Team 12
Ibrahim Mahmoud – Sec.1 BN.1
Ahmad Khaled – Sec.1 BN.3
Ahmed Essam – Sec.1 BN.7
Omnia Zakaria – Sec.1 BN.13

Geometry Adventure

Team 12

Software Design Specification (SDS)

CM Identifier: SDS_v1.12

Revision History

Sl. No.	Prepared/ Modified by	E-mail	Version	Date	Approved by	Descriptions/ Remarks
1.	Omnia Zakaria	zicomohyee@yahoo.com	1.1	8/4/2018	AK	Modified the template to our project.
2.	Ibrahim Mahmoud		1.1	8/4/2018	AK	Added the Class Diagrams
3	A. Khaled	Enrev91@gmail.com	1.12	10/4/2018	TBD	Cleaning up, class diagrams.

Distribution list

Name	E-mail	Notes
Eng. Ali El Seddeek	ALIELSEDDEEK@GMAIL.COM	Review of document and progress needed.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 Purpose of this Document

This SDS document defines the design of Geometry Adventure. It contains specific information about the architectural models used, the design patterns followed, and how the Game's systems are organized together to meet the requirements specified in the SRS document[1].

1.2 Scope

Geometry Adventure is a top-down shooter action game that runs on Android phones as well as Desktop Computers. A working prototype will be developed which includes player motion, combat with at least two types of enemies, multiple weapons, as well as all the subsystems that make these features feasible (artificial intelligence for searching, patrolling, and combat, as well as collision detection, the physics subsystem, etc.). In addition, a map creator will be developed that allows the creation of levels (placing tiles, planning enemy patrol routes) that the game will run and read. For more information, please refer to the SRS.

1.3 Table of Acronyms and Definitions

Term	Definition
SRS	Software Requirement Specification
SDS	Software Design Specification
ECS	Entity-Component-System

Table 1: Acronyms and Definitions

.

1.4 References

[1] SRS document, SEM 12, 10 March 2018 version.
[2] Wikipedia, "Entity-Component-System", 8 April 2018 version.
[3] Software Engineering Tutorial Notes.

1.5 Overview of Document

This document is mainly concerned with specifying the main aspects of the design of Geometry Adventure, Starting with the Architecture of the system and how the system is decomposed into a set of interacting subsystems to meet the requirements. Each subsystem consists of a number of classes that collaborate together to do the subsystem functionality. Different subsystems are described using class diagrams. Also sequence diagrams are used to illustrate how the work flows between various subsystems.

2. System Architecture

The architectural pattern used in Geometry Adventure is the *Entity-Component-System (ECS)* architectural pattern. ECS is an architectural pattern that is widely used in games because the flexibility and control it gives the developer while avoiding the performance pitfalls associated with complex inheritance trees. ECS follows the *composition over inheritance* principle which states that classes should achieve polymorphic behavior by their composition (containing objects of other classes that implement the desired functionalities) rather than inheritance. Please note that the level creator does *not* use this architecture, and that it uses the more familiar MVC.

The ECS architecture itself is rather simple: each object in the game is considered an *entity* which can simply be represented by a unique ID (or index in an array) along with an associated array of *components* (or component IDs/indices). A component is where data is held and is best thought of like a C-struct: a component contains data that one or more systems can access, and has no operations or methods itself. Operations take place in *systems* which are classes that operate on arrays of entities having one or more specified component. A system ideally stores very little data itself, since most of the data is available in the components it operates on. Systems communicate with each other through firing and listening to *events* and through the data in components. The *Manager* handles the requests systems might make (to fire events, retrieve entities or components, call their update functions, etc.).

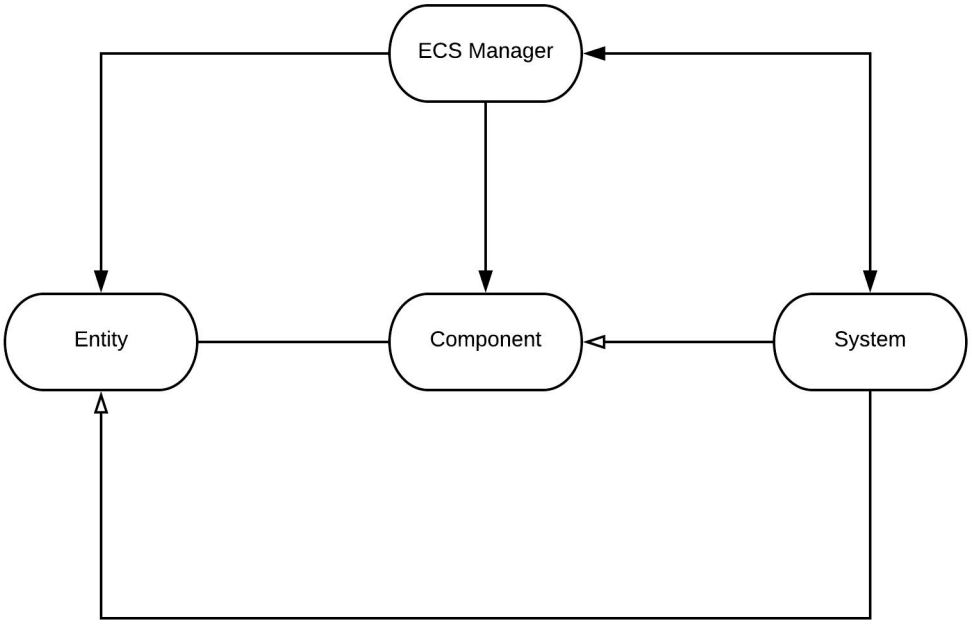

Figure 1: ECS Architecture

As an example for this, consider the operation of the Physics system:

- The physics system operates *only* on entities that have a physics component. The programmer must, through the ECS Manager, correctly assign to the relevant entities (like the Player & the enemies) physics components. The physics component contains data about the position, velocity, acceleration, and rotation angle of the entity it is attached to.
- The physics system is provided with an array of all the entities it can operate on by the ECS manager when it is added to the manager's list of systems.
- Every frame, the ECS manager calls the physics system's update function: the update function loops over all the entities managed by the systems and updates their physics component's using the entity's data (for example, altering the position based on the velocity).
- The physics system subscribes to the collision event when it is attached to the manager. Then, whenever a collision event is fired (by the collision system) the manager calls the physics

system's event handling function which allows the system to respond.

2.1 Design Patterns Description

2.1.1 Factory

The player can have multiple types of weapons throughout the game. With each weapon having different specification (e.g.), It would be redundant to fill the specs of each weapon in every time we create one. Therefore, a weapon factory is used to do this job. It takes the ID of the required weapon and returns an object of the weapon.

2.1.2 Observer

Each system is only concerned with the component it handles. But sometimes a system's job depends on other systems. So in this case, that system fires an event that it wants to do some job and systems that are needed to complete that job are notified by that event (e.g. Physics system handles movement of objects, but before it moves objects it asks the collision system if such movement would cause a collision or not.).

2.1.3 Adapter

Because the AI system needs to search the map and find the shortest path to the player when chasing him, a searching algorithm must be used to locate such path. However, searching algorithms operate on graphs while the map in the game is represented directly in continuous space, hence an adapter class (MapGraph) was created to allow the Map to be converted into the discrete Graph which the searching algorithm used (A* with the Manhattan Distance Heurestic) can operate on.

3. Design Models

3.1 Class Diagrams

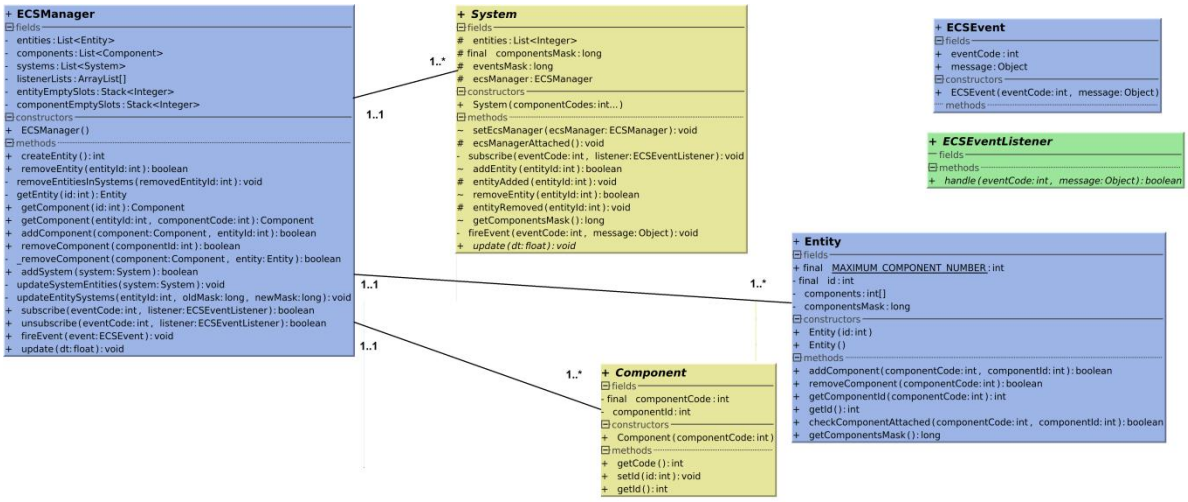

Figure 2: ECS Class Diagram

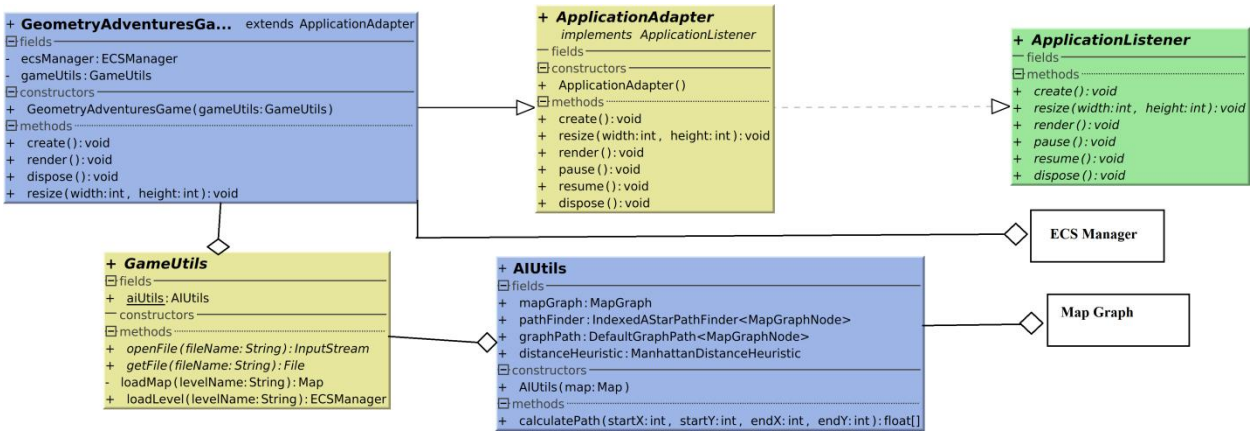

Figure 3: Base Game Class Diagram

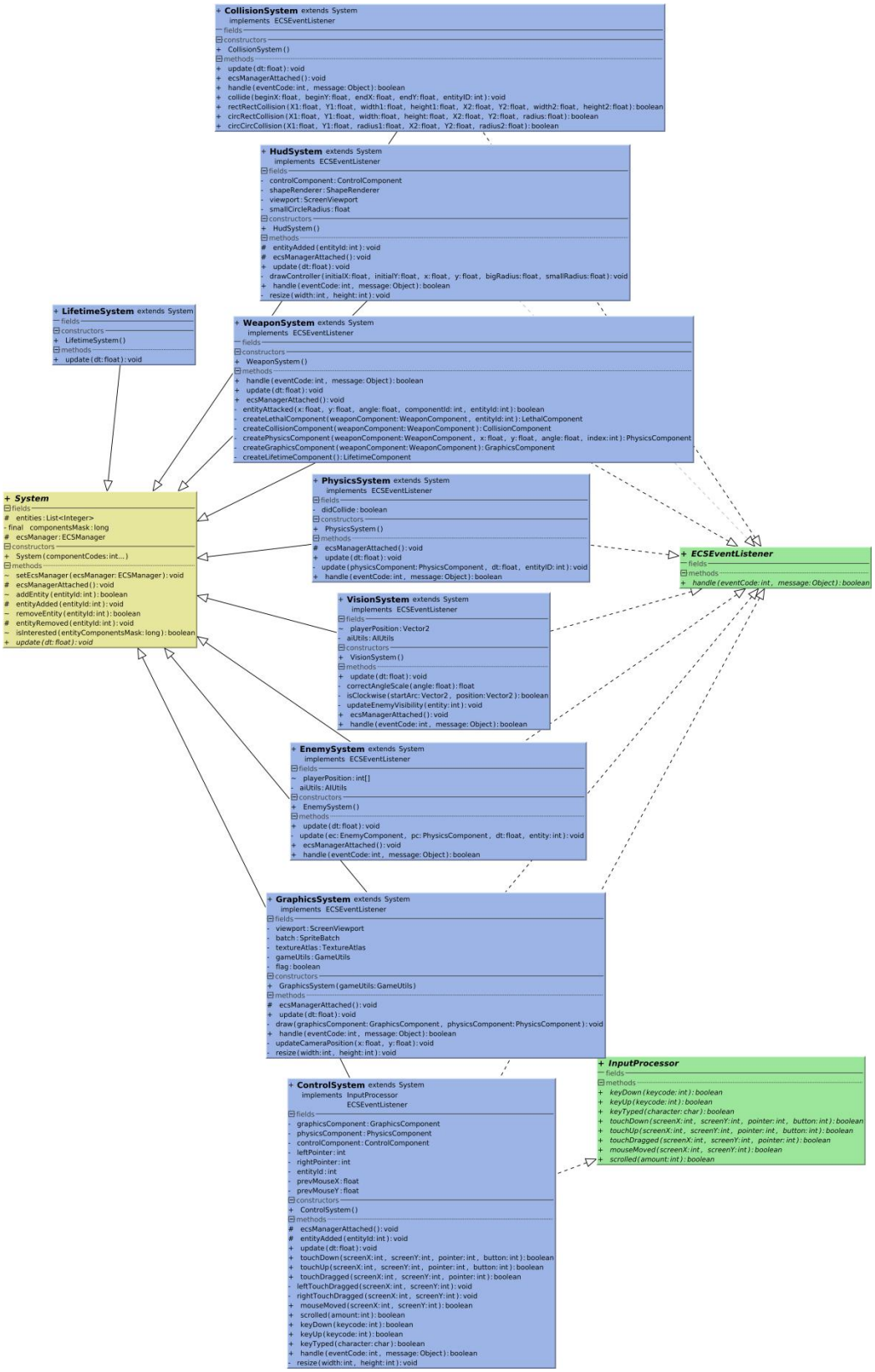

Figure 4: Systems Class Diagram

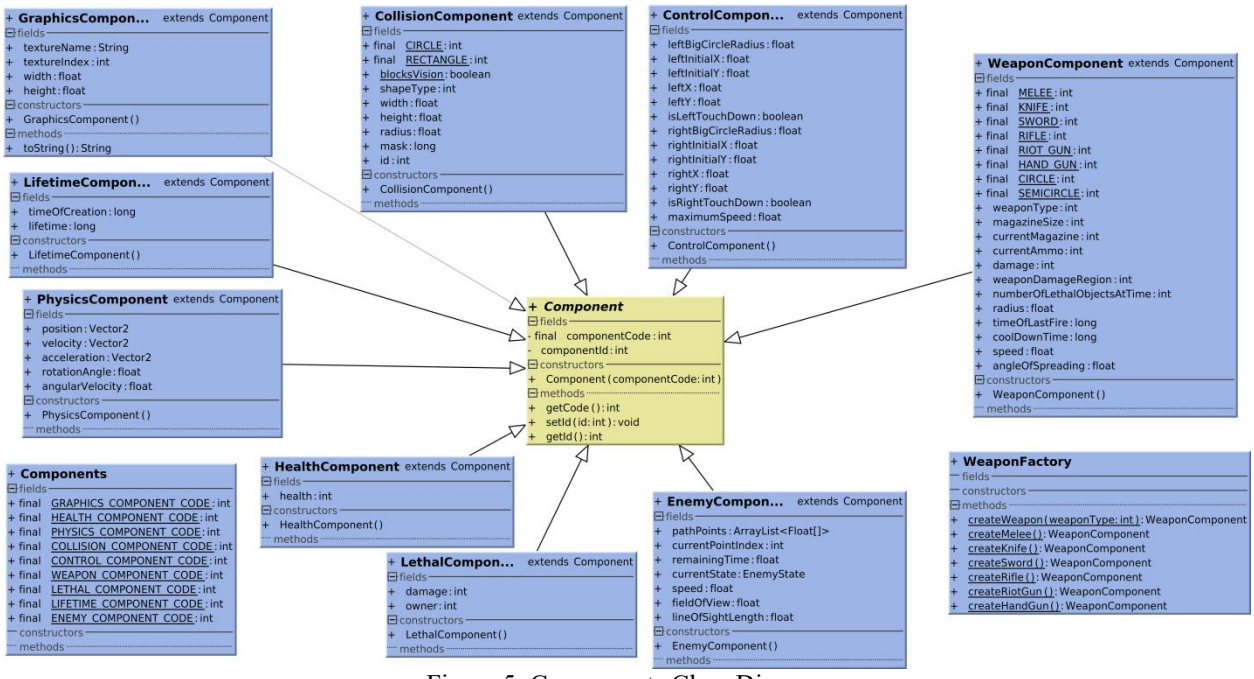

Figure 5: Components Class Diagram

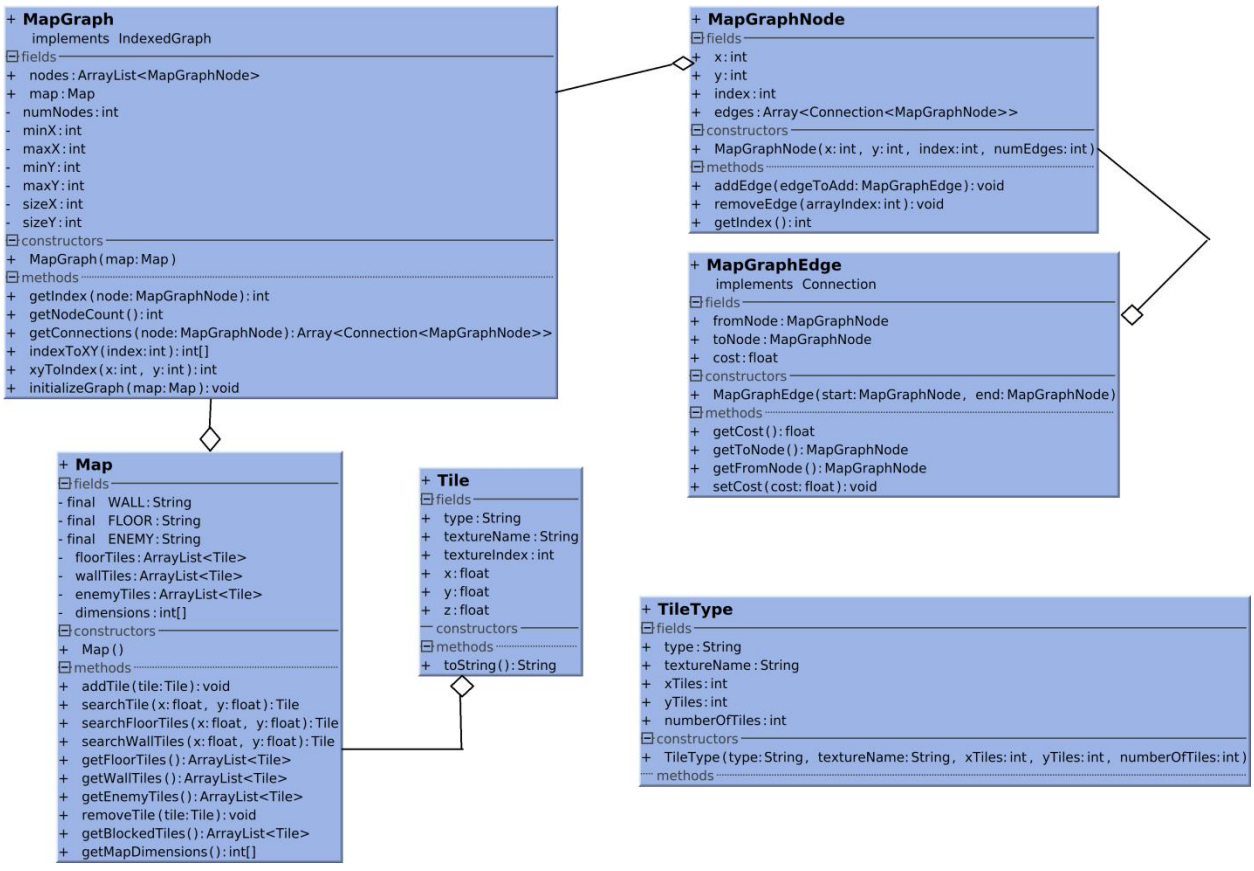

Figure 6: Model Class Diagram

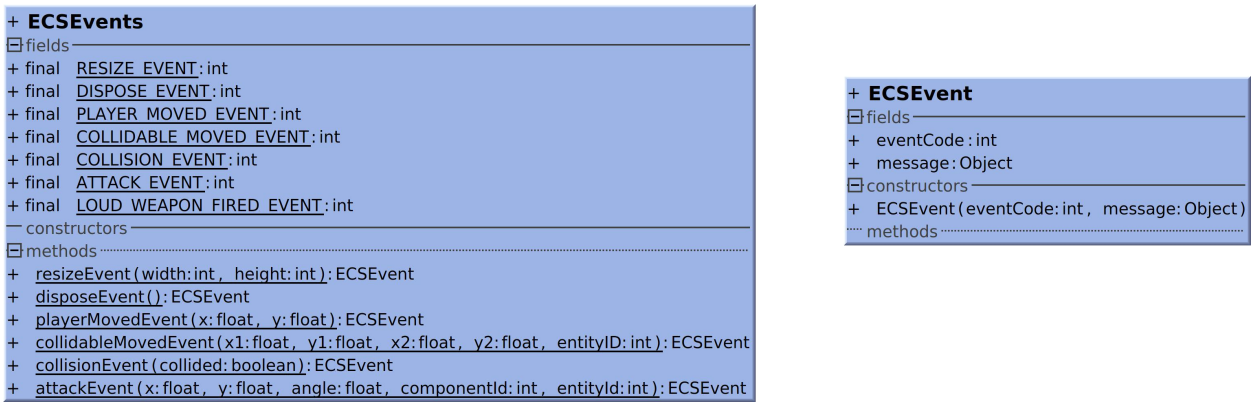

Figure 7: Events Class Diagram

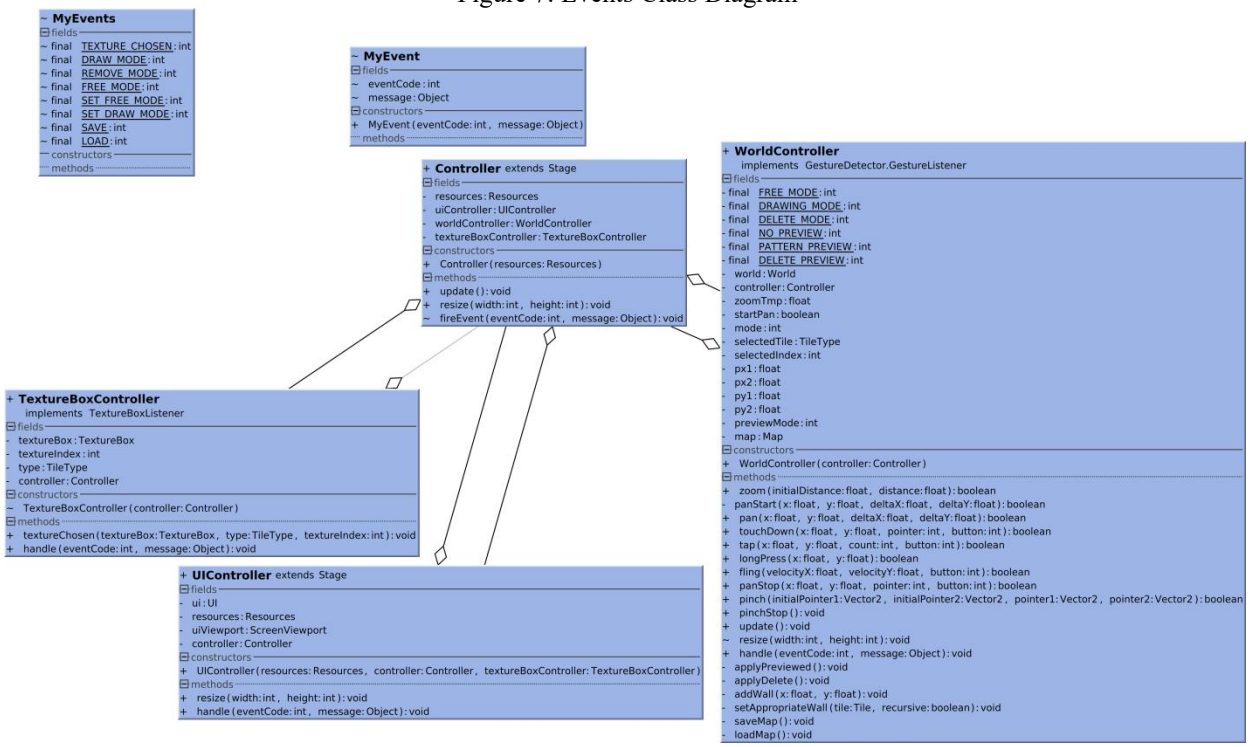

Figure 8: Map Creator Controller Diagram

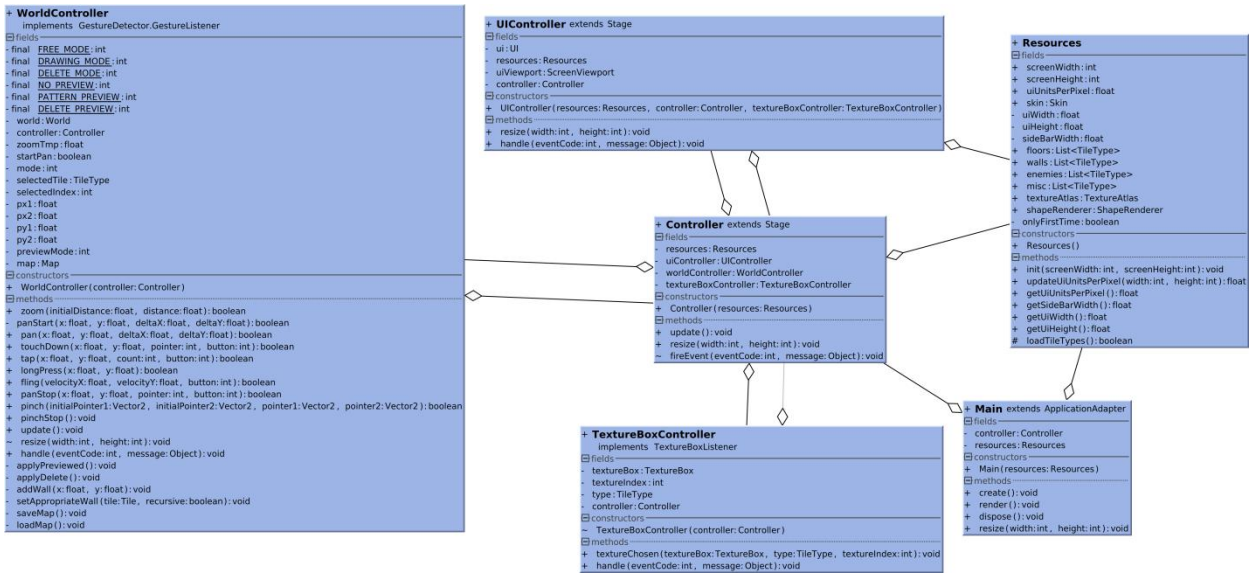

Figure 9: Map Creator Main Classes Diagram

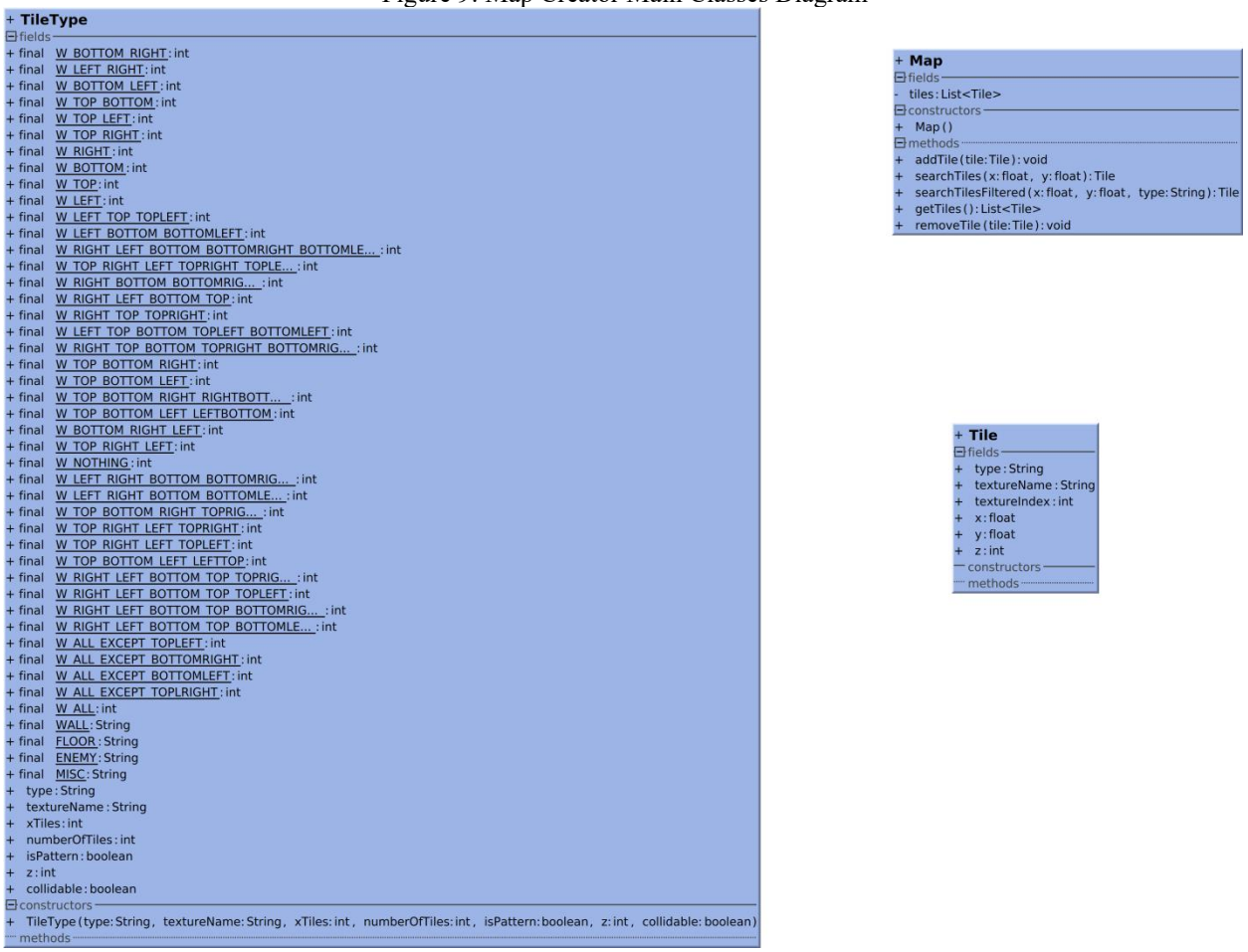

Figure 10: Map Creator Model Diagram

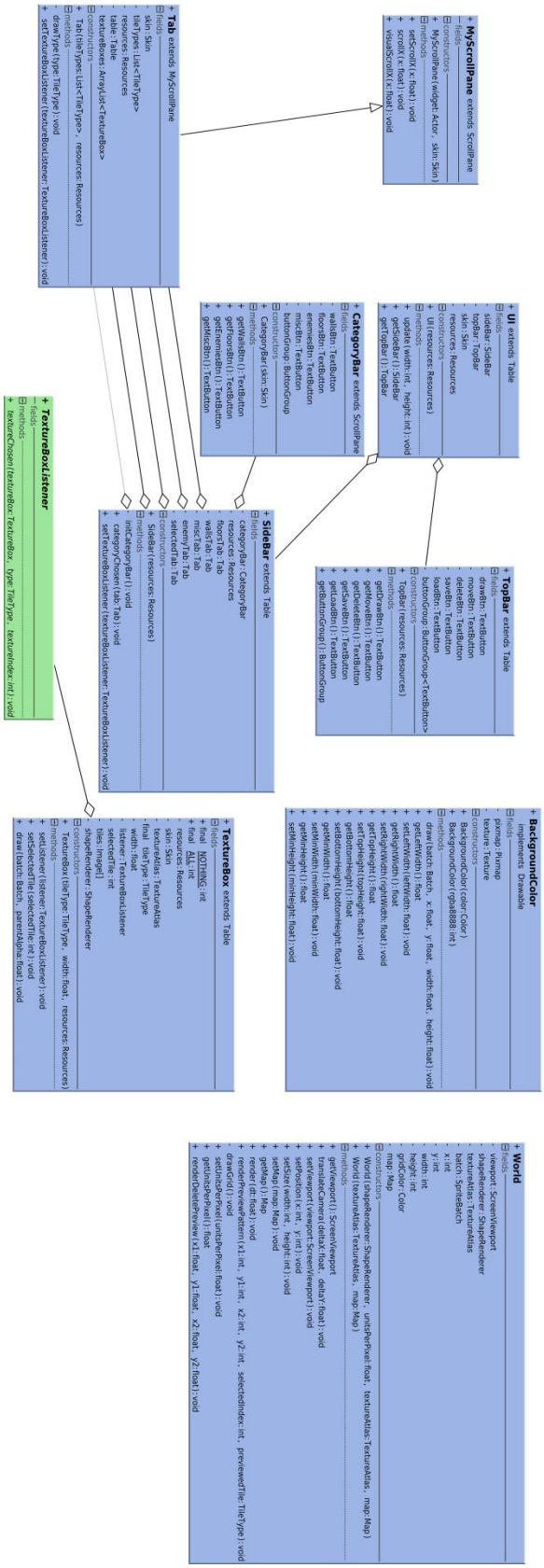

Figure 11: Map Creator View Diagram

3.2 Interaction Diagrams

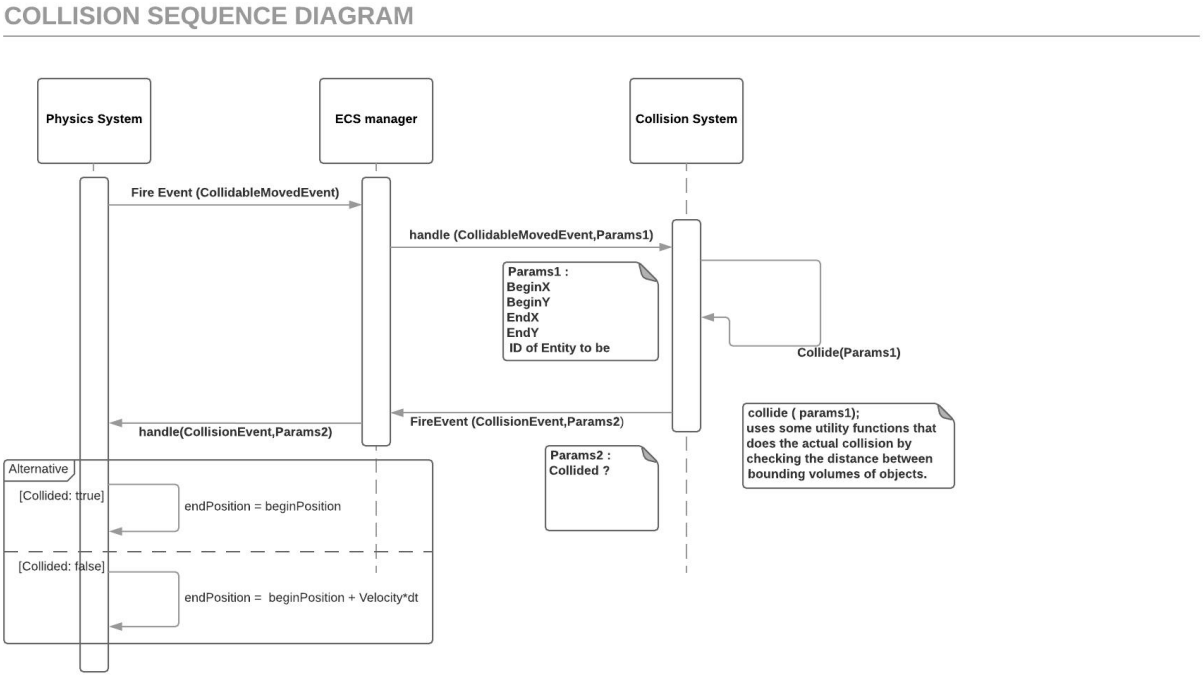

Figure 12: Collision Sequence Diagram

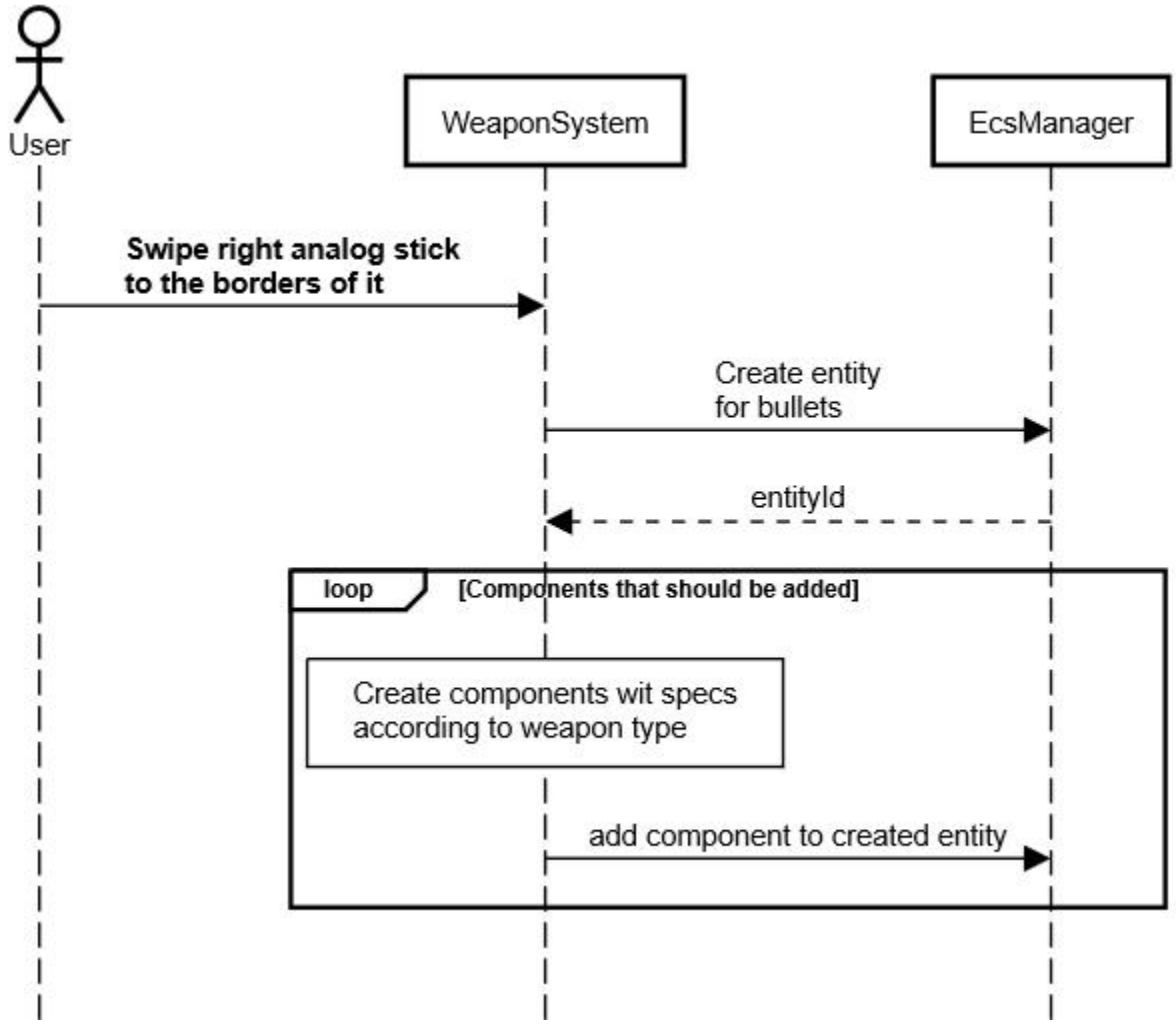

Figure 13: Player Firing Weapon

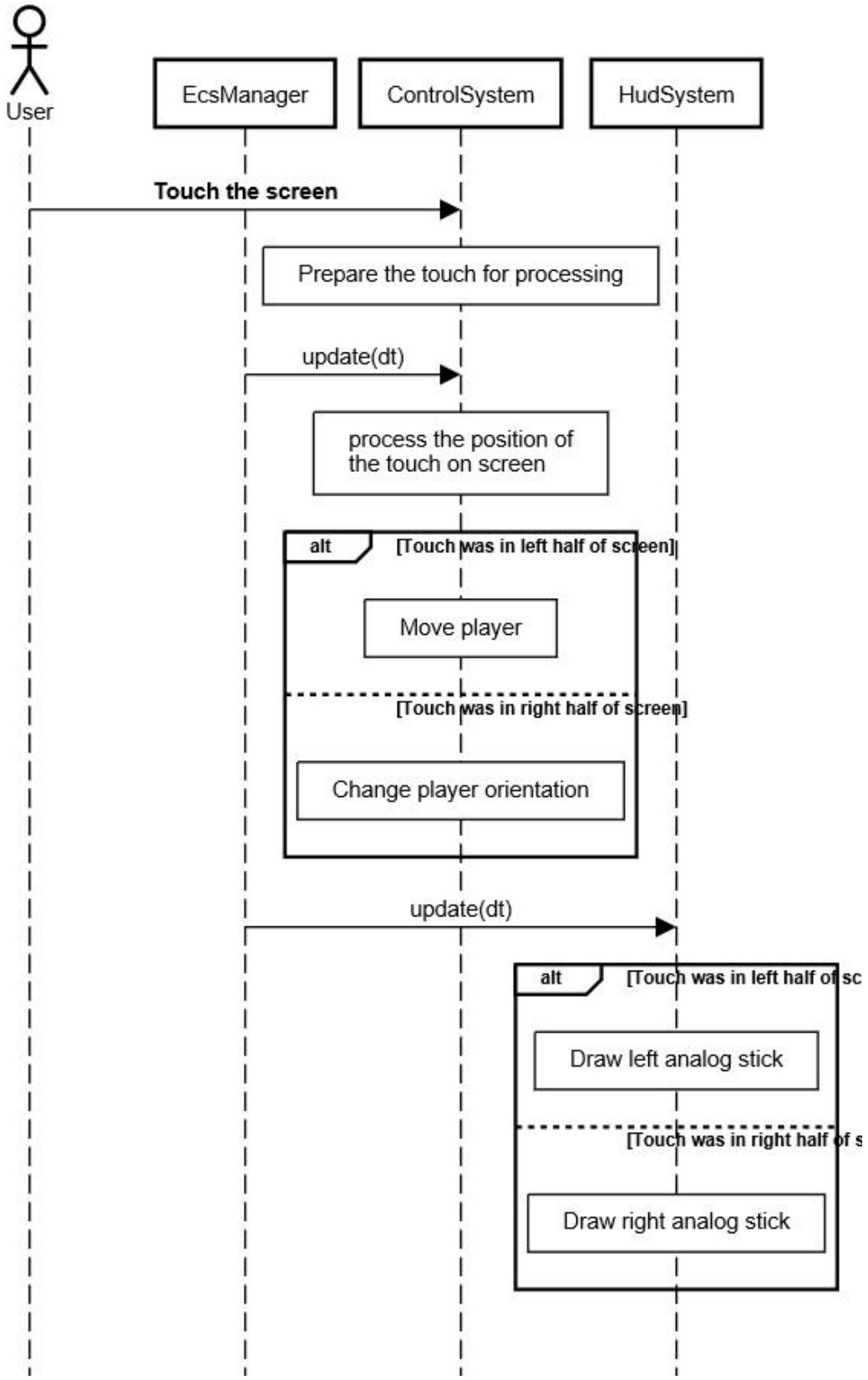

Figure 14: Player Motion

3.3 State Chart Diagrams

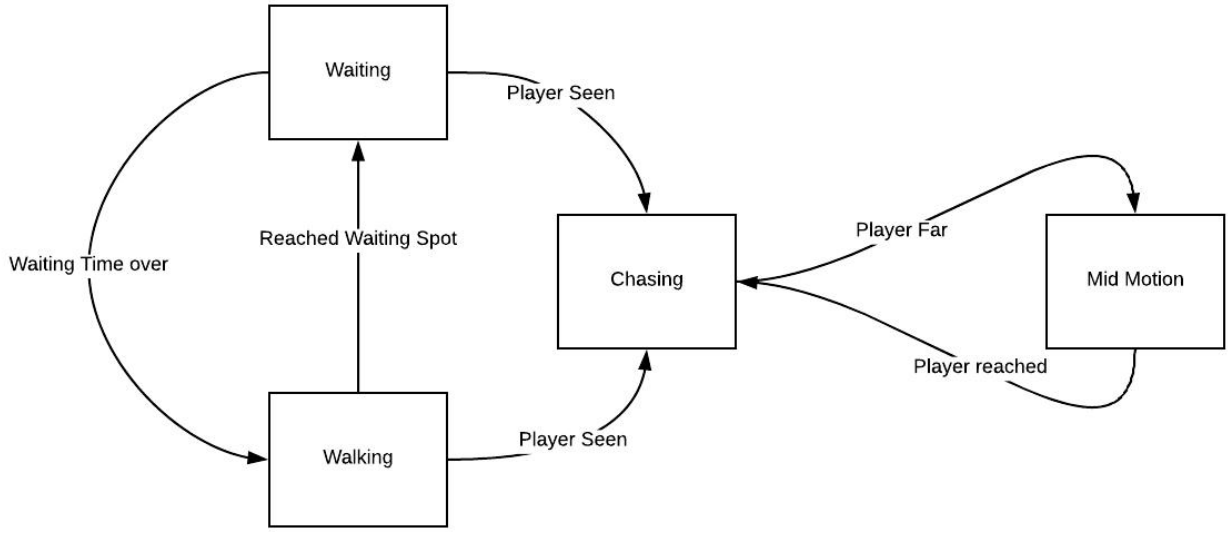

Figure 15: Enemy AI State Diagram

3.4 Design Element ID mapping

ID	Design Element(s)
ECS	ECS Manager, ECS System, ECSEvent, ECSEventListener, ECSEvents, Entity, Component
UTL	GameUtils, AIUtils, GeometryAdventuresGame, ApplicationAdapter, ApplicationListener
PHY	Collision System, Collision Component, Physics System, Physics Component
WEP	Weapon Component, Weapon System, Weapon Factory
CTL	Control Component, Control System
ENM	Enemy Component, Enemy System, Vision System
HEL	Health Component, Lethal Component
MAP	Map, Tile, TileType
MPG	MapGraph, MapGraphNode, MapGraphEdge
GFX	Graphics System, Graphics Component
LVC	Level Creator

Table 2: Design Element ID mapping

Due to the nature of the ECS architecture, many classes actually represent a single “object”, and our mapping reflects this.

4. System Deployment

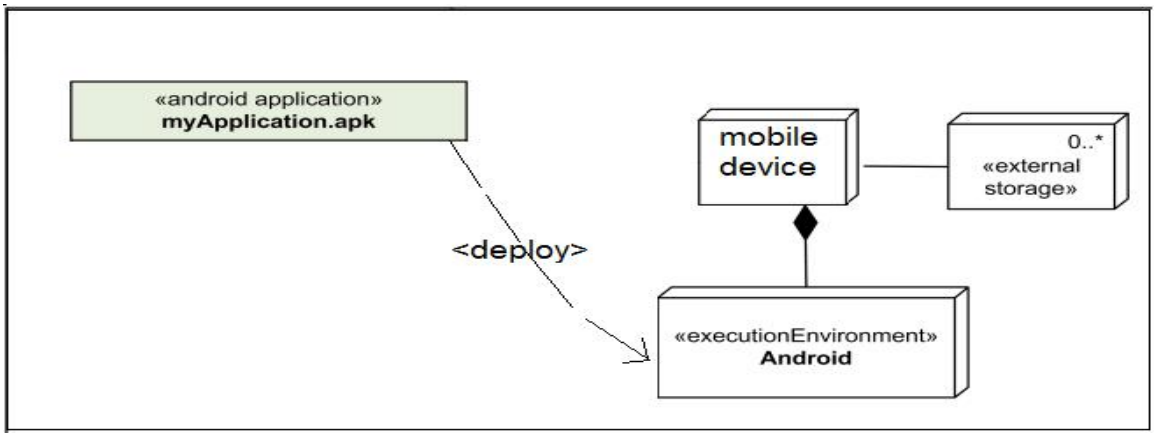

Figure 16: System Deployment Diagram

5. Traceability to Requirements

RID	UC1.1	UC1.2	UC2.1	UC2.2	UC2.3	UC2.4	UC2.5	UC3.1	UC3.2
ECS									
UTL									
PHY		✓							
WEP			✓		✓				
CTL		✓							
ENM			✓		✓				✓
HEL			✓						✓
MAP		✓		✓					
MPG			✓	✓					
GFX		✓				✓			
LVC								✓	

Table 3: Traceability to Requirements

The shaded rows are for architecture-specific and utility classes: these do not impact the user requirements. Please note that this diagram **is for the current iteration only** and that it can change for future iterations (and that certain features are lacking). Nonfunctional requirements have not been mentioned.