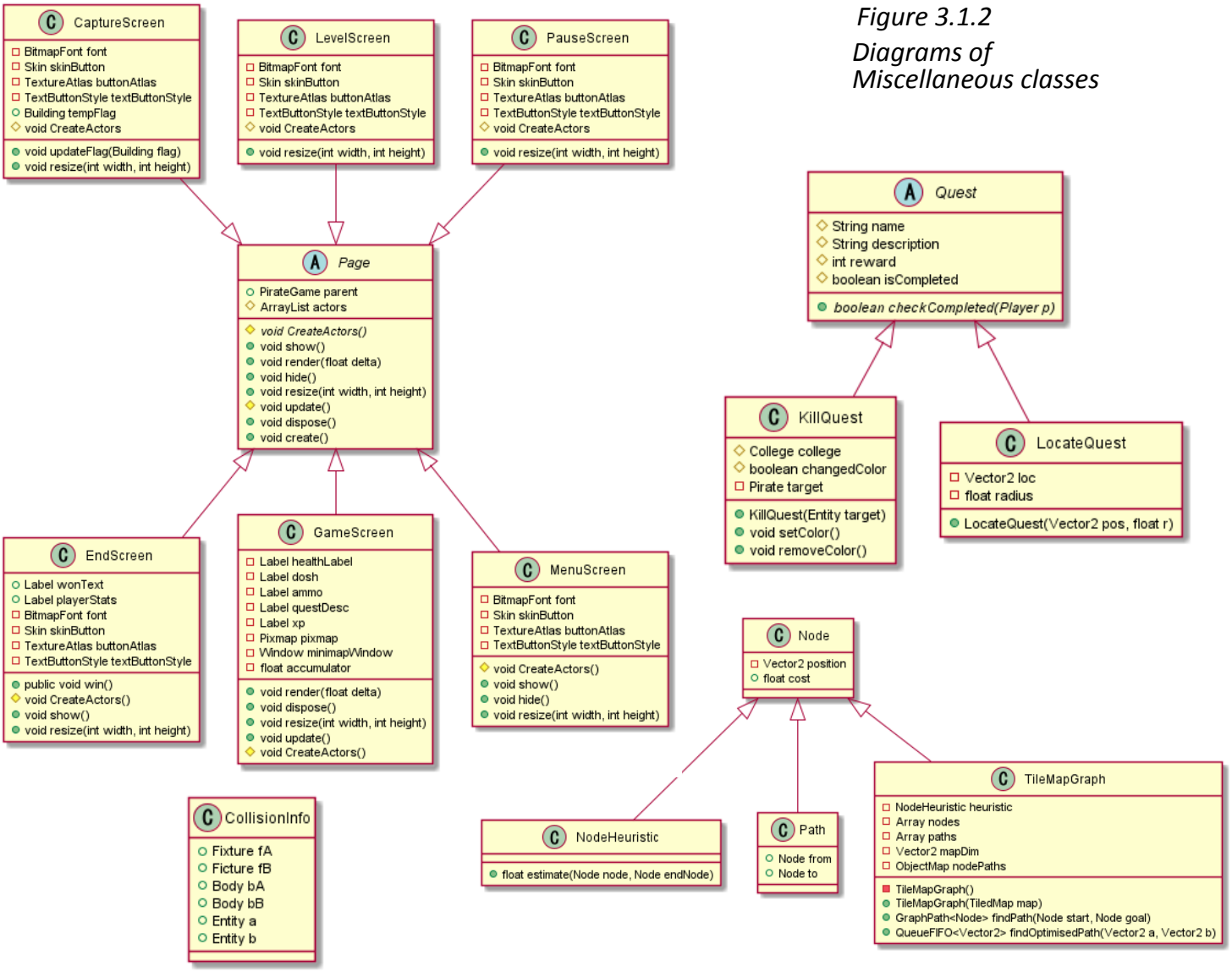

Figure 3.1.2
Diagrams of Miscellaneous classes

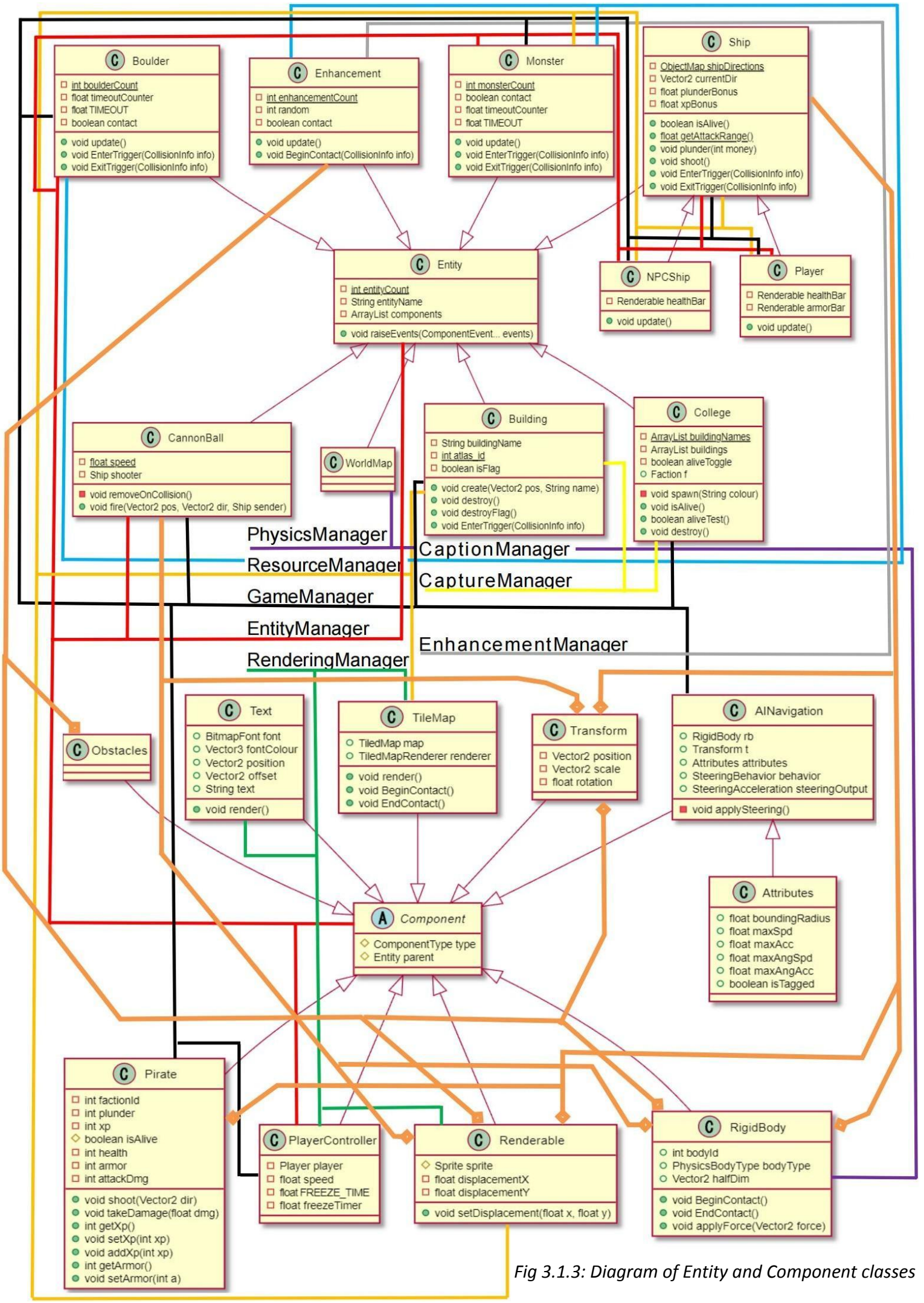

Fig 3.1.3: Diagram of Entity and Component classes

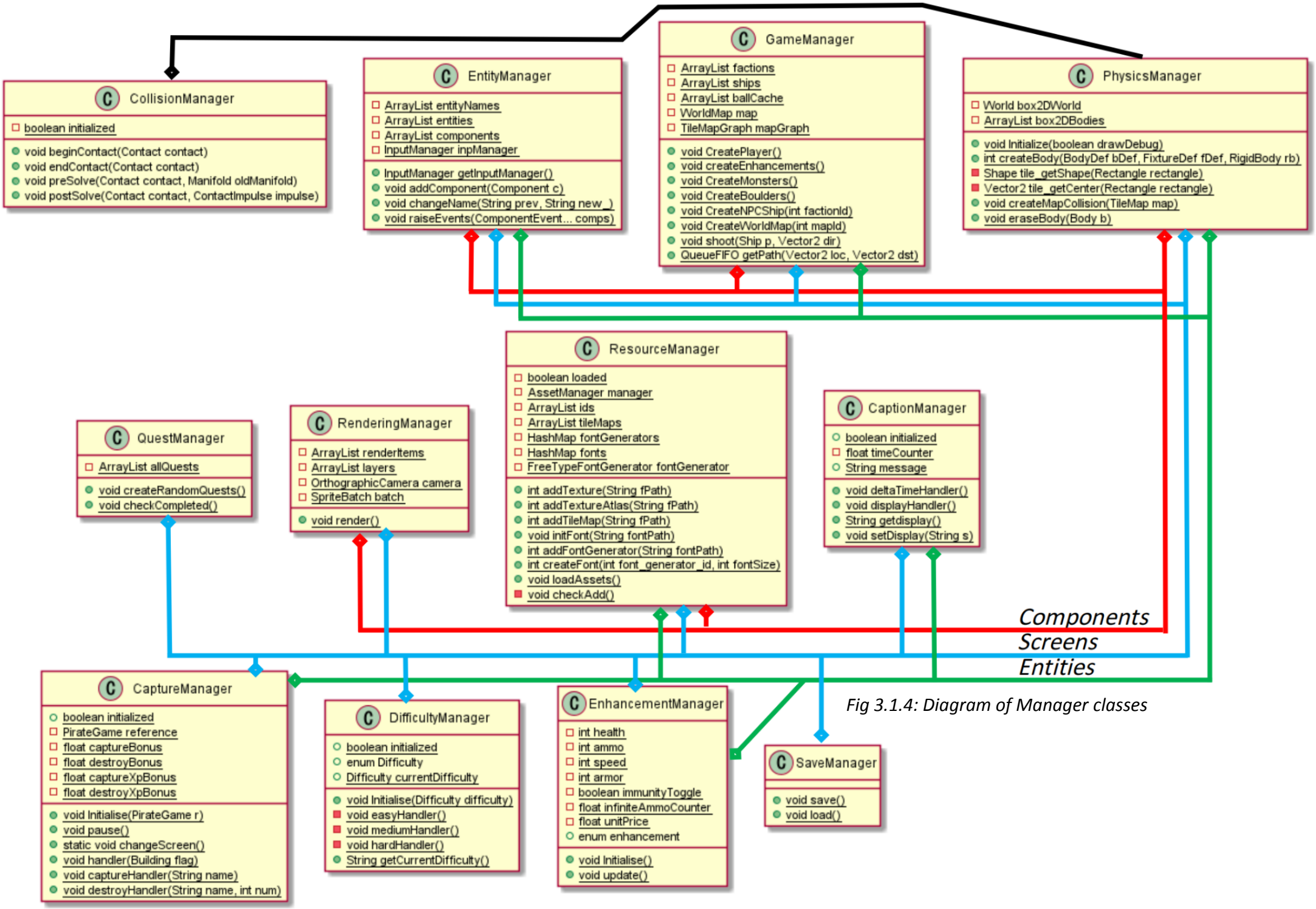

Fig 3.1.4: Diagram of Manager classes

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, Managers were connected to UI, Components and Entities rather than only game objects and Intangibles as fig 3.1.1 implies. Nevertheless, the plan given by the abstract architecture still gave a good foundation for working on the assessment.

Concrete architecture builds on the abstract in two main ways, by showing how the implementation of our code evolved and differed from our abstract plan, and by reflecting how different parts of our code enabled different functionalities (and thus solved different requirements).

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 and allows others to more easily locate the code that produces a specific functionality.

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. There isn't enough space and attempting to focus on the full scope of our code may detract from the core functionality and requirements of our program and assessment respectively.

In the interest of proving that our code has met our requirements, and how our code met these requirements, our concrete architecture only recorded key variables and methods that outline core functionality. Methods and variables not directly related to our requirements or program core functionality are largely omitted, especially if specific classes have a large amount of methods and variables.

We also 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 are not easy to distinguish in the architecture. 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 it is difficult to prove every assessment requirement purely relying on the architecture. A better reference to some functionality can only be found by perusing the rendered Javadocs (and/or code) associated with the game.

Relations to requirements

UR_POWERUPS

Referring to Enhancement Manager, more than 5 power ups are available for the user in the game. Powerups for ammo, health, armor, temporary speed, temporary infinite ammo and temporary immunity are supported and variables for the 6 power ups can be seen in the Concrete Architecture for the class.

UR_EARN_MONEY, UR_EARN_XP, FR_XP_UPDATE

The class Pirate in fig 3.1.3 has the 2 integer variables xp and plunder. The figure shows that the Entity Ship imports the Pirate class and has a relationship with it. NPCship and Player are both classes extended from Ship, and use the 2 variables xp and plunder to facilitate the player gain of plunder and xp.

FR_VIEWPORT_SCALING

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.

FR_GAME_WIN

In EndScreen class on fig 3.1.2, there is a label called wonText and a method called win(), which are responsible for displaying status of the completion of boss encounter.

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.