Ayai Design Document

Marjorie Bartell | Ben Kos | Kyle Sheehan
Andrew Gotow | Christian Benincasa | Dan Muller
Rory O’Kane

April 2015

Advisor: Santiago Ontanon

Contents

1. **Introduction**
    1.1. Purpose
        1.1.1. Scope
        1.1.2. Context Diagram
2. **Architecture**
    2.1. Overview
    2.2. Servers
        2.2.1. Web Server
        2.2.2. Authorization Server
        2.2.3. Socko Server
        2.2.4. Database
    2.3. Network Message Interpretation/Processing
    2.4. Game State
    2.5. Systems
        2.5.1. AI System
        2.5.2. Network System
3. **Detailed Design ***
    3.1. ECS Game Loop
        3.1.1. World
        3.1.2. Entity
        3.1.3. System
        3.1.4. EntityProcessingSystem
        3.1.5. TimedSystem
        3.1.6. IntervalSystem
        3.1.7. Component
        3.1.8. Game Loop
    3.2. Components
        3.2.1. Position
        3.2.2. Actionable
        3.2.3. Attack
        3.2.4. Bounds
        3.2.5. Character
        3.2.6. Frame
        3.2.7. Health
        3.2.8. Inventory
        3.2.9. Velocity
        3.2.10. Time
        3.2.11. Mana
        3.2.12. Stats
        3.2.13. Stat
        3.2.14. Transport
        3.2.15. NetworkingActor

3.2.16. Respawn
3.2.17. TileMap
3.2.18. ItemUse
3.2.19. Experience
3.2.20. Cooldown
3.2.21. Quest
3.2.22. QuestBag
3.2.23. Equipment
3.3. Items
3.3.1. Item
3.3.2. ItemType
3.3.3. Weapon
3.3.4. Weapon
3.4. Systems
3.4.1. MovementSystem
3.4.2. CollisionSystem
3.4.3. HealthSystem
3.4.4. RespawningSystem
3.4.5. FrameExpirationSystem
3.4.6. NetworkingSystem
3.4.7. NPCRespawningSystem
3.4.8. LevelingSystem
3.4.9. StatusEffectSystem
3.4.10. CooldownSystem
3.4.11. ItemSystem
3.4.12. AttackSystem
3.4.13. RoomChangingSystem
3.4.14. AISystem
3.5. Status Effects
3.5.1. Effect
3.5.2. TimeAttribute
3.5.3. OneOff
3.5.4. TimedInterval
3.5.5. Duration
3.5.6. Multiplier
3.6. Movement Processes
3.6.1. Action
3.6.2. MovementDirection
3.6.3. MovementDirection Case Classes
3.7. Collision Objects
3.7.1. QuadTree
3.7.2. Rectangle
3.8. Factories
3.8.1. ClassFactory
3.8.2. ItemFactory

3.8.3. QuestFactory
3.8.4. GraphFactory
3.8.5. EntityFactory
3.9. 3.9 Quest Generation *
3.9.1. 3.9.1 Overview
3.9.2. 3.9.2 Components
3.9.2.1. 3.9.2.1 GenerateQuest
3.9.2.2. 3.9.2.2 QuestHistory
3.9.3. 3.9.3 Systems
3.9.3.1. 3.9.3.1 QuestGenerationSystem
3.9.4. 3.9.4 Architecture
3.9.4.1. 3.9.4.1 Impact on existing architecture
3.9.4.2. 3.9.4.2 System Sequence Diagram
3.9.5. 3.9.5 Algorithms
3.9.5.1. 3.9.5.1 PaSSAGE
3.10. 3.10 Perception *
3.10.1. 3.10.1 Components
3.10.1.1. 3.10.1.1 Sense
3.10.1.2. 3.10.1.2 Vision
3.10.1.3. 3.10.1.3 Hearing
3.10.1.4. 3.10.1.4 Sound-Producing
3.10.1.5. 3.10.1.5 Memory
3.10.1.6. 3.10.1.6 Memory Contents
3.10.2. 3.10.2 Entities
3.10.2.1. 3.10.2.1 SoundEntity
3.10.3. 3.10.3 Systems
3.10.3.1. 3.10.3.1 Primary System: Perception System
3.10.3.2. 3.10.3.2 Seconary/Included System and Subsystems
3.10.3.2.1. 3.10.3.2.1 Vision System
3.10.3.2.2. 3.10.3.2.2 Hearing System
3.10.3.2.3. 3.10.3.2.3 Memory System
3.10.3.2.4. 3.10.3.2.4 Communication System
3.10.4. 3.10.4 Architecture
3.10.5. 3.10.5 Process and Design Patterns
3.10.5.1. 3.10.5.1 Sequence Diagram
3.10.5.2. 3.10.5.2 Entity Component System
3.10.5.3. 3.10.5.3 Observer Pattern
3.10.5.4. 3.10.5.4 Strategy Pattern
3.10.6. 3.10.6 Algorithms
3.10.6.1. 3.10.6.1 Bresenham’s line algorithm
3.10.6.2. 3.10.6.2 Wu’s line algorithm
3.11. 3.11 Pathfinding *
3.11.1. 3.11.1 Components
3.11.1.1. 3.11.1.1 Pathfinder

3.11.1.2. 3.11.1.2 AStar
3.11.1.3. 3.11.1.3 Dijkstra
3.11.1.4. 3.11.1.4 DistanceHueristic
3.11.1.5. 3.11.1.5 ManhattanDistance
3.11.1.6. 3.11.1.6 DiagonalDistance
3.11.2. 3.11.2 Systems
3.11.2.1. 3.11.2.1 PathfindingSystem
3.11.3. 3.11.3 Design
3.11.3.1. 3.11.3.1 Sequence Diagram
3.11.3.2. 3.11.3.2 Dependency Injection/Inversion of Control
3.11.3.3. 3.11.3.3 Strategy Pattern
3.11.4. 3.11.4 Algorithms/Data Structures
3.11.4.1. 3.11.4.1 A* search algorithm [1]
3.11.4.2. 3.11.4.2 Dijkstra's algorithm [2]
3.11.4.3. 3.11.4.3 Manhattan Distance
3.11.4.4. 3.11.4.4 Diagonal (Chebyshev) Distance
3.11.4.5. 3.11.4.5 Binary Heap (java.util.TreeSet)
3.11.5. 3.11.5 Architecture
3.11.6. 3.11.6 References
3.12. 3.12 Map Generation *
3.12.1. 3.12.1 WorldGenerator
3.12.2. 3.12.2 MapGenerator3.9 Quest Generation *
4. **Network System**
4.1. NetworkMessageQueue
4.2. NetworkMessageInterpreter
4.3. NetworkMessageProcessor
4.4. SockoServer
4.5. AuthorizationProcessor
5. **Ayai Web Application**
5.1. Overview
5.2. Login Page
5.3. Character Creation
5.4. Character Selection
5.5. Changing Settings
6. **Ayai World Editor**
6.1. Searching
6.2. Creating and Editing a New Entry
7. **Game Client**
7.1. Overview
7.2. Graphics
7.2.1. Display
7.2.2. UIElement
7.2.3. UnitFrame
7.2.4. Chat

7.2.5. Inventory
7.2.6. QuestLog
7.2.7. Quest
7.2.8. PeopleList
7.2.9. Settings Menu
7.3. Game
7.3.1. Ayai
7.3.2. GameStateInterface
7.3.3. InputHandler
7.4. Net
7.4.1. Connection
7.4.2. MessageReceiver
8. **Database Design**
9. **9 Game Configuration File** *
9.1. 9.1 Purpose
9.2. 9.2 Design
10. **Glossary**

1 Introduction

1.1 Purpose

This document specifies the entire software architecture and design for the Ayai MMORPG game and framework. The design decisions directly relate to the functionality, performance, constraints, attributes, and interfaces of the system. Ayai is a massively multiplayer online game that allows developers to implement research level AI and test its functionality with a potential base of approximately 20 players. Also provided is an open source framework that eases development of 2D web-based MMORPGS. In order to achieve these goals, the framework focuses on scalability, security, accessibility, and flexibility.

1.2 Scope

This document describes the software architecture and design for the initial release of Ayai, as described in the Ayai Software Requirements Document. Additionally, this document covers the second release of Ayai. The intended audience of this document exclusively includes the designers, developer, testers, and open-source developers who may use this framework.

1.3 Context Diagram

The context diagram shown in Figure 1 shows how the major components of the Ayai system fit into context with other components.

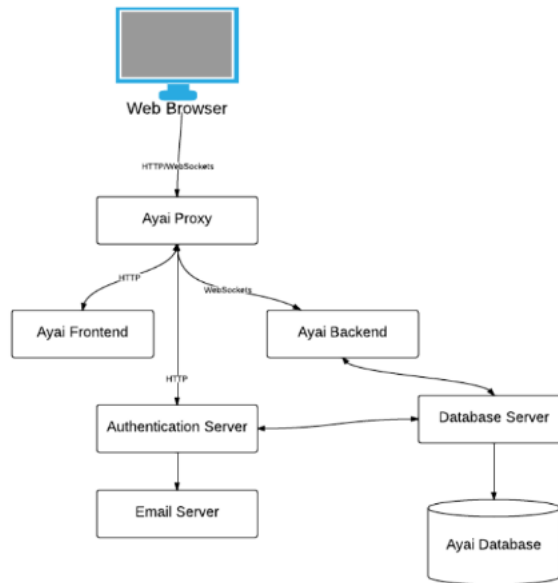

Figure 1: Context Diagram

The web server serves the Ayai frontend as a static web page. The authorization server handles authentication requests. Ayai's backend uses WebSocket connections to receive messages from the browser and return relevant game state and events.

2 Architecture

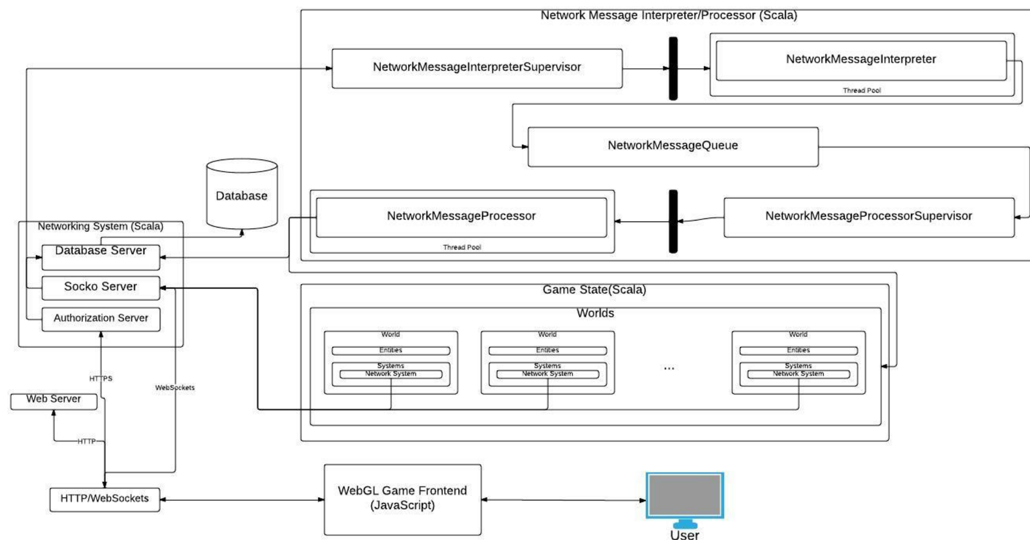

Figure 2: Architecture Diagram

2.1 Overview

The architecture behind Ayai is a collection of distinct, loosely coupled systems that divide responsibilities into appropriate groups and categories. The Ayai architecture also takes advantage of the actor model of concurrency in order to process the heaviest tasks in a distributed and concurrent manner. Notable portions of the system include the web server which provides a copy of the frontend for each player, the collection of servers that handle various authorization and database operations, and the distributed systems of supervisors and actors that interpret and process commands from the user, process changes to the game state, and returns updated data back to the user.

2.2 Servers

2.2.1 Web Server

The frontend of the Ayai project is a website comprising of static HTML5 content. The web server nginx has the task of receiving all HTTP/HTTPS/WebSocket connections from the user. Nginx was chosen due to its static page serving performance and capabilities and reverse proxy features. When the user first browses to the Ayai website, nginx returns a static copy of the website. However, if the user has an authentication request or is sending a game command, nginx proxys the request to the appropriate server

2.2.2 Authorization Server

A simple authorization server provides authentication for the Ayai system. As WebSockets do not natively support authentication, HTTPS is used in tandem with WebSockets in order to provide user security and authorization. Users send their credentials over HTTPS using the Basic Access Authentication mechanism and, if validated, receives a temporary token to validate their WebSocket connection.

2.2.3 Socko Server

The WebSocket server (created using the Socko library) accepts WebSocket connections forwarded by nginx and expects them to be in the form of a game related command (move, attack, etc.). The Socko server then forwards these network messages to a NetworkMessageInterpreterSupervisor, in preparation to be interpreted and then queued for processing.

2.2.4 Database

Ayai employs a light weight flat-file Java Database engine called H2. The Database stores user credentials and various portions of dynamic game state, such as maps,

inventories, character skills, locations, and experience. Various systems of the game store dynamic portions of the game state to the database at an infrequent rate (approximately once per 10 seconds). The entity factories retrieve this information when a character logs in.

2.3 Network Message Interpretation/Processing

The Network System Section (section 4) describes the mechanics of the Network Message system in further detail. The Socko server receives network messages that need to be interpreted for meaning and content before being processed. The NetworkMessageInterpreterSupervisor has a thread pool of NetworkMessageInterpreters, each of which understands a message received and places a game command into the NetworkMessageQueue. At each game tick, the NetworkMessageQueue is cleared and given to the NetworkMessageProcessorSupervisor for processing.

2.4 Game State

The game state in Ayai is represented as an Entity Component System, which stores, manages, and processes game state. Worlds separate players by in-game locality and stores data as entities with components.

2.5 Systems

Systems are then in charge of processing changes and game logic, applying these changes to the relevant components. Systems are placed on tiers, so that higher tiers must complete before a lower tier starts to process.

2.5.1 AI System

The AI System processes new information about the world and makes appropriate decisions related to the artificial intelligence of entities and the game itself. This includes low level decision making, such as movement and attacking for specific non player characters, to high level decision making, such as the creation of quests, enemies, and other necessary game entities.

2.5.2 NetworkSystem

At the lowest tier exists the NetworkSystem, which serializes the game state, calculates messages to return back to players, and sends messages back over the WebSocket connection to the frontend.

3 Detailed Design

3.1 ECS Game Loop

This section defines the ECS system and the main backend driver (called a Game Loop). These properties go into detail about the workings of the main loop of the system. The ECS system is a small system that consists of three main properties which are the Systems, Entities, and Components.

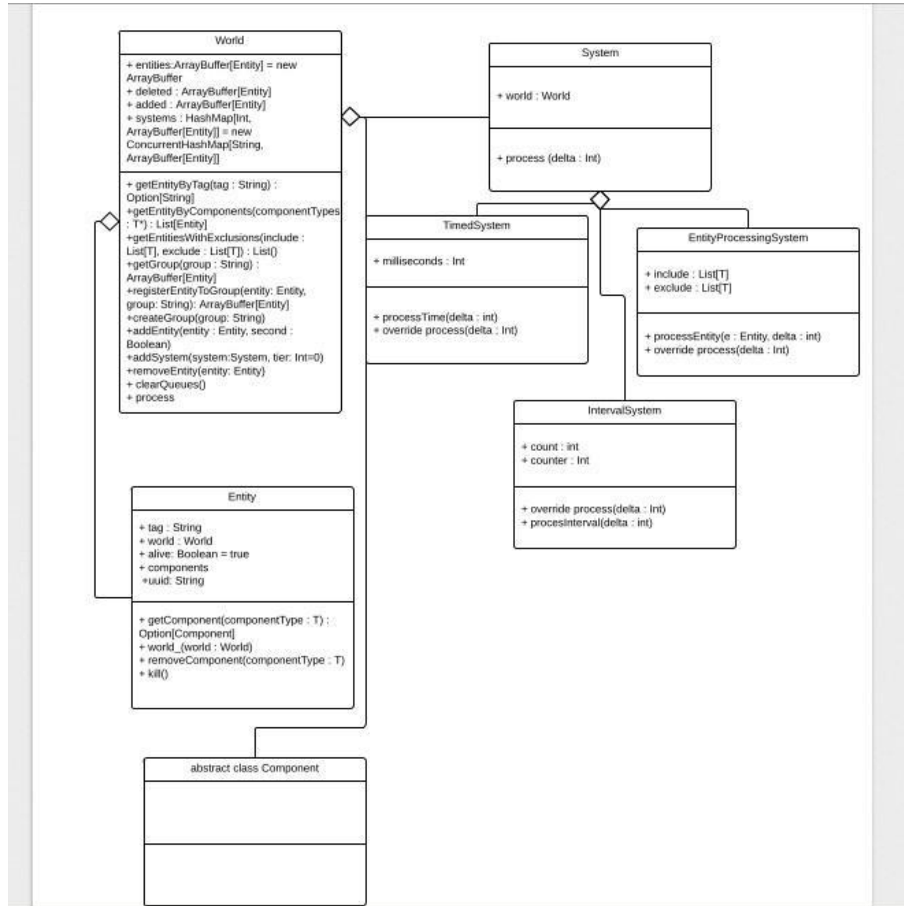

Figure 3: engine diagram

3.1.1 World

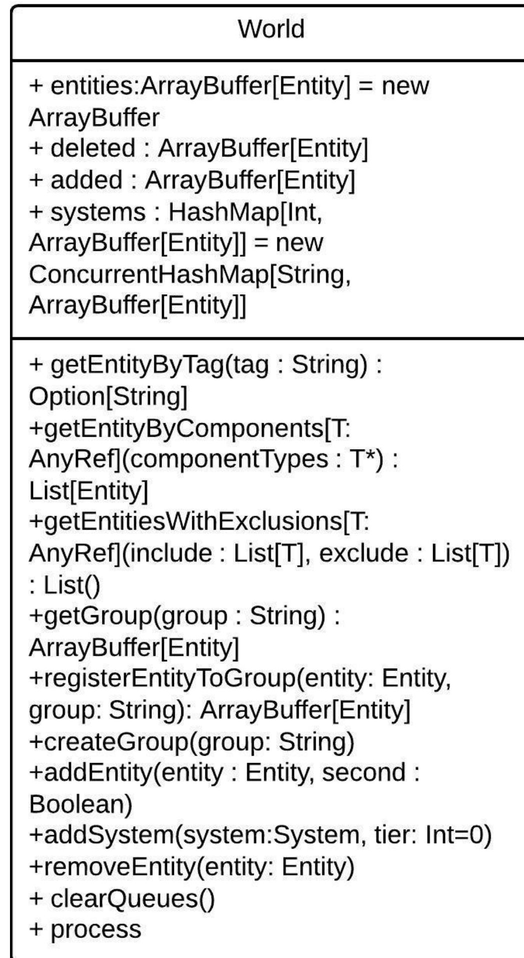

Figure 4: World Class Diagram

A World holds all entities, systems and processes and filters entity information

Attributes

Name	Type	Description
entities	ArrayBuffer[Entity]	Holds all known entities in a list
system	ArrayBuffer[System]	Holds all systems added to the world
Deleted	ArrayBuffer[Entity]	Holds all entities that are primed for deletion, but cannot be removed until after system process
Added	ArrayBuffer[Entity]	Holds all entities that are primed for addition to entities, but cannot

		be added to main list until system process is finished

Operations

Operation: getEntityByTag(tag : String) : Option[String]
Input : Tag - the unique tag of the entity
Output : Returns an option for an entity
Description : Finds an entity with a given tag and returns option on it.

Operation: getEntityByComponents(componentTypes : T*) : List[Entity]
Input : ComponentTypes : T - a list of types of component classes
Output : Returns a list of entities
Description : Takes a list of component types and returns a list of entities which have all the given components

Operation: getEntitiesWithExclusions(include : List(T), exclude : List(T)) : List[Entity]
Input : Include : List(T) - a list of types of component classes you want to find exclude : List(T) - a list of types you want to exclude from the find
Output : Returns a list of entities
Description : Takes a list of component types you want to search for in entities and a list of component types you do not want an Entity to have and returns a list of entities which match.

Operation: getGroup(group : String) : ArrayBuffer[String]
Input : Group : String - a group name
Output : List of Entities
Description : Returns list of entities that are matched to group

Operation: registerEntityToGroup(entity : Entity, group : String) : ArrayBuffer[Entity]
Input : entity : Entity - an entity to add
group : String - group to add to Output :
The group you are adding to
Description : Adds an entity to a group and returns that group

Operation: addEntity(e : Entity, second : Boolean)
Input : e : Entity - entity to add to world second : Boolean
- did this get called from entity itself
Output : None
Description : Add entity to world

Operation: createEntity(tag : String) : Entity

Input : tag : String - tag which to identify item
Output : Entity which is created
Description : Create and return a new entity, not added to world

Operation: addSystem(system : System)

Input : system : System - The system to add to world and processing cycle
Output : None
Description : Adds systems to the world systems list and is included in next process cycle

Operation: process()

Input : None
Output : None
Description : Runs process() on all systems that are included in the world

3.1.2 Entity

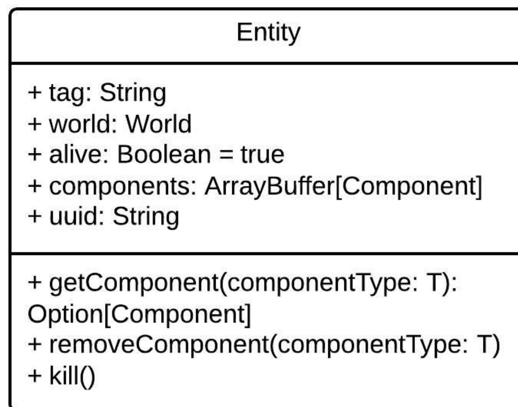

Figure 5: Entity Class Diagram

An Entity holds all data (Components) needed to be processed by a system for a specific function (characters, items, enemies).

Attributes

Name	Type	Description
Tag	String	Unique tag to look for entity
World	World	World which Entity belongs to
Alive	Boolean	Is an entity alive or dead
Components	ArrayBuffer[Component]	List of components

uuid	String	Unique id for character

Operations

Operation: getComponent(componentType : T) : Option[Component]
Input : ComponentType : T - classOf component to find
Output : Returns an option for the component
Description : Searches for a component in the list, and returns an option on it

Operation: removeComponent(componentType : T)
Input : ComponentTypes : T - classOf Component to find
Output : None
Description : Takes a component type and removes it from list of components

Operation: kill() Input
: None
Output : None
Description : Removes entity from the world it is a part of.

3.1.3 System

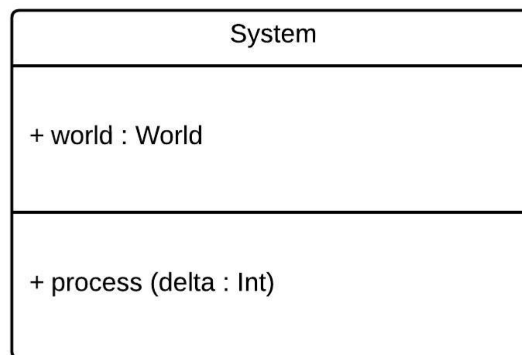

Figure 6: System Class Diagram

Systems are the framework's way of processing and manipulating data. Overriding the process function allows for the system to do work on the list of entities it uses.

Attributes

Name	Type	Description
world	World	The world it is a member of

Operations

Operation: process(delta : Int)
Input : delta : Int - The time difference from the last frame
Output : None
Description : Abstract defined function needing to be overwritten

3.1.4 EntityProcessingSystem

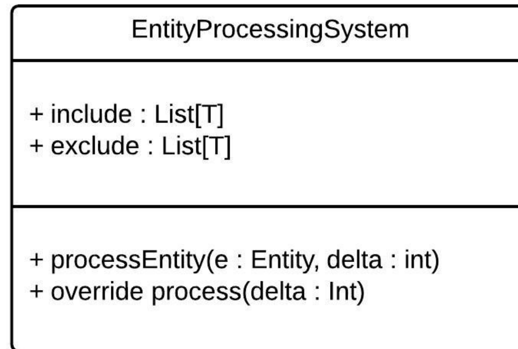

Figure 7: System Class Diagram

An EntityProcessingSystem inherits from System and allows for users to manipulate one Entity at a time. Also includes list inputs to exclude and include certain components.

Attributes

Name	Type	Description
Include	List[Component]	List of components which are used for filtering in the needed components
Exclude	List[Component]	List of components which are used for filtering out unneeded components

Operations

Operation: process(delta : Int)
Input : delta : Int - The time difference from the last frame
Output : None
Description : Calls processEntity and filters the list of entities

Operation: processEntity(entity : Entity, delta : Int)
Input : delta : Int - The time difference from the last frame entity : Entity - the filtered entity needed for processing
Output : None
Description : Calls entities one by one and processes the information based on implementation

3.1.5 TimedSystem

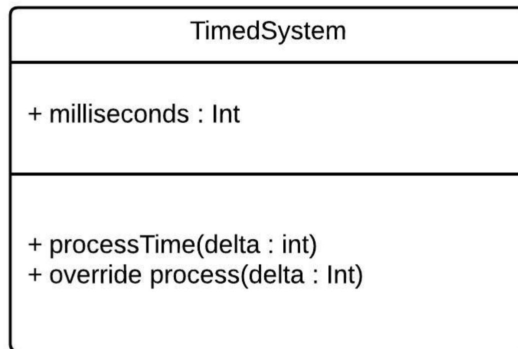

Figure 8: Timed System Class Diagram

A TimedSystem only runs after the amount of time given to it. Used for processing that needs to be done on a timed interval.

Attributes

Name	Type	Description
milliSeconds	Int	Amount of time that must pass before system processes again
start	Int	The time when the system started counting for next run

Operations

Operation: process(delta : Int)
Input : delta : Int - The time difference from the last frame
Output : None
Description : Calls processTime and checks to see if enough time has passed

Operation: processTime(delta : Int)
Input : delta : Int - The time difference from the last frame entity : Entity - the filtered entity needed for processing Output : None
Description : Is called after certain amount of time given by milliseconds.

3.1.6 IntervalSystem

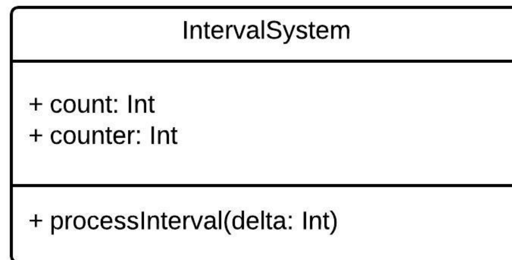

Figure 9: System Class Diagram

A IntervalSystem only runs after a certain amount of frames has passed. Used for processing that needs to be done on a frame interval.

Attributes

Name	Type	Description
count	Int	The amount of frames that must pass before the system processes again
counter	Int	The current amount of frames that have been passed since the last run

Operations

Operation: process(delta : Int)
Input : delta : Int - The time difference from the last frame
Output : None
Description: Calls processInterval and checks to see if enough frames have passed

Operation: processInterval(delta : Int) Input : delta : Int - The time difference from the last frame entity : Entity - the filtered entity needed for processing Output : None
Description : Is called after certain amount of frames have been passed.

3.1.7 Component

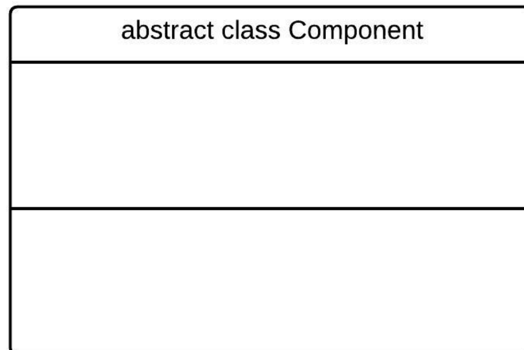

Figure 10: Component Class Diagram

Component is an empty class, but is used as an identifier for grouping data together.

3.1.8 Game Loop

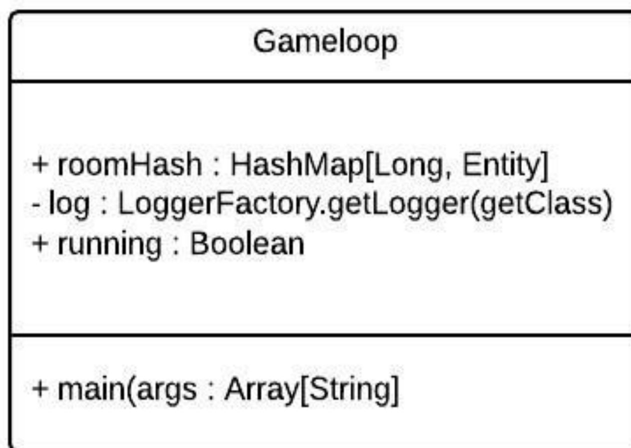

Figure 11: The Game Loop

GameLoop.scala is the main driver of the Ayai framework. It loads in all Constants, maps, and compiles the rooms together, and sets all worlds with the appropriate systems and information.

Attributes

Name	Type	Description

roomHash	HashMap[Long, Entity]	A map of the roomId to the RoomEntity and its Map information components
log	Logger	A logger which allows for printing to a log file
running	Boolean	Is the main loop still running

Operations

Operation: main

Input : None

Output : None

Description : Sets up the worlds needed to run the game, sets up all network connections, and loads all rooms from files.

3.2 Components

Components are aspects of entities. An entity is comprised of one or more components which specify behaviors that the entity might have. For example a player entity would be comprised of a position, bounds, health, inventory, mana, and character component.

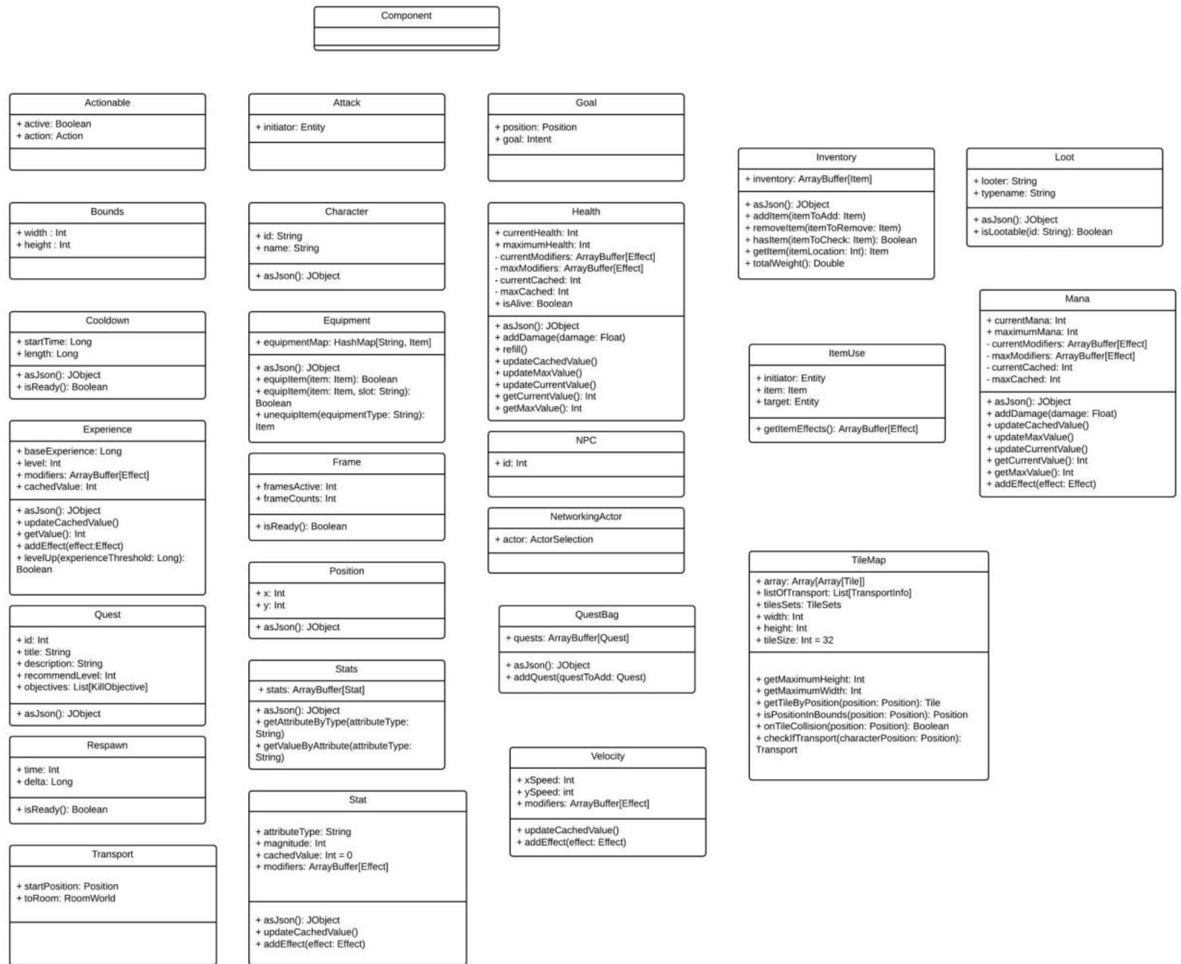

Figure 12: All components inheriting from component

3.2.1 Position

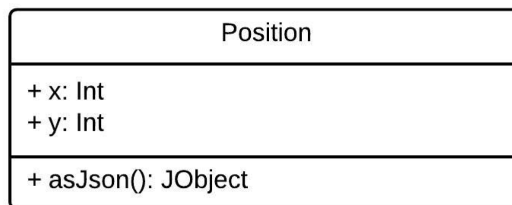

Figure 13: Position Class Diagram

Name	Type	Description
x	Int	The position on the x-coordinate plane of the entity
y	Int	The position on y-coordinateplane entity

3.2.2 Actionable

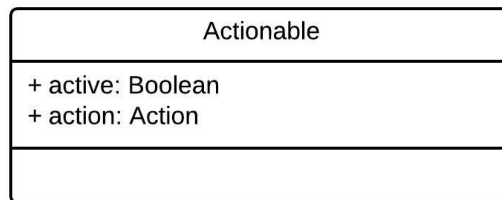

Figure 14: Actionable Class Diagram

Name	Type	Description
active	Boolean	Is the component in an active state
action	Action	The action that the component is doing

3.2.3 Attack

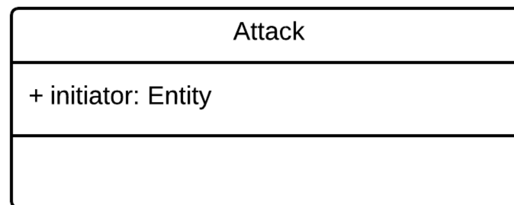

Figure 15: Attack Class Diagram

Name	Type	Description
initiator	Int	Who initiated the attack
victims	ArrayBuffer[Entity]	List of entities of who the attack has collided with

3.2.4 Bounds

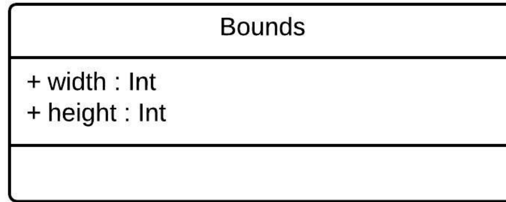

Figure 16: Bounds Class Diagram

Name	Type	Description
width	Int	The total width of the bounding box
height	Int	The total height of the bounding box

Name **Type** **Description**

3.2.5 Character

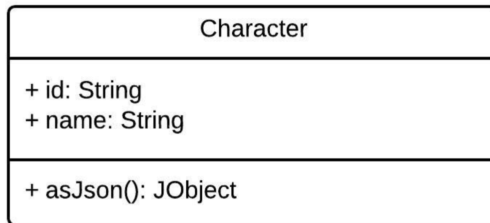

Figure 17: Character Class Diagram

Name	Type	Description
id	String	The unique string of the character
name	String	The name of the character

3.2.6 Frame

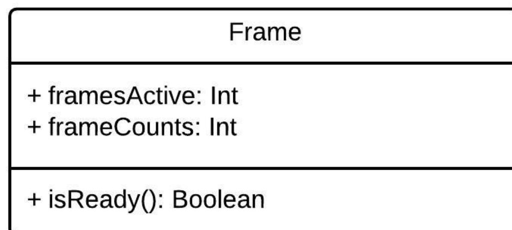

Figure 18: Frame Class Diagram

Name	Type	Description
framesActive	Int	The amount of frames that must be passed to run again
frameCounts	Int	The current amount of frames that have passed

3.2.7 Health

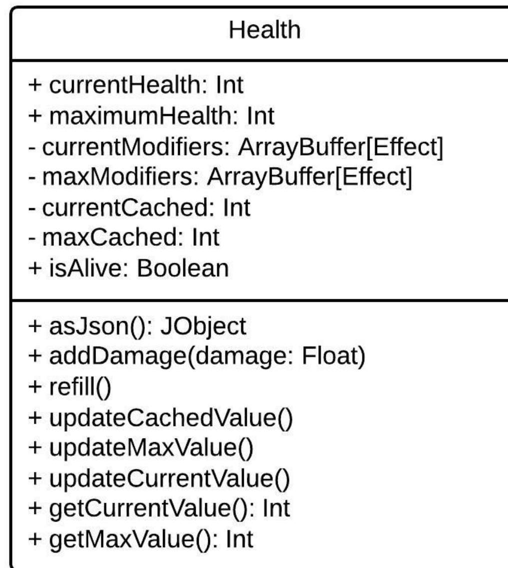

Figure 19: Health Class Diagram

Name	Type	Description
currentHealth	Int	The current value of health
maximumHealth	Int	The maximum amount of health value
currentModifiers	ArrayBuffer[Effect]	The current effects that are effecting the currentHealth value
maxModifiers	ArrayBuffer[Effect]	The current effects that are effecting the maximumHealth value
currentCached	Int	The value of currentHealth with all effects calculated
maxCached	Int	The value of maximumHealth with all effects calculated

isAlive	Boolean	Is current health less than zero

Operation: addDamage(damage: Float)
Input : damage: Float -the amount of damage to subtract from the currentHealth
Output : None
Description : Calculate damage to subtract from currentHealth

Operation: refill()
Input : None
Output : None
Description : Sets the currentHealth to maximumHealth

Operation: updateCachedValue()
Input : None
Output : None
Description: Updates the cached values of both maximum Health and currentHealth

Operation: updateMaxValue()
Input : None
Output : None
Description : Updates the cached values of maximumHealth by processing the effects on the component

Operation: updateCurrentValue()
Input : None
Output : None
Description : Updates the cached values of currentHealth by processing the effects on the component

Operation: getCurrentValue()
Input : None
Output : Returns the cached value for currentHealth
Description : Returns the cached value for currentHealth

Operation: getMaxValue()
Input : None
Output : Returns the cached value for maximumHealth
Description : Returns the cached value for maximumHealth

3.2.8 Inventory

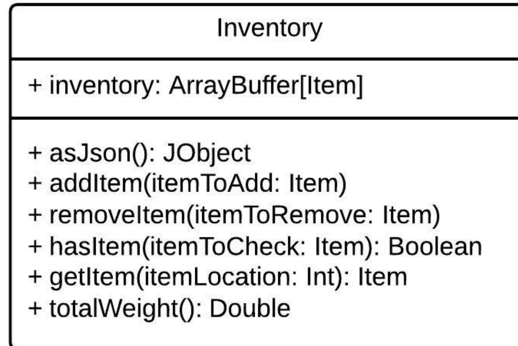

Figure 20: Inventory Class Diagram

Name	Type	Description
inventory	ArrayBuffer[Item]	A list of items

Operation: addItem(itemToAdd: Item)
Input : Item to add to inventory list Output :
None
Description : Adds Item to inventory

Operation: removeItem(itemToRemove: Item)
Input : Item to remove from inventory list Output :
None
Description : Removes Item from inventory

Operation: hasItem(itemToCheck: Item): Boolean
Input : Item to check in inventory list
Output : Returns if item exists in list
Description : Checks to see if given item exists in list

Operation: getItem(itemLocation: Int): Item
Input : The slot that the item exists in Output :
Returns the item
Description : Retrieves item from list

Operation: totalWeight(): Int
Input : None
Output : Returns total weight of inventory
Description : Returns the weight of all items in inventory

3.2.9 Velocity

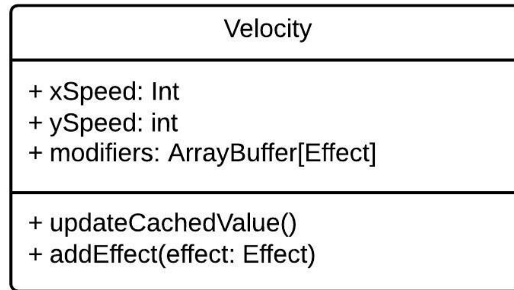

Figure 21: Velocity Class Diagram

Name	Type	Description
xSpeed	Int	Speed in the xDirection
ySpeed	Int	Speed of the y direction
modifiers	ArrayBuffer[Effect]	The current effects that are effecting both speed values

Operation: addEffect(effect: Effect)
Input : effect: Effect - the effect to add Output :
None
Description : Adds effect to modifiers

Operation: updateCachedValue()
Input : None
Output : None
Description : Updates the cached value

3.2.10 Time

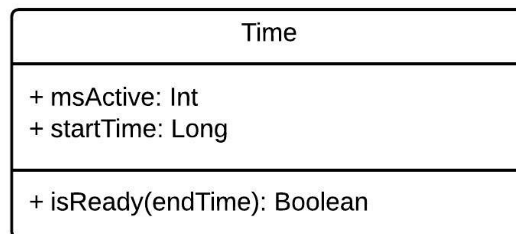

Figure 22: Time Class Diagram

Name	Type	Description
msActive	Int	The amount of msSeconds until the component is activated
startTime	Long	The time of last frame ending

3.2.11 Mana

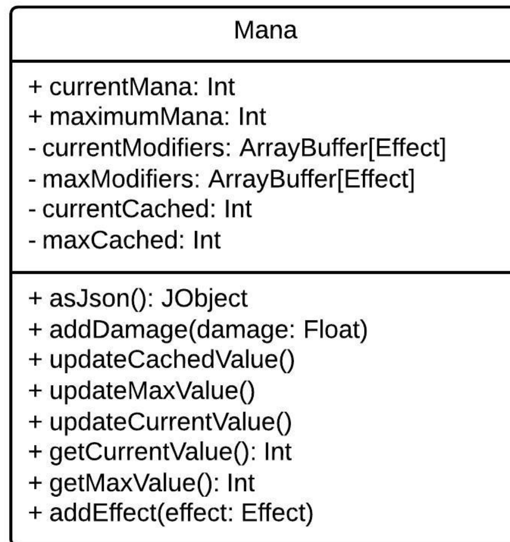

Figure 23: Mana Class Diagram

Name	Type	Description
currentMana	Int	The current value of health
maximumMana	Int	The maximum amount of health value
currentModifiers	ArrayBuffer[Effect]	The current effects that are effecting the currentMana value
maxModifiers	ArrayBuffer[Effect]	The current effects that are effecting the maximumMana value
currentCached	Int	The value of currentMana with all effects calculated
maxCached	Int	The value of maximumMana with all effects calculated

isAlive	Boolean	Is current health less than zero

Operation: addDamage(damage: Float)

Input : damage: Float - the amount of damage to subtract from the currentMana
Output : None
Description : Calculate damage to subtract from currentMana

Operation: updateCachedValue()

Input : None
Output : None
Description : Updates the cached values of both maximumMana and current-Mana

Operation: updateMaxValue()

Input : None
Output : None
Description : Updates the cached values of maximumMana by processing the effects on the component

Operation: updateCurrentValue()

Input : None
Output : None
Description : Updates the cached values of currentMana by processing the effects on the component

Operation: getCurrentValue()

Input : None
Output : Returns the cached value for currentMana
Description : Returns the cached value for currentMana

Operation: getMaxValue()

Input : None
Output : Returns the cached value for maximumMana
Description : Returns the cached value for maximumMana

Operation: addEffect(effect: Effect)

Input : effect: Effect - the effect to add
Output : None
Description : Adds effect to modifiers (modifier depends on type in effectType)

3.2.12 Stats

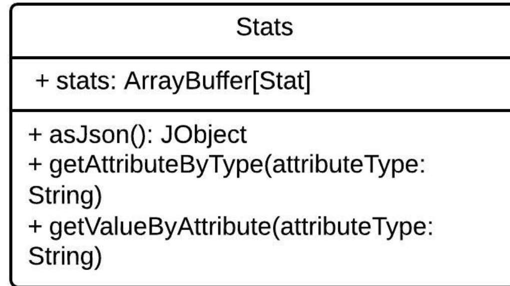

Figure 24: Stats Class Diagram

Name	Type	Description
stats	ArrayBuffer[Stat]	List of stats

Operation: updateCachedValue()
Input : None
Output : None
Description : Updates the cached values of all stored stats

Operation: getValueByAttribute(attributeType: String): Int
Input : Based on attribute type return the value
Output : Returns the current cached value of the given attribute
Description : Returns the current cached value of the given attribute

3.2.13 Stat

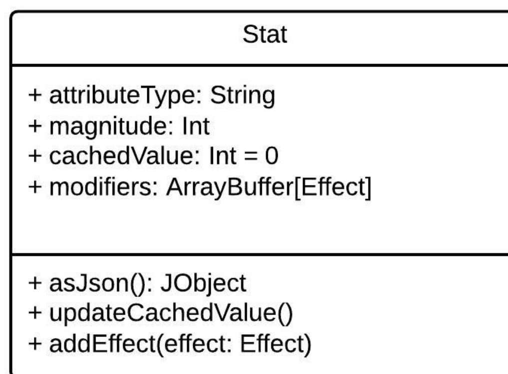

Figure 25: Stat Class Diagram.

Name	Type	Description
attributeType	String	The string of an attribute
magnitude	Int	Current value of attribute
cachedValue	Int	Current cached value of attribute
modifiers	ArrayBuffer[Effect]	The current effects that are effecting the stat

Operation: updateCachedValue()
Input : None
Output : None
Description : Updates the cached value of the stat by processing the effects on the component

Operation: addEffect(effect: Effect)
Input : effect: Effect - the effect to add
Output : None
Description : Adds effect to modifiers (modifier depends on type in effectType)

3.2.14 Transport

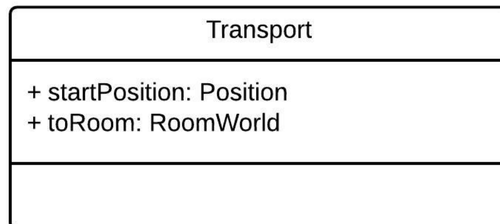

Figure 26: Transport Class Diagram

Name	Type	Description
toRoom	Room	Specifies the room to which to transport
startPosition	Position	Specifies the position in toRoom

3.2.15 NetworkingActor

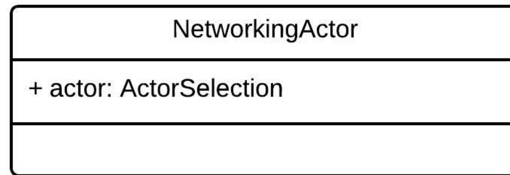

Figure 27: Networking Actor Class Diagram

Name	Type	Description
actor	ActorSelection	The connection to the receiving player

3.2.16 Respawn

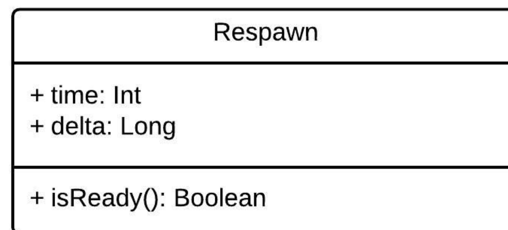

Figure 28: Respawn Class Diagram

Name	Type	Description
time	Int	Defaulted to 1500 ms, and is the amount of time until player can respawn
delta	Long	The time that a player died

3.2.17 TileMap

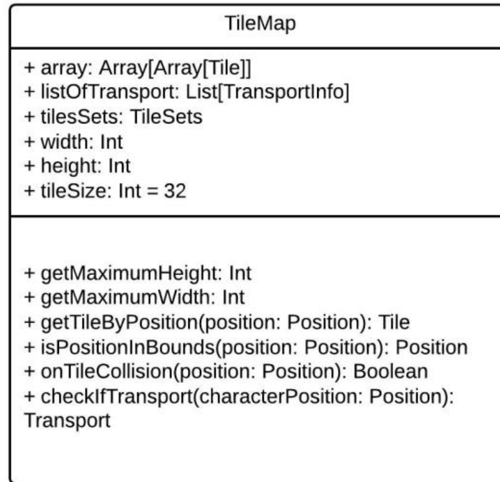

Figure 29: TileMap Class Diagram

Name	Type	Description
array	Array[Array[Tile]]	Dimensional Array of tiles
listOfTransport	List[TransportInfo]	A list of transport locations on a map
tileSets	TileSets	A list of tileset files
file	String	the JSON file that represents the map
width	Int	the width of tiles of map
height	Int	the height of tiles of map
tileSize	Int	number of pixels of an individual tile

Operations

Operation: getMaximumHeight() : Int

Input : None

Output : Number of pixels in height

Description : Returns the height multiplied by the tileSize to get the number of pixels in the y-axis

Operation: getMaximumWidth() : Int

Input : None

Output : Number of pixels in width

Description : Returns the width multiplied by the tileSize to get the number of pixels in the x-axis

Operation: getTileByPosition(position : Position) : Tile
Input : position : Position - the position to convert to tile
Output : The tile referenced by position
Description : Returns the tile that is in the area of the given position

Operation: valueToTile(value : Int) : Int
Input : a pixel location
Output : the value divided by tileSize
Description : Returns the value given divided by tileSize

Operation: isPositionInBounds(position : Position) : Position
Input : position : Position - the position to check
Output : returns new position, if old value was not valid
Description : Given a position, checks to see if tile location is not valid, and returns a valid position

Operation: onTileCollision(position : Position) : Boolean
Input : position : Position - the position to check
Output : returns true/false if position is on unwalkable tile
Description : Given a position, checks to see if tile location is valid

Operation: checkIfTransport(characterPosition : Position) : Transport
Input : characterPosition : Position - the position to check
Output : Returns a transport object if tile is a transport tile
Description : Given a position, checks to see if tile location is a transportable tile and returns the information

3.2.18 ItemUse

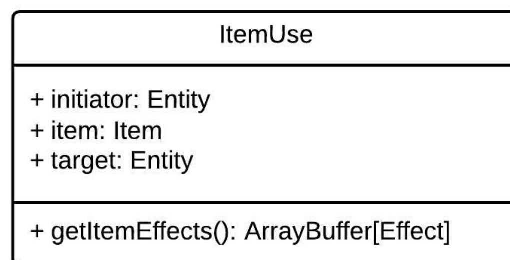

Figure 30: ItemUse Class Diagram

The ItemUse component is acted upon by the ItemSystem and is used to convey information about when items are used by a player.

Name	Type	Description
initiator	Entity	What entity used the item
item	Item	the item that was used
target	Entity	What entity was targeted by the initiator

Operations

Operation: getItemEffects() : ArrayBuffer[Effect

Input : None

Output : The list of effects on an item

Description : Returns the list of effects that an item has on them (would be processed by the ItemSystem)

3.2.19 Experience

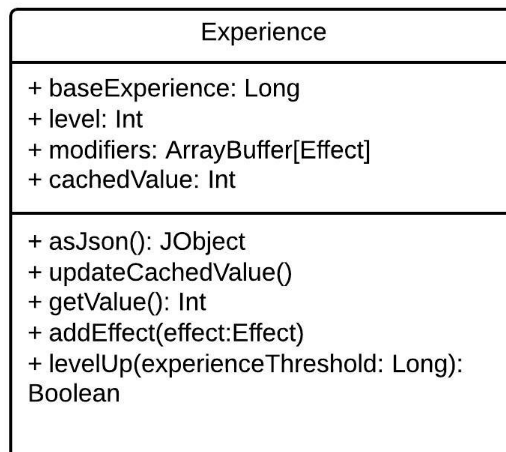

Figure 31: Experience Class Diagram

Experience is gained from when players complete tasks or kill enemies. When a player gains enough experience then they can level up and adds more power to their stats.

Name	Type	Description
baseExperience	Long	The total amount of experience
level	Int	Current level of entity
modifiers	ArrayBuffer[Effect]	The current effects that are effecting baseExperience

Operation: updateCachedValue()
Input : None
Output : None
Description : Updates the cached value of experience

Operation: getValue(): Int
Input : None
Output : Returns the cached value for experience
Description : Returns the cached value for experience

Operation: levelUp(experienceThreshold: Long): Boolean
Input : The threshold for the next level
Output : Returns if the player has leveledUp
Description : Checks to see if the players baseExperience is higher than the experience threshold of the next level

Operation: addEffect(effect: Effect)
Input : effect: Effect - the effect to add
Output : None
Description : Adds effect to modifiers (modifier depends on type in effectType)

3.2.20 Cooldown

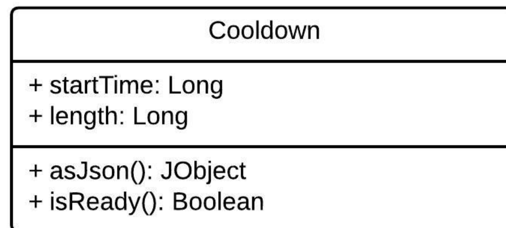

Figure 32: Cooldown Class Diagram

Keeps a time to see if a player can perform another action. If the cooldown is active then a player cannot do an action such as attack or use an item. Is acted up by the cooldown system.

Name	Type	Description
startTime	Long	The start time when the cooldown was set
length	Long	Length in seconds for how long cooldown will last

Operation: isReady(): Boolean

Input : None

Output : Returns if enough time has passed

Description : Returns to see if enough time has passed and cooldown is down

3.2.21 Quest

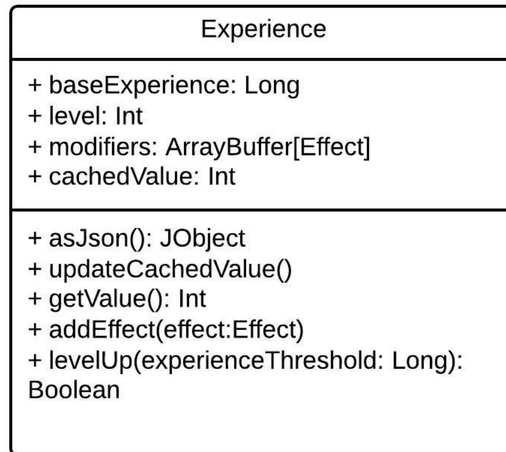

Figure 33: Experience Class Diagram

Information about quests and objectives to complete in the game (is not a component, but is used with quest bag)

Name	Type	Description
id	Int	The quest id
title	String	Title of the quest
description	String	The description and details of the quest
recommendLevel	Int	The recommended level that a player should be to do the quest
objectives	List[KillObjective]	The objectives to complete the quest

Operation: isReady(): Boolean
Input : None
Output : Returns if enough time has passed
Description : Returns to see if enough time has passed and cooldown is down

3.2.22 QuestBag

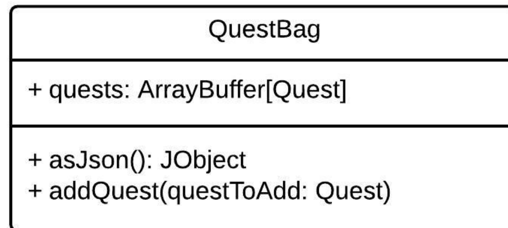

Figure 34: QuestBag Class Diagram

Is a component that holds information about a players held quests

Name	Type	Description
quests	ArrayBuffer[Quest]	An entities held quests

Operation: addQuest(questToAdd: Quest)
Input : The quest to add to quests list Output :
None
Description : Adds quest to quests list

3.2.23 Equipment

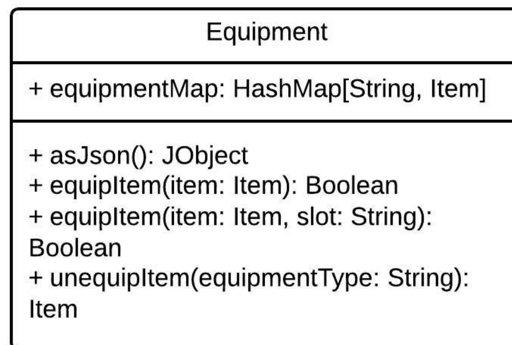

Figure 35: Equipment Class Diagram

The player's equipment is what allows them to greatly increase their stats by providing the ability to equip weapons and armor.

Name	Type	Description
equipmentMap	HashMap[String, Item]	Maps an item slot to an item

Operation: equipItem(item: Item): Boolean

Input : The item to equip

Output : Returns if the equip was successful

Description : Tries to equip an item based on the items information, will return false if failed

Operation: equipItem(item: Item, slot: String): Boolean

Input : The item to equip and the slot to equip to

Output : Returns if the equip was successful

Description : Tries to equip an item based on the slot given, will return false if failed

Operation: equipItem(equipmentType: String): Item

Input : The slot to unequip from

Output : Returns the item that was unequiped

Description : Tries to unequip an item based on the slot given, will return the item that was in the slot

3.3 Items

Items are used throughout the game as potentially quest items, weapons, armor, or consumables (potions, mana potions, and stat increases or decreases)

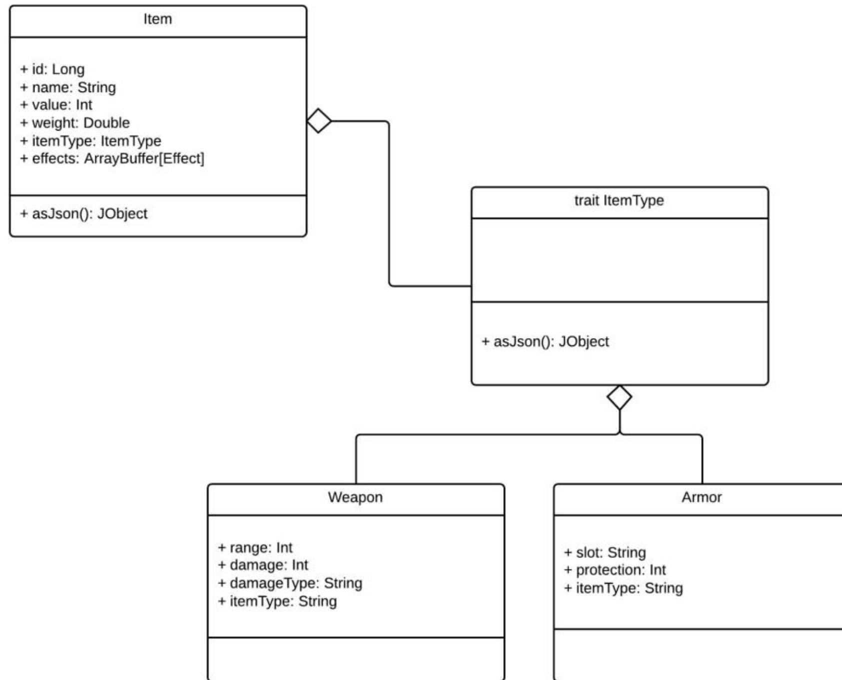

Figure 36: Items

3.3.1 Item

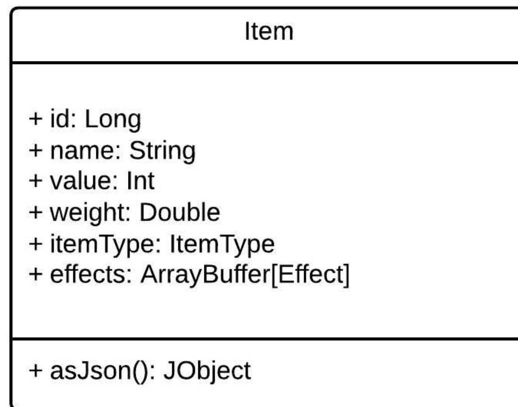

Figure 37: Item Class Diagram

Attributes

Item is a class that holds all information about an item including its effects and descriptions. When an item is used it can either be equipped by a player (based on item type) or be used by a player to perform an action.

Name	Type	Description
id	Long	the item id
name	String	The name of the item
value	Int	The value that the item will use when consumed or equipped
weight	Double	The weight of the item
itemType	ItemType	Additional information about an item such as weapon or armor.
effects	ArrayBuffer[Effect]	The effects that an item will do when used or equipped

3.3.2 ItemType

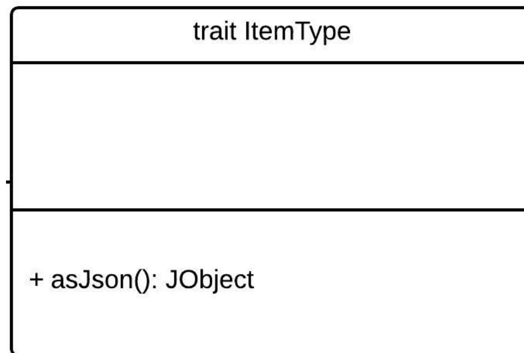

Figure 38: ItemType Class Diagram

ItemType is an abstract class that can be extended to hold additional information for items. Also contains asJson function export information from needed class.

3.3.3 Weapon

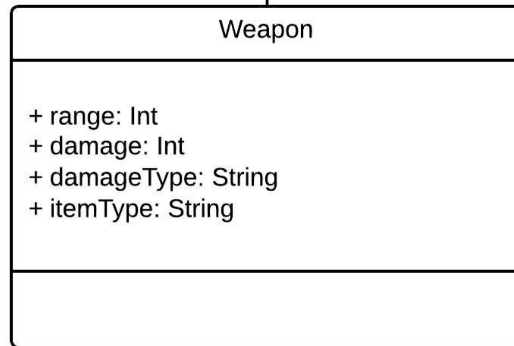

Figure 39: Weapon Class Diagram

Attributes

A weapon can be equipped in the weapon1 or weapon2 equipment slots. Raises a players offensive stats.

Name	Type	Description
range	Int	the number of pixels the attack can be extended
damage	Int	The amount of damage the attack will do
damageType	String	The type of damage the weapon will inflict (only physical)
itemType	String	The slot that it will be equipped onto (weapon1 or weapon2)

3.3.4 Weapon

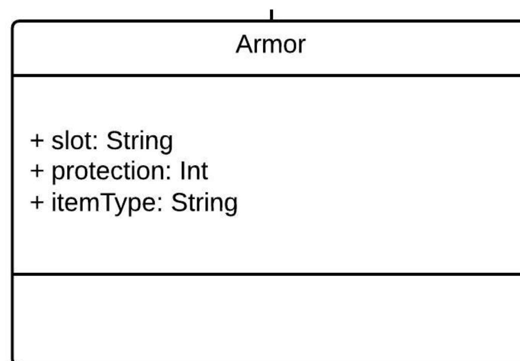

Figure 40: Armor Class Diagram

Attributes

An armor can be equipped in the head, torso, legs, or feet equipment slots. Raises a players defensive stats.

slot	String	The slot that will be equipped to
damage	Int	The amount of protection that will raise the defense stat
itemType	String	The slot that it will be equipped to

3.4 Systems

Systems, as described in Section 3.1.3, are used to manipulate component data.

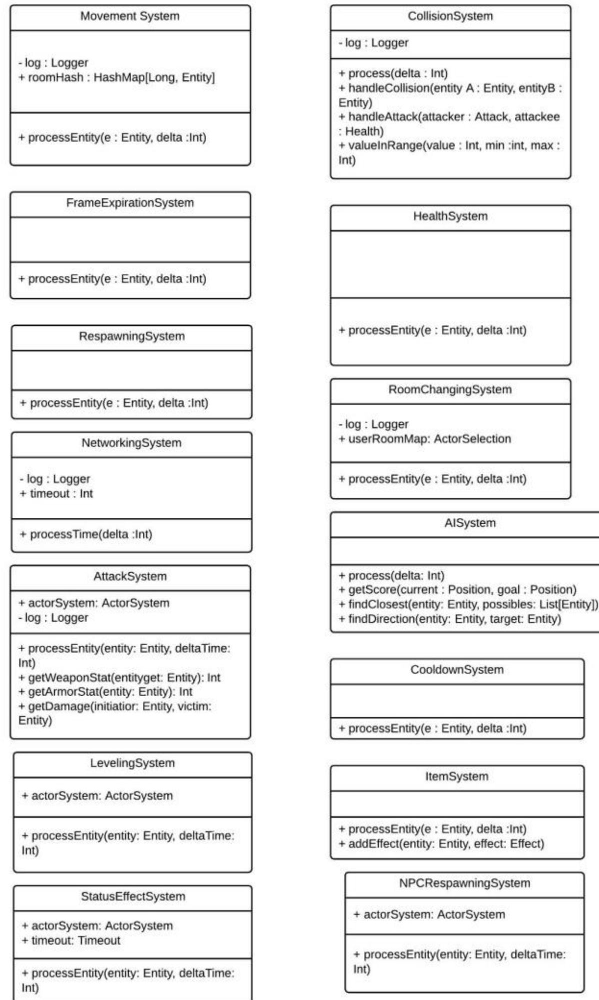

Figure 41: Systems

3.4.1 MovementSystem

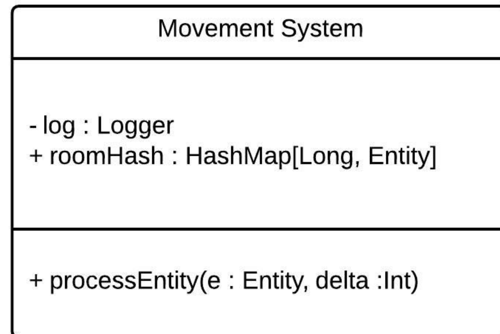

Figure 42: MovementSystem Class Diagram

Attributes

The movement system inherits the EntityProcessingSystem and requires an Entity to have the Position, Velocity, Actionable, and Character components.

Name	Type	Description
roomHash	HashMap[Long, Entity]	A map of the roomId to the RoomEntity and its Map information components
log	Logger	A logger which allows for printing to a log file

Operations

Operation: processEntity(e : Entity, delta : Int)

Input : e : Entity - entity which possesses the necessary components delta : Int - time difference from last frame

Output : None

Description : Checks to see if the player is moving, then retrieves the room the player is in, and then checks to see if the position the player is in is valid and then attaches transport component to entity.

3.4.2 CollisionSystem

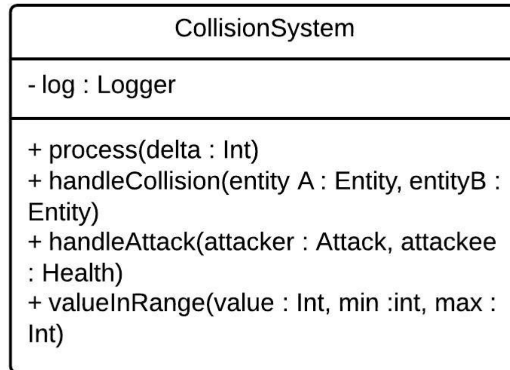

Figure 43: Collision System Class Diagram

Attributes

The collision system inherits from the normal System class and goes through each ROOM to gather its entities and uses QuadTrees to find entities which it may interact with. After finding eligible items it does collision detection and does the required actions (whether if an attack is colliding, or two players touching).

Name	Type	Description
log	Logger	A logger which allows for printing to a log file

Operations

Operation: process(delta : Int)
Input : delta : Int - time difference from last frame
Output : None
Description : Puts all room entities in quadtree, then retrieves each section of quadtree and runs collision detection.

Operation: handleCollision(entityA : Entity, entityB: Entity) Input :
entityA : Entity - first entity for collision checking
entityB : Entity - second entity for collision checking
Output : None
Description : Checks to see if the two entities overlap

Operation: handleCollision(attacker : Attack, attackee : Health)

Input : attacker : Attack - attack component which calculates damage done to attackees health

attackee : Health - health of victim, which damage is reduced from

Output : None

Description : Handles damage calculation of colliding attack and character entities

Operation: valueInRange(value : Int, min : Int, max :Int) : Boolean

Input : value : Int - value to see if between min and max min : Int - Bounds in which value must be greater than max :Int - Bounds in which value must be less

Output : Detects if given components are in range of each other

Description : Checks to see if value given is between the min and max

Operation: excludeList(entities: List[Entity],exclusionList: List[T]):List[Entity]

Input : entities : List[Entity] - list of entities exclusionList : List[T] - list of components to exclude

Output : Returns the list of entities that do not contain components from exclusionList

Description : Filters out exclusionList from list of entities

Operation: hasExclusion(entity : Entity, exclusionList : List[T]) : Boolean

Input : entity : Entity - entity to check

exclusionList : List[T] - list of components to exclude

Output : Returns if the entity contains any of the excluded components Description : Checks entity components to see if it contains any components from exclusion list

3.4.3 HealthSystem

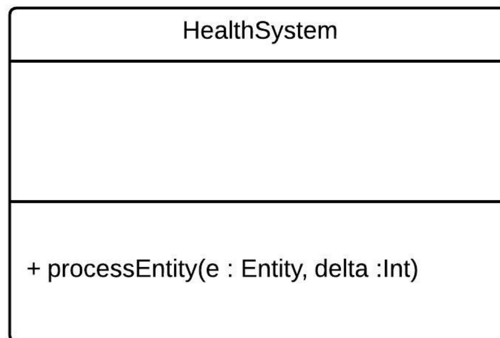

Figure 44: HealthSystem Class Diagram

Health System inherits from EntityProcessingSystem and checks Entity health to see if it should be killed and removed from game.

Operations

Operation: processEntity(entity : Entity, delta : Int)
Input : entity : Entity - entity to process delta : Int - time difference from last frame
Output : None
Description : Processes entity health

3.4.4 RespawningSystem

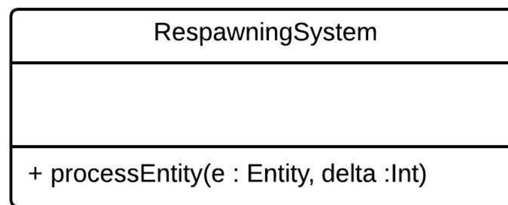

Figure 45: Respawning System Class Diagram

RespawningSystem inherits from EntityProcessingSystem and checks Entities who are dead and respawns the characters.

Operations

Operation: processEntity(entity : Entity, delta : Int)
Input : entity : Entity - entity to process delta : Int - time difference from last frame
Output : None
Description : Processes entity respawn

3.4.5 FrameExpirationSystem

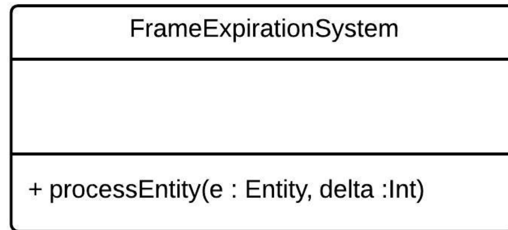

Figure 46: FrameExpirationSystem Class Diagram

FrameExpirationSystem inherits from EntityProcessingSystem and checks Entities that contain a Frame component and check to see if action is needed.

Operations

Operation: processEntity(entity : Entity, delta : Int)
Input : entity : Entity - entity to process delta : Int - time difference from last frame
Output : None
Description : Processes entity and checks frame component

3.4.6 NetworkingSystem

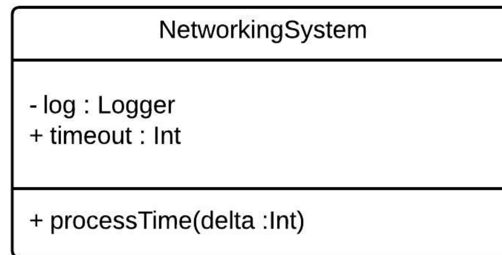

Figure 47: NetworkingSystem Class Diagram

Attributes

The networking system inherits the TimedSystem and after a certain amount of time updates all game players.

Name	Type	Description
roomHash	HashMap[Long, Entity]	A map of the roomId to the RoomEntity and its Map information components
log	Logger	A logger which allows for printing to a log file
timeout	Int	Time that a message has to compile

Operations

Operation: processTime(delta : Int)
Input : delta : Int - time difference from last frame
Output : None
Description : Processes compiling of messages and sending of messages to players

3.4.7 NPCRespawningSystem

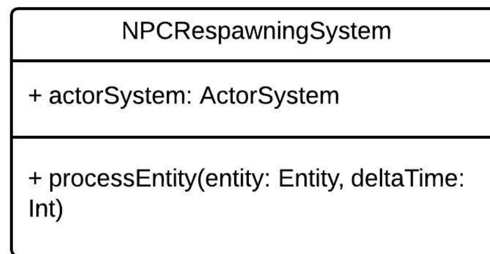

Figure 48: NPCRespawningSystem Class Diagram

Attributes

The NPCRespawningSystem is responsible for restoring any NPCs that need to be respawned and which are designated as being able to respawn.

Name	Type	Description
actorSystem	ActorSystem	Holds actors in game that allows system to query for information

Operations

Operation: processEntity(delta : Int)
Input : delta : Int - time difference from last frame
Output : None
Description : Processes compiling of messages and sending of messages to players

3.4.8 LevelingSystem

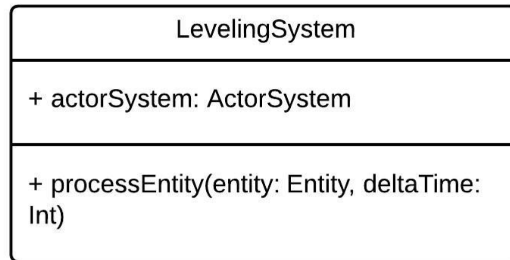

Figure 49: LevelingSystem Class Diagram

Attributes

The LevelingSystem is meant to calculate a players experience and determine if levelup is needed.

Name	Type	Description
actorSystem	ActorSystem	Holds actors in game that allows system to query for information

Operations

Operation: processEntity(delta : Int)
Input : delta : Int - time difference from last frame
Output : None
Description : Processes compiling of messages and sending of messages to players

3.4.9 StatusEffectSystem

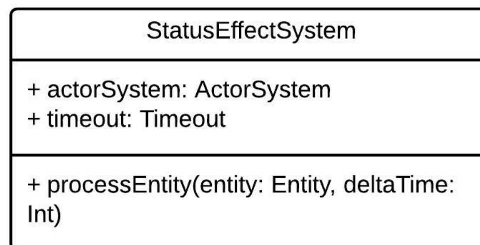

Figure 50: StatusEffectSystem Class Diagram

Attributes

The StatusEffectSystem is meant to calculate all status effects on a character per cycle and determine if the effects should be removed and calculate all values needed throughout the cycle.

Name	Type	Description
actorSystem	ActorSystem	Holds actors in game that allows system to query for information

Operations

Operation: processEntity(delta : Int)

Input : delta : Int - time difference from last frame

Output : None

Description : Processes compiling of messages and sending of messages to players

3.4.10 CooldownSystem

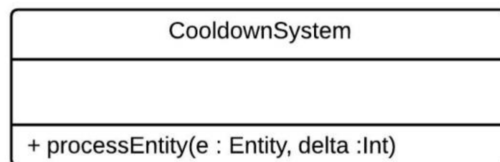

Figure 51: CooldownSystem Class Diagram

. **Attributes**

The cooldown system works with the cooldown component to stop players from attacking or using items too quickly

Name	Type	Description

actorSystem	ActorSystem	Holds actors in game that allows system to query for information

Operations

Operation: processEntity(delta : Int)

Input : delta : Int - time difference from last frame

Output : None

Description : Processes compiling of messages and sending of messages to players

3.4.11 ItemSystem

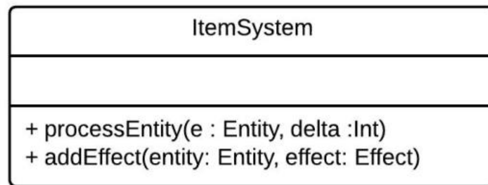

Figure 52: ItemSystem Class Diagram

Attributes

The ItemSystem checks to see if any items need to be processed on characters.

Name	Type	Description
actorSystem	ActorSystem	Holds actors in game that allows system to query for information

Operations

Operation: processEntity(delta : Int)

Input : delta : Int - time difference from last frame

Output : None

Description : Processes compiling of messages and sending of messages to players

3.4.12 AttackSystem

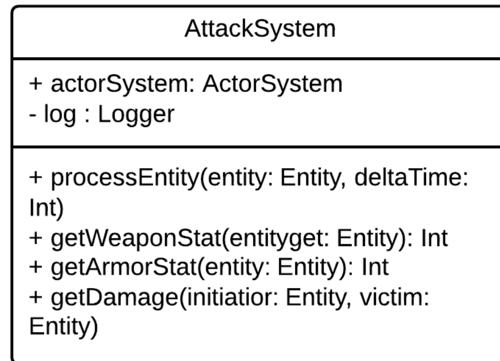

Figure 53: AttackSystem Class Diagram

<u>**Attributes**</u>

The AttackSystem processes attack messages from the processors

Name	Type	Description
actorSystem	ActorSystem	Holds actors in game that allows system to query for information

<u>**Operations**</u>

Operation: processEntity(delta : Int)
Input : delta : Int - time difference from last frame
Output : None
Description : Processes compiling of messages and sending of messages to players

Operation: getWeaponStat(entityGet: Entity)
Input : entity to get stat off of
Output : Outputs the total attack damage
Description : Processes all attack stats that a player has on them and compiles them together

Operation: getArmorStat(entityGet: Entity)

Input : entity to get stats off of

Output : Outputs the total defensive value

Description : Processes all defensive stats that a player has on them and compiles them together

Operation: getDamage(initiator: Entity, victim: Entity)

Input : initiator - entity who initiated attack victim - person who was attacked Output : None

Description : Processes both defensive and attack stats that a victim and initiator have and compiles damage to receive on victim.

3.4.13 RoomChangingSystem

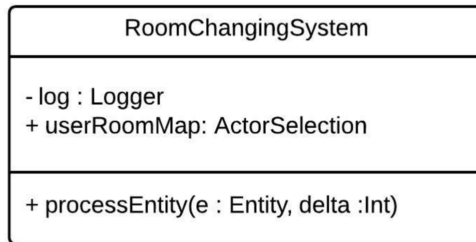

Figure 54: RoomChangingSystem Class Diagram

Attributes

The RoomChangingSystem inherits the EntityProcessingSystem and checks to see that if an Entity contains a "Transport" component and changes the processing entities room.

Name	Type	Description
roomHash	HashMap[Long, Entity]	A map of the roomId to the RoomEntity and its Map information components
log	Logger	A logger which allows for printing to a log file

Operations

Operation: processEntity(entity : Entity, delta : Int)

Input : entity : Int - entity to process, delta : Int - time difference from last frame

Output : None
Description : Processes and sends all player messages

3.4.14 AISystem

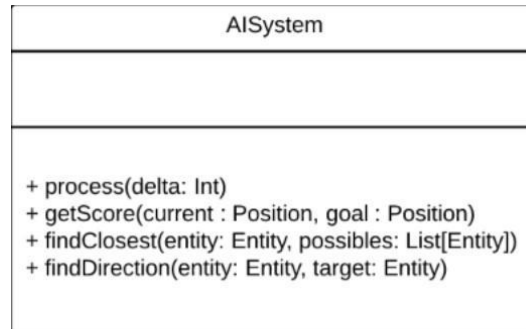

Figure 55: AISystem Class Diagram

The AI system inherits from the normal System and calculates all artificial intelligence based decisions.

<u>**Operations**</u>

Operation: process(delta : Int)
Input : delta : Int - time difference from last frame
Output : None
Description : Calculates AI commands

Operation: getScore(current : Position, goal : Position)
Input : current : Position - current position of AI agent
Input : goal : Position - current position of target
Output : score : Int
Description : Calculates score based on distance from target

findClosest(entity: Entity, possibles: List[Entity])
Input : entity : Entity - Starting entity
Input : possibles : List[Entity]
Output : entity : Entity
Description : Returns closed entity to starting entity

findDirection(entity: Entity, target: Entity)
Input : entity : Entity - Starting entity

Input : target : Entity - Target entity
Output : MoveDirection
Description : Returns MoveDirection for entity to move towards target

3.5 Status Effects

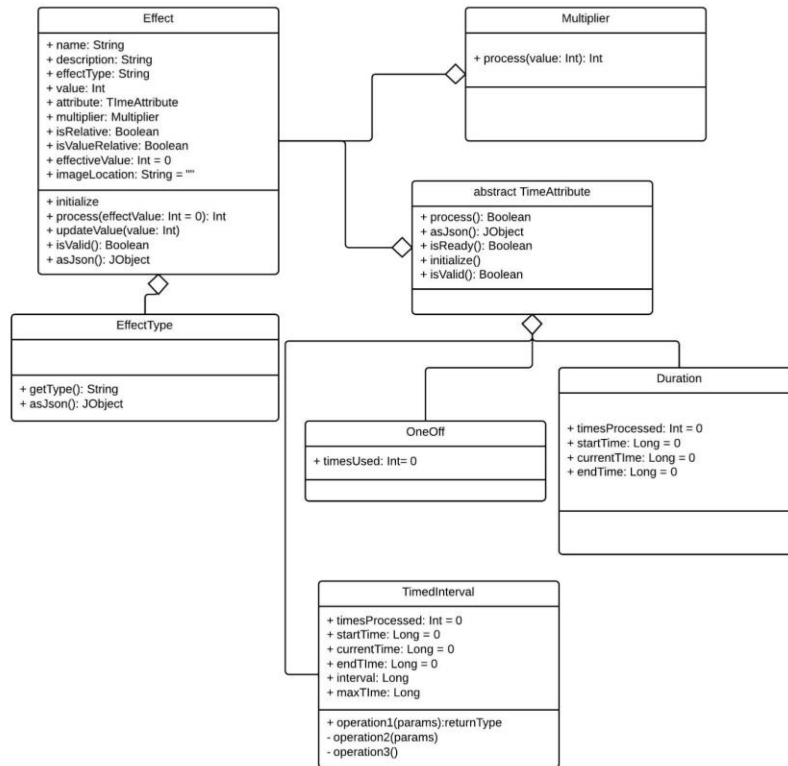

Figure 56: Ayai Status Effect

The status effect system comprises of an Effect class that takes in a Multiplier, EffectType string, and a Time Attribute. It can be used with 5 main components (Health, Mana, Stats, Experience, and Velocity).

3.5.1 Effect

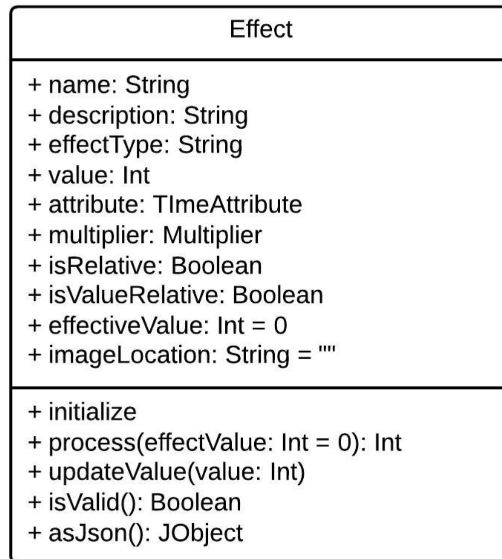

Figure 57: Effect Class Diagram

Attributes

The Effect class holds all information about an effect. Effects are used to change statistics for a temporary time by being attached to a weapon or used on an item.

Name	Type	Description
name	String	Name of the effect
description	String	Description of the effect
effectType	EffectType	Information about the effect
value	Int	Value that effect will use
attribute	TimeAttribute	Details of how often effect will run
isRelative	Boolean	Is the effect adding to the current value of the effected type (if false, the value will change to an absolute value)

isValueRelative	Boolean	is computed value determined by the current value of the type
effectiveValue	Int	The compiled value of the effect to use

Operations

Operation: process(effectValue: Int = 0)
Input : effectValue: Int - is defaulted to zero, but if the process is determined by an outside value (such as current health) then that value needs to be given
Output : Returns the computed value
Description : Processes and updates the effective value for the cycle or effect

Operation: updateValue(value: Int)
Input : value: Int - the outside value used to process
Output : None
Description : Updates effective value

Operation: isValid()
Input : None
Output : If effect is still valid
Description : Checks to see if effect is still valid by testing time attribute

3.5.2 TimeAttribute

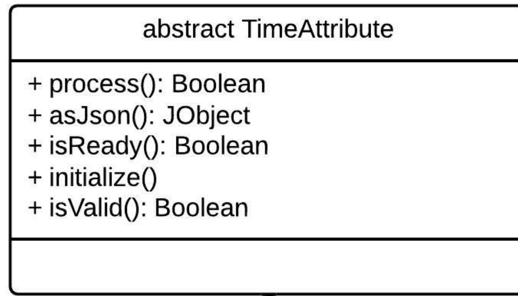

Figure 58: TimeAttribute Class Diagram

The TimeAttribute class is an abstract class that is used to determine how long and how much an effect needs to run.

Operations

Operation: process()
Input : None
Output : Returns if the value has been changed
Description : Processes the updated values of the time

Operation: isReady()
Input : None
Output : Returns if effect is ready to run
Description : Returns if the effect is ready to run

Operation: initialize()
Input : None
Output : None
Description : Sets all values to initial settings

Operation: isValid()
Input : None
Output : Boolean
Description : Returns if the effect should be removed from the game or character

3.5.3 OneOff

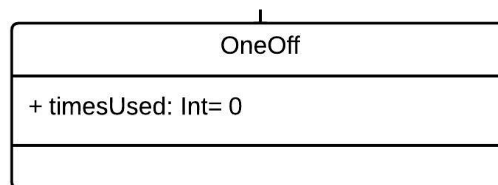

Figure 59: OneOff Class Diagram

Attributes

The OneOff class is extended from the TimeAttribute and is meant to run the effect immediately and only once. Once it has been run isValid and isReady will be true and false, respectively.

Name	Type	Description
timesUsed	Int	How many times has the effect been run

3.5.4 TimedInterval

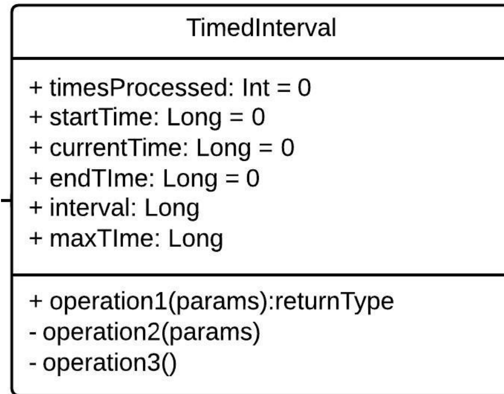

Figure 60: TimedInterval Class Diagram

Attributes

The TimedInterval class is extended from the TimeAttribute and is meant to be run at a set interval for a set amount of time.

Name	Type	Description
timesProcessed	Int	How many times has the effect been run
startTime	Long	When was the effect initialized
currentTime	Long	What is the currentTime
endTime	Long	When will the effect end
interval	Int	How many seconds should the effect be processed (in seconds)
maxTime	Long	How long should effect last for (in seconds)

3.5.5 Duration

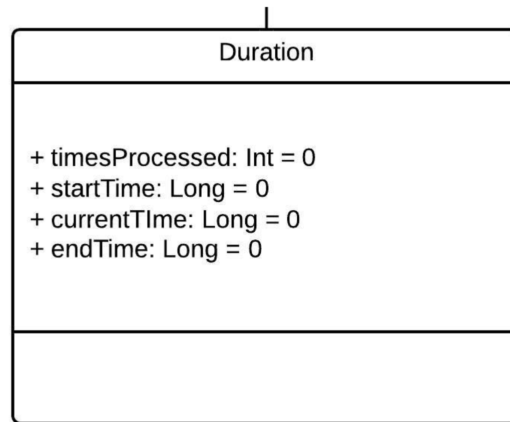

Figure 61: Duration Class Diagram

Attributes

The Duration class is extended from the TimeAttribute and is meant to be run for the length given. The effect is processed is meant to be run once, but is removed once the time is up (meant for temporary stat increases)

Name	Type	Description
timesProcessed	Int	How many times has the effect been run
startTime	Long	When was the effect initialized
currentTime	Long	What is the currentTime
endTime	Long	When will the effect end
maxTime	Long	How long should effect last for (in seconds)

3.5.6 Multiplier

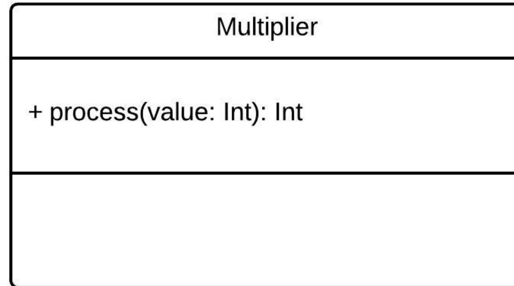

Figure 62: Multiplier Class Diagram

Attributes

The Multiplier class has an internal value which it will use to multiply with the effects value to create the effects effective value. For example, if the multiplier value is .5 and the effect wants to use current health's value, it will decrease the value to half of what it was.

Name	Type	Description
value	Float	The multiplier value to use with the effect value

Operations

Operation: process(effectValue: Int = 0)
Input : effectValue: Int - the value to multiply
Output : Returns the multiplied value
Description : Multiplies the given value by the multiplier value

3.6 Movement Processes

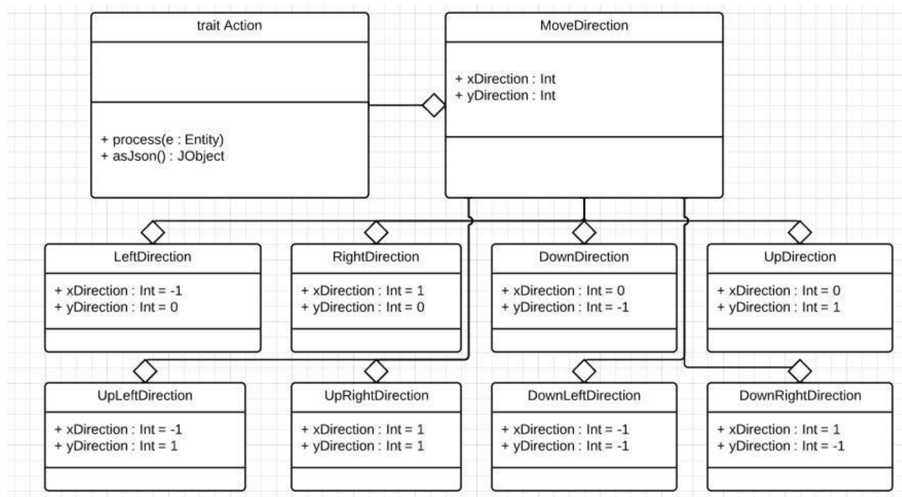

Figure 63: Action System Diagram

Movement in the Ayai framework are based on an Action trait. These actions are used for players and are currently only used to process Movements. Movements consist of an X and Y direction and the process function moves the entity in the needed direction.

3.6.1 Action

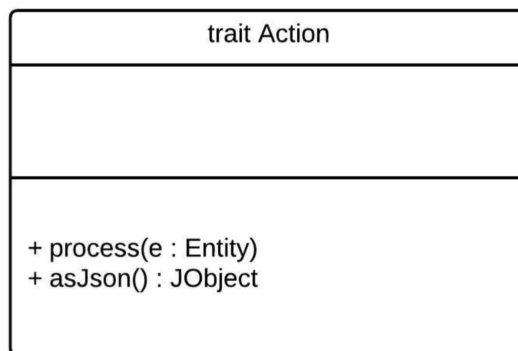

Figure 64: Action Class Diagram

The action is a trait that has a process function and an asJson function. The process function is used to process the given entity and asJson is to print out the state of the action.

3.6.2 MovementDirection

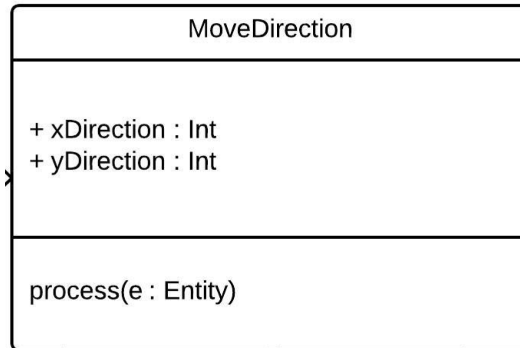

Figure 65: MoveDirection Class Diagram

The movement direction consists of an X and Y direction and the process function of MovementDirection takes the X and Y direction and multiples the user's velocity component to move in the correct direction.

3.6.3 MovementDirection Case Classes

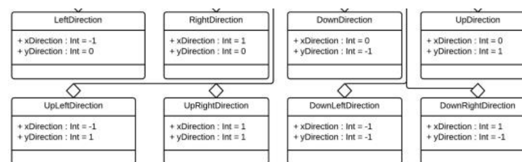

Figure 66: Case Classes MoveDirections Class Diagram

There are 8 states that the movement direction is allowed to be in. These case classes have predefined X and Y directions and override the asJson method to print out the correct state to the user.

These are 8 case classes are defined as:

- LeftDirection has an X direction of -1 and Y direction of 0
- RightDirection has an X direction of 1 and Y direction of 0
- UpDirection has an X direction of 0 and Y direction of 1
- DownDirection has an X direction of 0 and Y direction of -1
- UpLeftDirection has an X direction of -1 and Y direction of 1
- UpRightDirection has an X direction of 1 and Y direction of 1
- DownLeftDirection has an X direction of -1 and Y direction of -1
- DownRightDirection has an X direction of 1 and Y direction of -1

3.7 Collision Objects

3.7.1 QuadTree

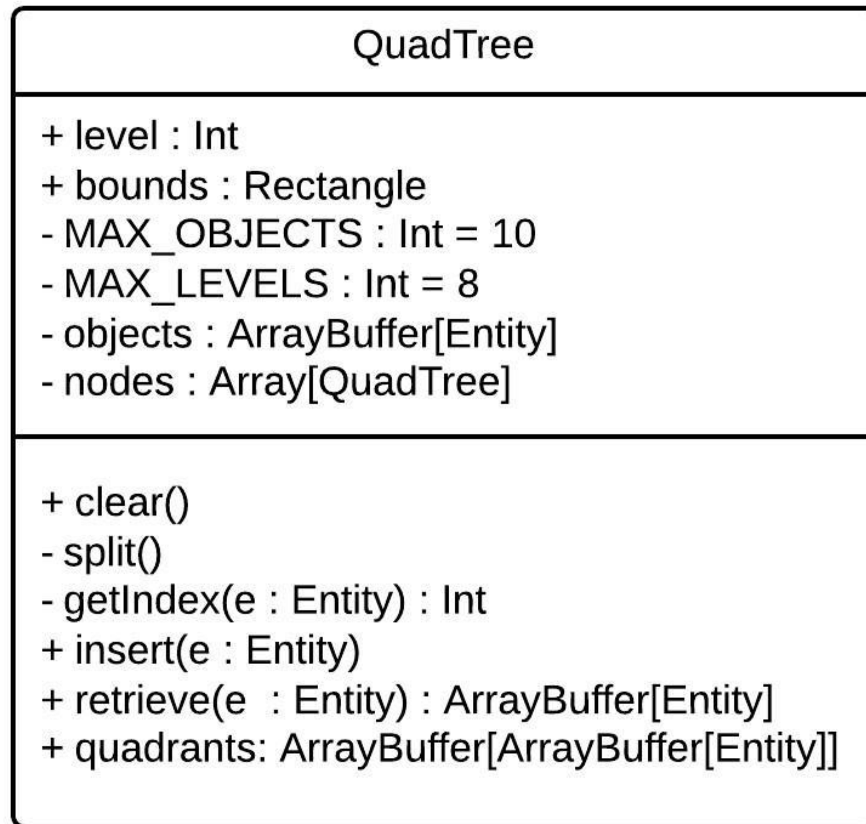

Figure 67: QuadTree Class Diagram

Quadtrees are tree data structure that is used to find the entities that are most likely to collide with each other. The QuadTree splits itself into four tree nodes which themselves have four nodes. Once a certain amount of items have been put into a node, it further splits itself up and divides those entities up. It allows for users to detect entities without checking against each one and run a collision detection algorithm on a smaller set of entities.

Name	Type	Description
level	Int	Depth of tree node
bounds	Rectangle	What bounds does this node take care of

MAXOBJECTS	Int	The max number of objects in a quadtrees
MAXLEVELS	Int	The maxdepth of the quadtree
objects	ArrayBuffer[Entity]	The list of objects the list hold
nodes	ArrayBuffer[QuadTree]	The nodes of a quadtree

Operations

Operation: clear()
Input : None
Output : None
Description : Clear all nodes below

Operation: split()
Input : None
Output : None
Description : Creates four nodes on current quadtree

Operation: getIndex(e : Entity) : Int
Input : e : Entity - entity to find index for
Output : Quadrant entity is in
Description : Finds the entity in the quadtree and returns quadrant

Operation: insert(e : Entity)
Input : e : Entity - entity to insert Output :
None
Description : Inserts entity into quadtree

Operation: retrieve(e : Entity) : ArrayBuffer[Entity]
Input : e : Entity - the entity to check against
Output : ArrayBuffer of entities
Description : Using Entity, retrieves all entities in given entity quadrant

3.7.2 Rectangle

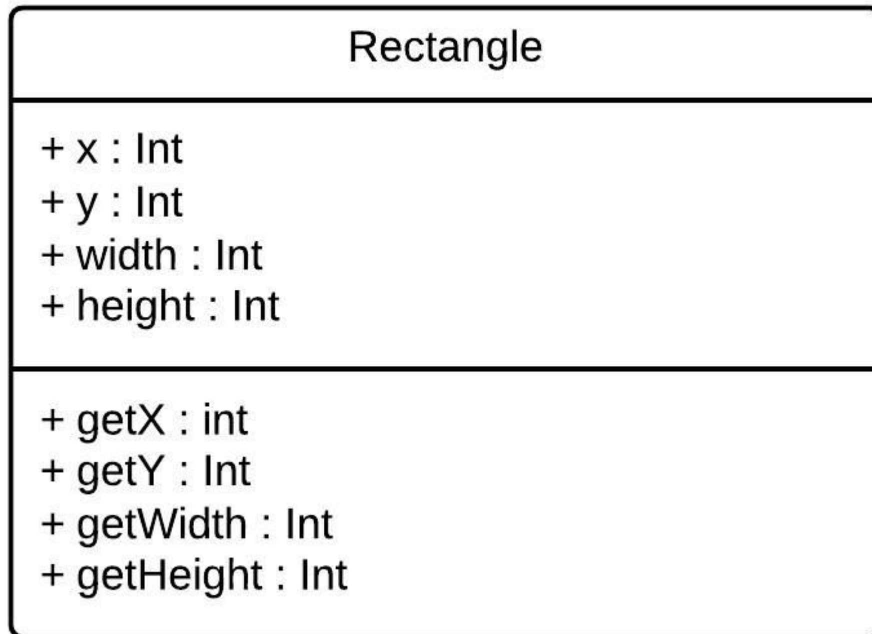

Figure 68: Rectangle Class Diagram

Name	Type	Description
x	Int	Top left corner location of rectangle (x-axis)
y	Int	Top left corner location of rectangle (y-axis)
width	Int	Width of Rectangle
height	Int	Height of Rectangle

Name **Type** **Description**

3.8 Factories

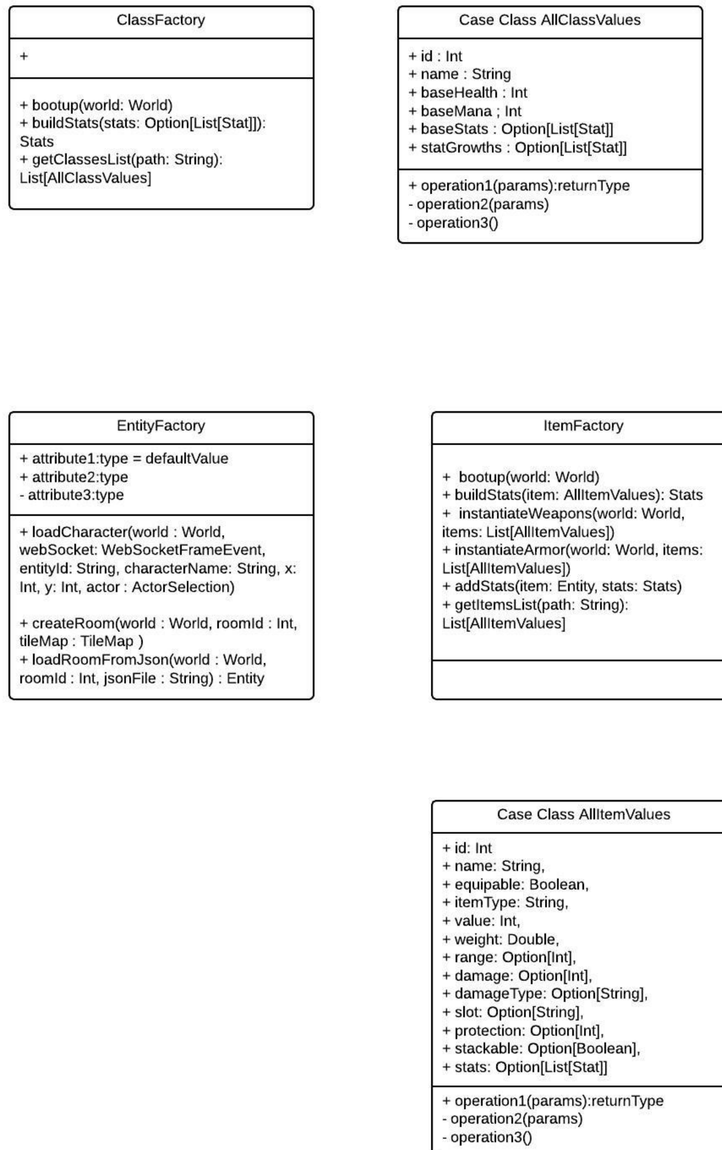

Figure 69: Factories

There are three factories that Ayai uses. The ItemFactory, ClassFactory, and EntityFactory and all are needed to fill in the information needed for the specific type.

3.8.1 ClassFactory

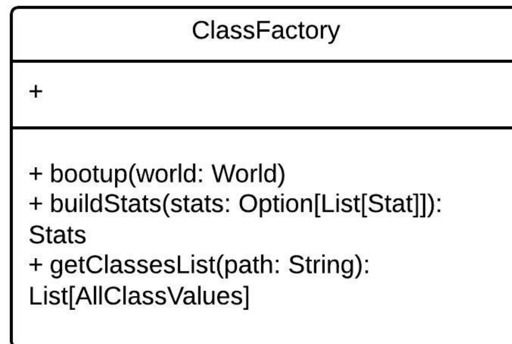

Figure 70: ClassFactory Class Diagram

ClassFactory is used on bootup to create all initial classes in the game and store them in memory.

Operations

Operation: bootup(world : World)

Input : world : World - the world to store classes

Output : None

Description : Read all necessary input files and create Classes

Operation: buildStats(stats : Option[List(Stats)]) : Stats

Input : stats : Option[List(Stats)] - an option for returned stats

Output : Returns stats created

Description : Takes in an Option for Stats and returns the potential stats class

Operation: getClassesList(path : String) : List[AllClassValues]

Input : path : String - path to a classes file

Output : Returns a list of classes retrieved from file

Description : Takes in a path file and return all classes read in

3.8.2 ItemFactory

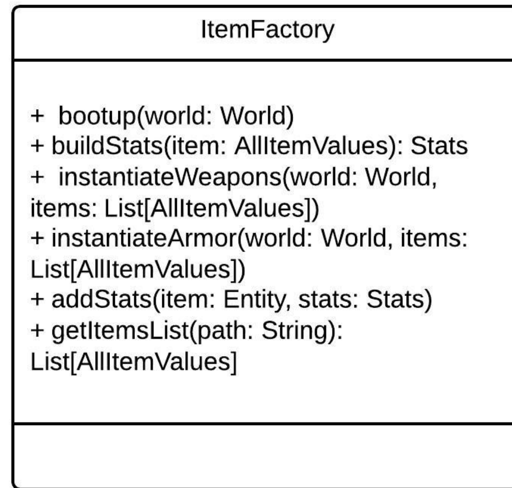

Figure 71: ItemFactory Class Diagram

ItemFactory is used on bootup to create all initial items in the game and store them in memory.

Operations

Operation: bootup(world : World)

Input : world : World - the world to store items

Output : None

Description : Read all necessary input files and create items

Operation: buildStats(item : AllItemValues) : Stats

Input : stats : AllItemValues - a case class with info of item

Output : Returns stats created

Description : Takes in a AllItemValues for Stats and returns the stats class

Operation: getItemsList(path : String) : List[AllItemValues]

Input : path : String - path to a classes file

Output : Returns a list of classes retrieved from file

Description : Takes in a path file for items and returns all items read in

Operation: addStats(item : Entity, stats : Stats)

Input : item : Entity - Entity to add stats to

stats : Stats - Stats to add to Entity

Output : None

Description : Adds given stats file to item entity

Operation: instantiateWeapons(world: World, items: List[AllItemValues])

Input : world : World - world to add item entities to
items : List[AllItemValues] - List of weapons items
Output : None
Description : Adds weapons to world

3.8.3 QuestFactory

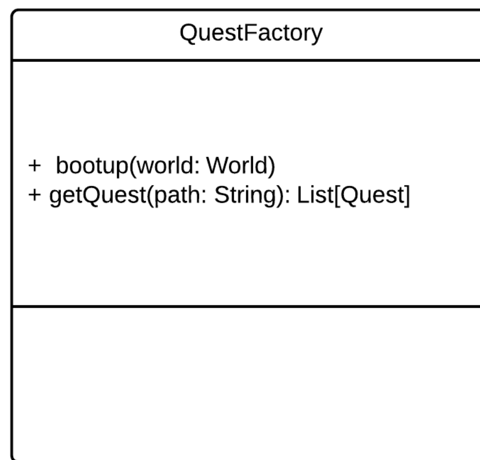

Figure 72: QuestFactory Class Diagram

QuestFactory is used on bootup to create all initial quests in the game and store them in memory.

Operations

Operation: bootup(world : World)

Input : world : World - the world to store quests
Output : None
Description : Read all necessary input files and create quests

Operation: getQuest(path: String)

Input : path : String - file of stored quests
Output : Returns a List[Quest] of quests loaded in
Description : Adds quests to game

3.8.4 GraphFactory

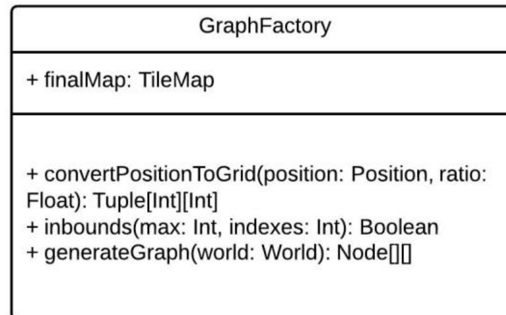

Figure 73: GraphFactory Class Diagram

GraphFactory is used with the AI components to see if a AI's position is in the map and to generate a graph of allowable positions

<u>**Operations**</u>

Operation: convertPositionToGrid(position: Position, ratio:Float)

Input : position: Position - the position to use
ratio: Float - Ratio to divide position by Output :
Returns graph of allowable positions
Description : Convert the position to a grid

Operation: generateGraph(world: World)

Input : world: World - world to collect tilemap from
Output : Returns a 2D array of nodes
Description : Generates a graph of nodes

Operation: inbounds(max: Int, indexes: Int*)

Input : max: Int - max length to check against
indexes: Int* - list of indexes to check inbounds Output : Returns boolean if one does not match
Description : Checks if list of indexes is in range

3.8.5 EntityFactory

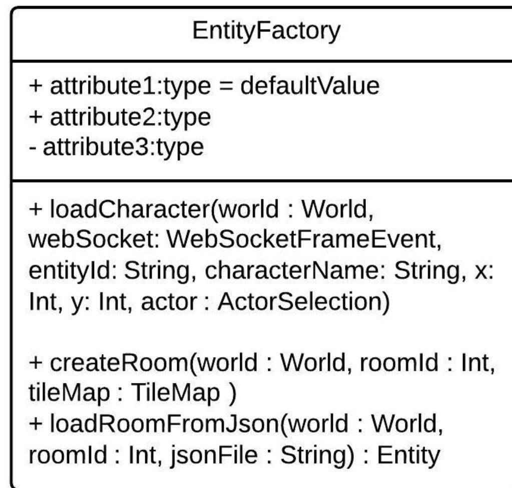

Figure 74: EntityFactory Class Diagram

EntityFactory is used to create the initial room's files and import them into the world and also used to create all character entities.

Operations

Operation: loadCharacter(world: World,webSocket: WebSocketFrameEvent, entityId: String, characterName: String, x: Int, y: Int, actor : ActorSelection)
Input : world : World - the world to add player entity
entityId: String - the database id for the player
characterName: String - the players name x: Int - the x
coordinate of player y: Int - the y coordinate of player
actor : ActorSelection - the Connection to the player computer
Output : None
Description : Create character entity and create components based on given information

Operation: createRoom(world : World, roomId : Int, tileMap : TileMap)
Input : world : World - the world to add room too roomId :
Int - the Id to give to room
tileMap : TileMap - The tilemap component to add to the room
Output : None
Description : Creates room entity and gives entity roomId and tilemap component

Operation: loadRoomFromJson(world: World,roomId: Int,jsonFile: String) : Entity

Input : world : World - the world to add room too
roomId : Int - the Id to give to room jsonFile : String - file to read and create room with

Output : Returns created room Entity

Description : Takes in room JSON File and reads in values and creates Entity with it

3.9 Quest Generation *

3.9.1 Overview

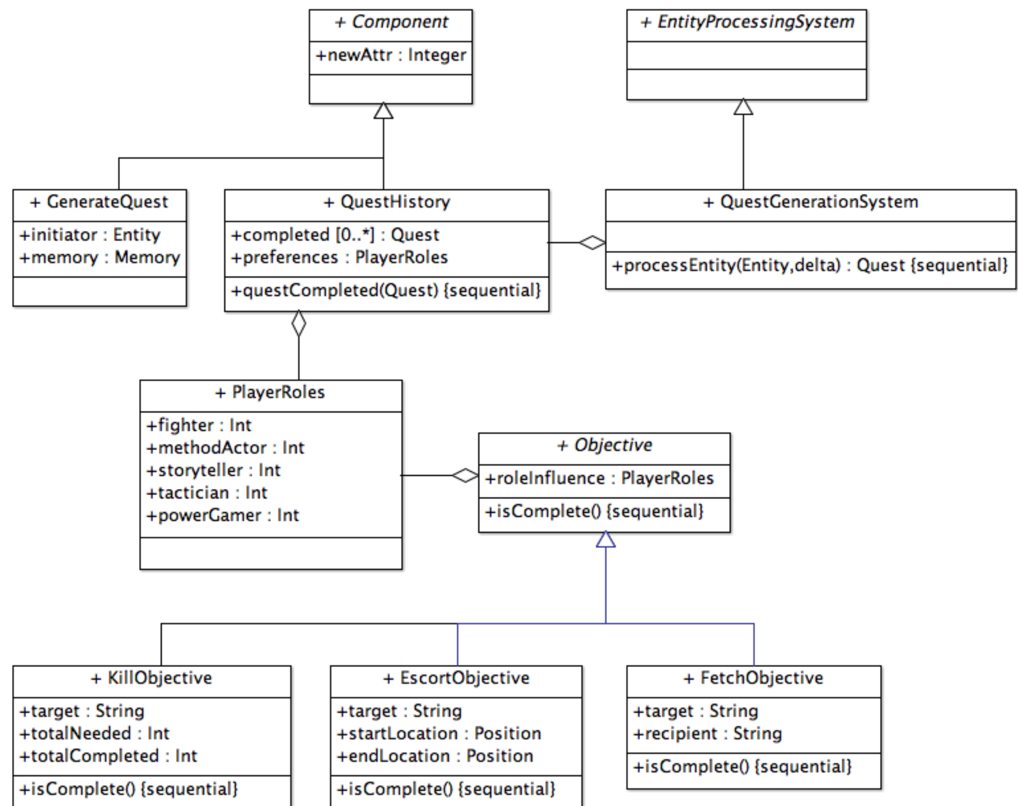

Figure 3.9.1 Quest Generation

The Quest Generation system is intended to instantiate Quest objects, and all their related components at any point during the main application game loop. This allows for the creation of quests with objectives directly corresponding to the world state, and content tailored to the gameplay preferences and playstyle of a given player.

3.9.2 Components

A number of components are defined to aid in the passage of data back and forth between the QuestGenerationSystem, and the individual entities corresponding to objects in the game world.

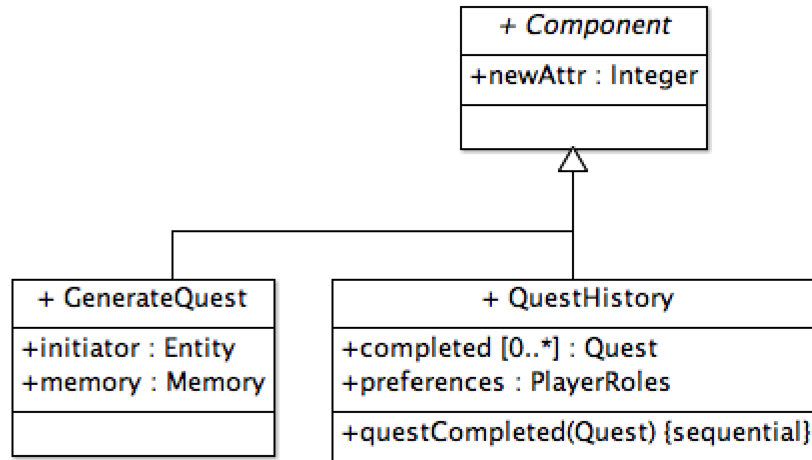

Figure 3.92 Components of Quest Generation

3.9.2.1 GenerateQuest

GenerateQuest is a component used for passing data back and forth between a given entity and the QuestGenerationSystem instance in the world.

Attributes

Name	**Type**	**Description**
initiator	Entity	A reference to the entity which initiated the request to the quest generation system.
memory	Memory	A reference to the Memory component used in the perception system.

3.9.2.2 QuestHistory

QuestHistory is a component which contains a record of all quests an entity has completed, as well as a continually updated model of player preferences.

Attributes

Name	**Type**	**Description**
completed	Quest []	A list of all quest objects which have been marked “completed” for this entity.
preferences	PlayerRoles	An object containing the current estimate of the gameplay preferences of a player.

Operations

Operation: QuestCompleted(Quest: quest)
Input: The quest which should be added to the completed quest list.
Output: None
Description: Adds a quest passed as a parameter to the list of completed quests, stored within the QuestHistory component, updating the preferences field.

3.9.3 Systems

All manipulation of data is handled by a single EntityProcessingSystem, which is responsible for the majority of actions in the quest generation framework.

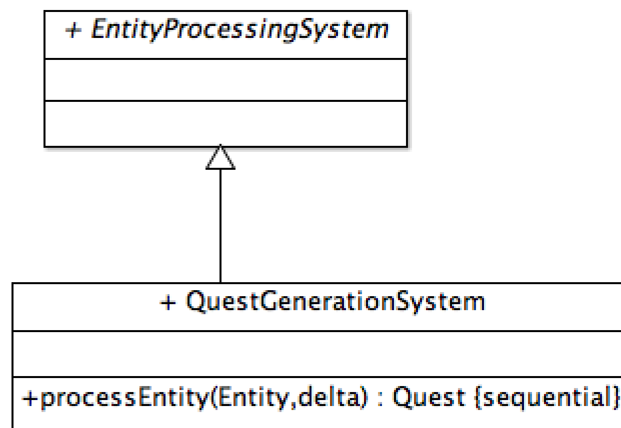

Figure 3.9.3 QuestGenerationSystem

3.9.3.1 QuestGenerationSystem

This system extends the base EntityProcessingSystem, and is designed to operate on entities which contain the GenerateQuest component, generating a new quest, returning it to the original initiator. Quests instantiated by this system will be constructed based on input from other components, such as the QuestHistory, contained within the initiating entity.

Operations

Operation: processEntity (E: Entity, delta: Integer)
Input: the entity to be processed, the time difference from the last frame.
Output: Quest
Description: Processes information passed by entity, and passes a new quest object back to the initiating entity.

3.9.4 Architecture

3.9.4.1 Impact on existing architecture

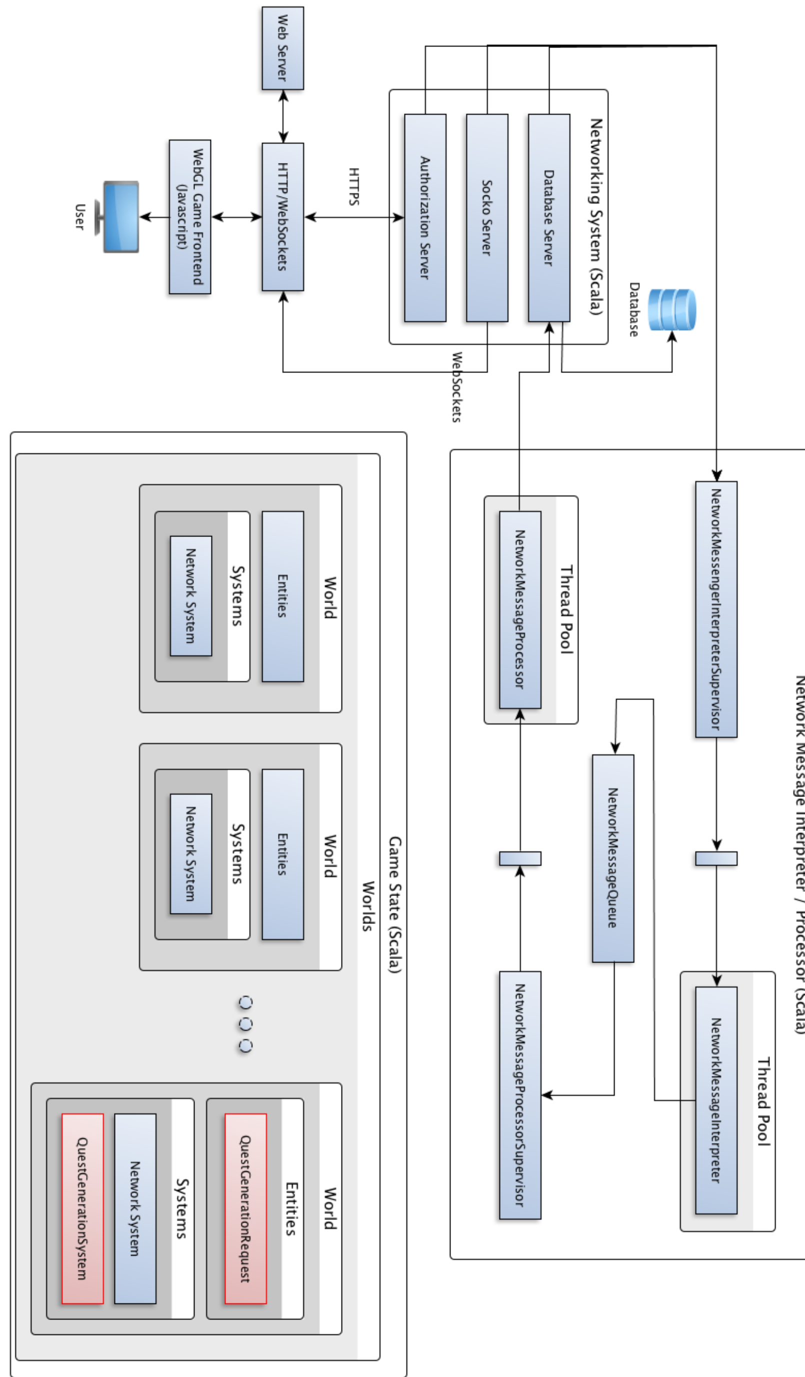

Figure 3.9.4.1 A Impact

The above diagram shows where in the greater system the Quest Generation System lives.

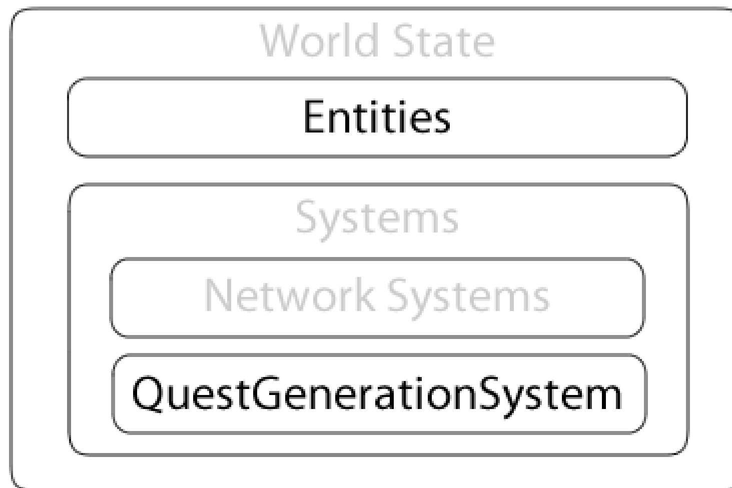

Figure 3.9.4.1 B Impact

The quest generation framework integrates into the existing project at the ECS level, and does not interact with the underlying architecture. Required components are attached to existing entities in the game world. The quest generation system is registered with the existing Entity Processing System framework.

3.9.4.2 System Sequence Diagram

The Quest Generation process is managed primarily by three objects. The NPC entity, the Player entity, and the Quest Generation System. All character entities in the game world have an attached QuestBag component. When a player interacts with an NPC, the first quest in their QuestBag component is presented, and the player is allowed to accept or reject the proposal. Should they accept, the quest is added to the player's QuestBag.

When a QuestBag component presents the last quest in the queue, another must be generated to take its place. Here, a "GenerateQuest" component is instantiated, and added to the NPC entity object. This acts as a way to transmit relevant data to the QuestGenerationSystem, which employs the EntityProcessingSystem interface to locate and process all GenerateQuest components. The EntityProcessingSystem uses information about the source, and destination entities to instantiate and configure a new Quest object adding Objectives to the Quest's internal objective list to match the preferences of the player stored in the QuestHistory component, before appending it to the source QuestBag. Once this process is complete, the newly generated quest may be presented to the player.

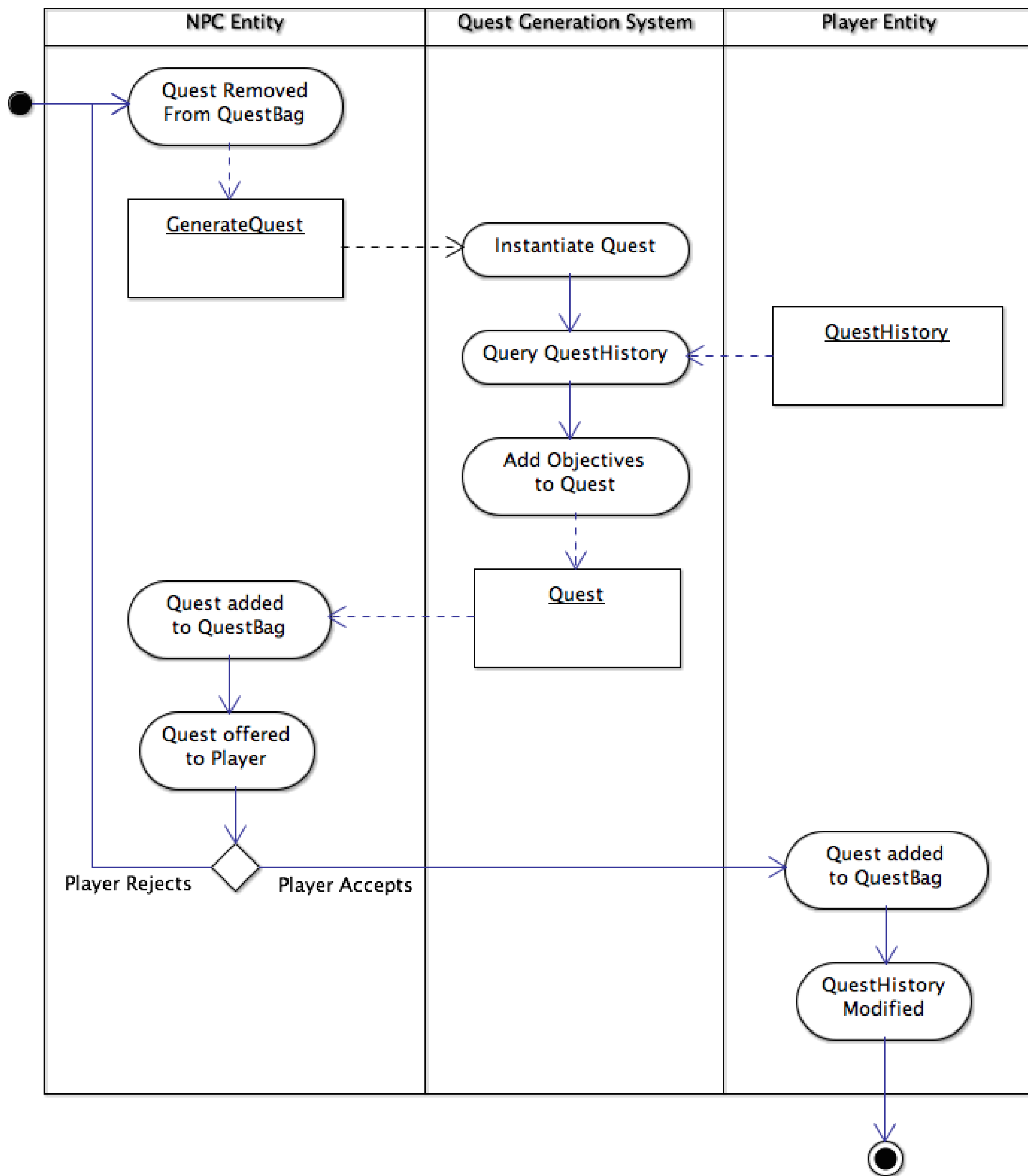

Figure 3.9.4.2 Quest Generation Sequence Diagram

3.9.5 Algorithms

3.9.5.1 PaSSAGE

http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-041.pdf

A simplified method of representing player preferences within a game world by a series of roles. The actions of players modify the influence of each role on story and content generation. This model will be utilized in quest generation to steer the scope, and type of content being presented to players.

3.10 Perception *

3.10.1 Components

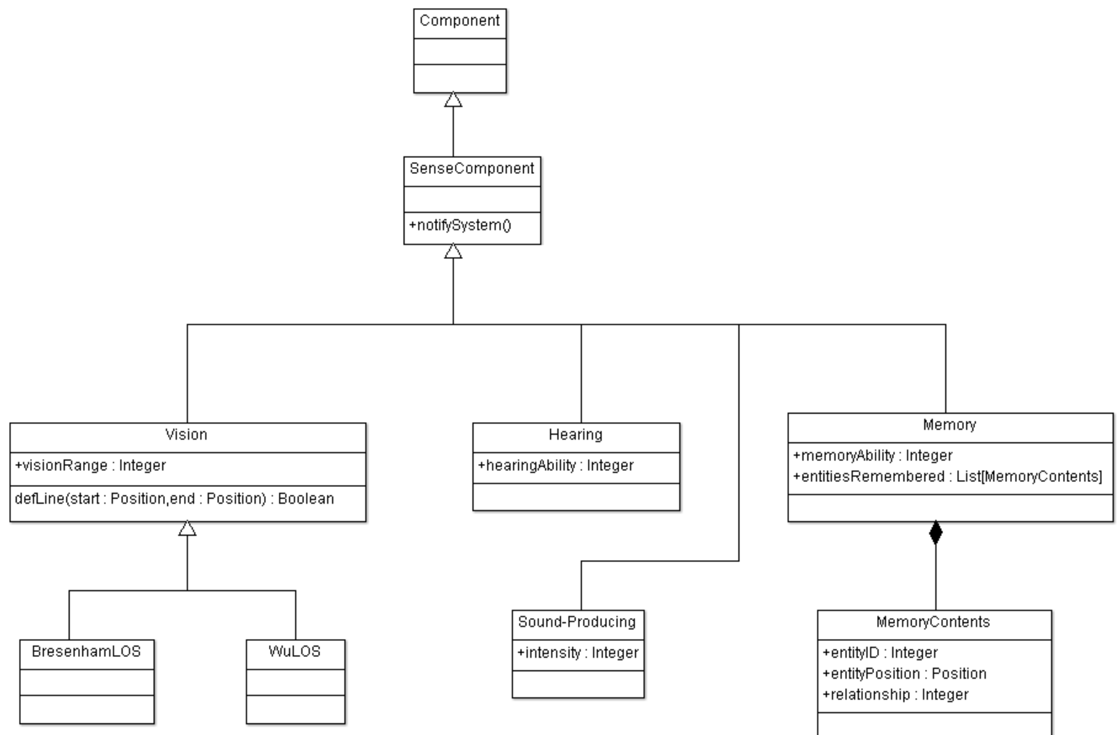

Figure 3.10.1 Components

3.10.1.1 Sense

Sense Component is an abstract component that is attached to an entity to indicate the attached entity can perceive through

Operation: notifySystem()
Input: None
Output: None
Description: Notifies relevant systems of state changes. Overwritten to link to relevant system.

3.10.1.2 Vision

Vision is a component that would indicate the attached entity is capable of seeing things.

Attributes

Name	Type	Description
visionRange	Integer	A rating of how far the attached entity can see. Used in calculating Line of Sight.

Operations

Operation: defLine(start: Position, end: Position): Boolean
Input: Starting point and ending point of the line to be drawn.
Output: Boolean indicating whether there unobstructed is line of sight between two entities.
Description: Draws a line between two positions and determines whether there is unobstructed line of sight along the line. BresenhamLOS and WuLOS will provide implementations that use their respective algorithms.

Algorithms

- Bresenham's line algorithm
- Wu's line algorithm

3.10.1.3 Hearing

Hearing is a component that indicates the attached entity can listen to sounds.

Attributes

Name	Type	Description
hearingAbility	Integer	A rating of how well the attached entity can listen to sounds.

3.10.1.4 Sound-Producing

Attributes

Name	**Type**	**Description**
intensity	Integer	A rating of how loud the associated sound entities will be when created..

3.10.1.5 Memory

Memory is a component that indicates the attached entity can remember things. The Memory system is also used in Quest Generation.

Attributes

Name	**Type**	**Description**
memoryAbility	Integer	A rating of how well the attached entity can remember things.
entitiesRemembered	List[MemoryContents]	A collection of the things the attached entity remembers.

3.10.1.6 Memory Contents

Memory Contents is a data structure used by the Memory sense to show what information is remembered.

Attributes

Name	**Type**	**Description**
entityID	Integer	The ID of the entity
entityPosition	Position	The last remembered postion of the entity.
relationship	Integer	A score showing the indicated entities' opinion of the attached entity. A higher score means a more favorable opinion.

3.10.2 Entities

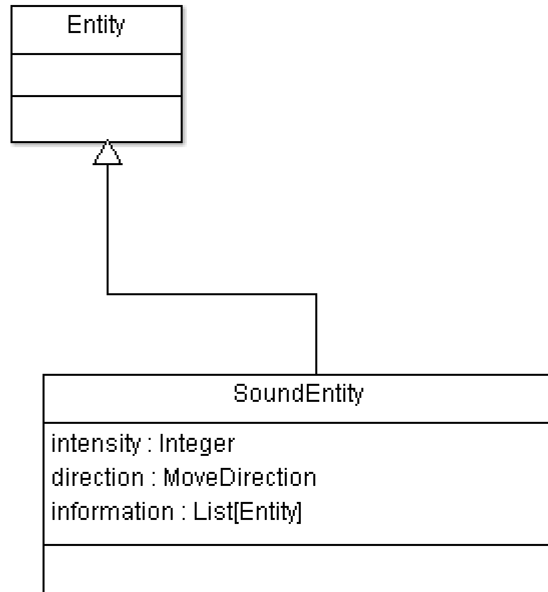

Figure 3.10.2 Entities

3.10.2.1 SoundEntity

An entity that contains aspects similar to a real world sound wave. These include direction, intensity, and data or information.

Attributes

Name	**Type**	**Description**
intensity	Integer	A rating corresponding to the loudness of the sound
direction	MoveDirection	The direction the sound is traveling.
information	List[Entity]	The entity or entities that the sound contains. Can be empty.

3.10.3 Systems

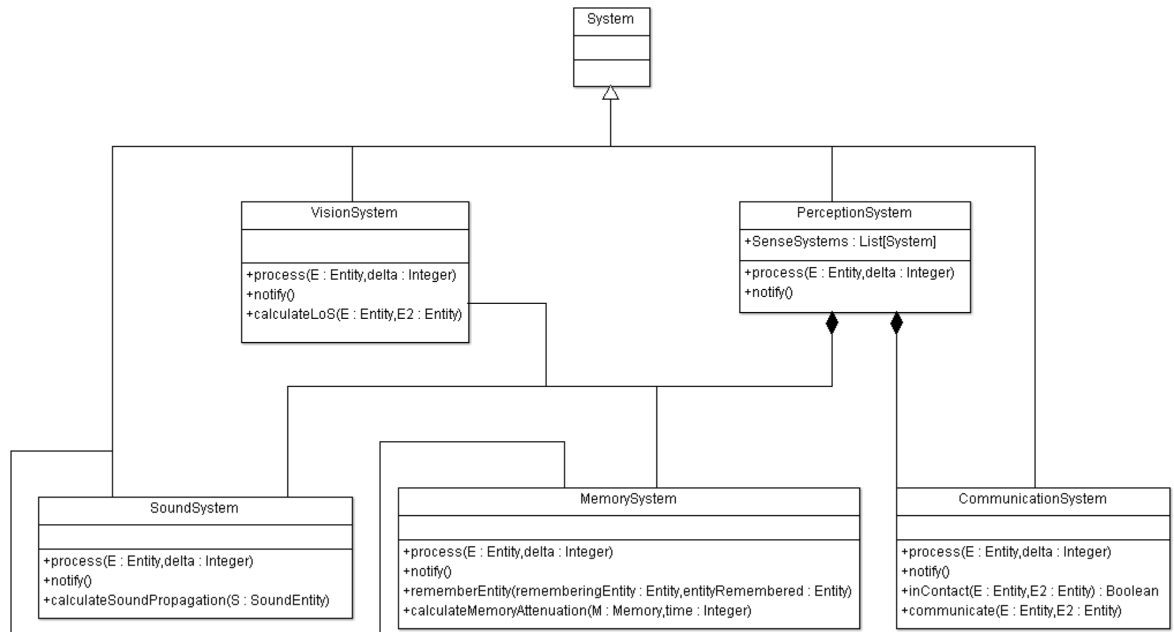

Figure 3.10.3 Systems

3.10.3.1 Primary System: Perception System

This system serves as the main thoroughfare for all senses defined at a given point. It allows for entities to be assigned an unlimited amount of defined senses and will control their various systems and subsystems.

Attributes

Name	Type	Description
SenseSystems	List[System]	A collection of systems relevant to defined senses.

Operations

Operation: process(E: Entity, delta: Integer)
Input: the entity to be processed, the time difference from the last frame.
Output: None
Description: Process the Entity, pass to relevant subsystems.

Operation notify()
Input: None
Output: None
Description: Handle any state changes in the observed entity / component.

3.10.3.2 Seconary/Included System and Subsystems

3.10.3.2.1 Vision System

This system controls an entities ability to use a sense of sight. Entities will have set limitations on aspects such as how far things can be seen, and algorithms determining line of sight calculations

<u>Operations</u>

Operation: process(E: Entity, delta: Integer)
Input: the entity to be processed, the time difference from the last frame.
Output: None
Description: Process the entity's Line of Sight.

Operation notify()
Input: None
Output: None
Description: Handle any state changes in the observed entity / component.

3.10.3.2.2 Hearing System

This system controls an entities ability to use a sense of hearing. Entities are able to determine whether there is a sound entity within range and information about that sound entity (defined above). Links to the pathfinding system to determine sound propagation.

<u>Operations</u>

Operation: process(E: Entity, delta: Integer)
Input: the entity to be processed, the time difference from the last frame.
Output: None
Description: Process the entity's Line of Sight.

Operation notify()
Input: None
Output: None
Description: Handle any state changes in the observed entity / component.

Operation: calculateSoundPropagation(S: SoundEntityr)
Input: the SoundEntity to be propagated
Output: None
Description: Spread the sound entity over an area, affected by the sound's intensity and direction.

3.10.3.2.3 Memory System

The memory system controls an entity's ability to remember things, adding remembered entities into their memory and calculating memory degradation over time.

<u>Operations</u>

Operation: process(E: Entity, delta: Integer)
Input: the entity to be processed, the time difference from the last frame.
Output: None
Description: Process the entity's memory

Operation notify()
Input: None
Output: None
Description: Handle any state changes in the observed entity / component.

Operation: rememberEntity(rememberingEntity: Entity, entityRemembered: Entity)
Input: the entity whose memory bank is being added to, the entity that is being remembered.
Output: None
Description: Process the Entity, pass to relevant subsystems.

Operation calculateMemoryAttenuation(M: Memory, time: Integer)
Input: the memory object that is being degraded, the amount of time that has passed.
Output: None
Description: Two entities who are able to communicate share perceived entities between each other

3.10.3.2.4 Communication System

The communication system provides the ability for two entities to share perceived entities between one another.

<u>Operations</u>

Operation: process(E: Entity, delta: Integer)
Input: the entity to be processed, the time difference from the last frame.
Output: None
Description: Process the entity

Operation notify()
Input: None
Output: None
Description: Handle any state changes in the observed entity / component.

Operation: inContact(E: Entity, E2: Entity): Boolean
Input: the two entities who are trying to communicate
Output: Boolean indicating whether these entities can or can not communicate
Description: Process the Entity, pass to relevant subsystems.

Operation communicate(E: Entity, E2: Entity)
Input: the two entities who are trying to communicate
Output: None
Description: Two entities who are able to communicate share perceived entities between each other

3.10.4 Architecture

The perception framework integrates into the existing project at the ECS level.

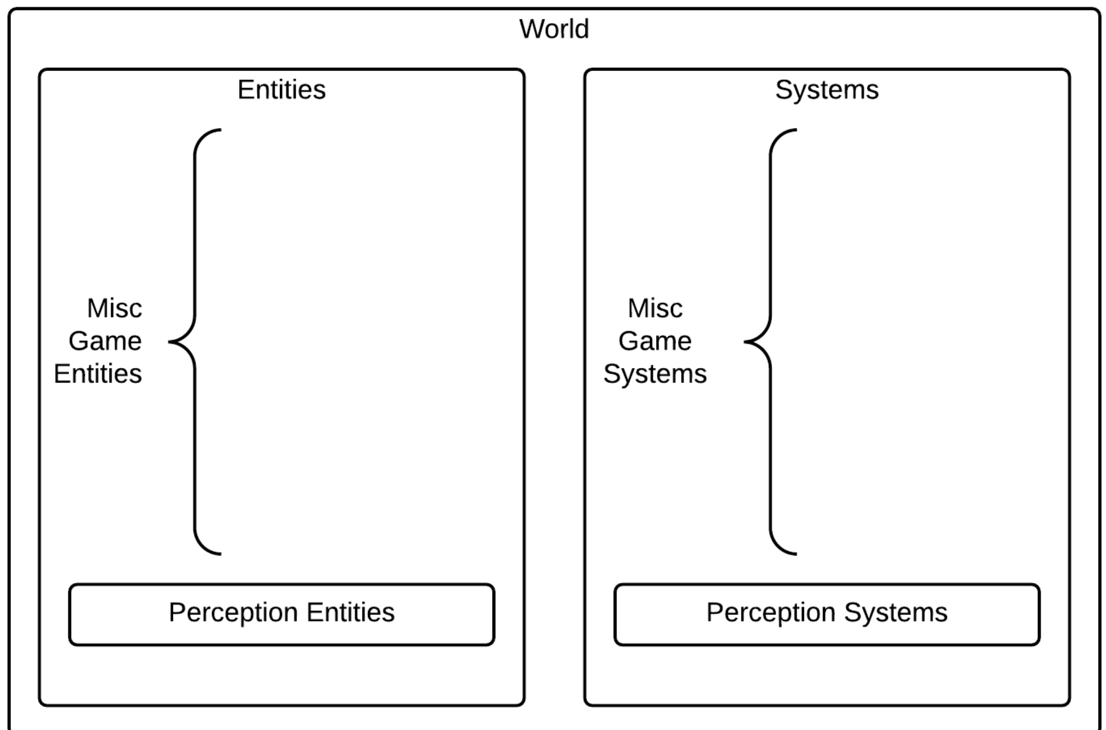

Figure 3.10.4 A Integration

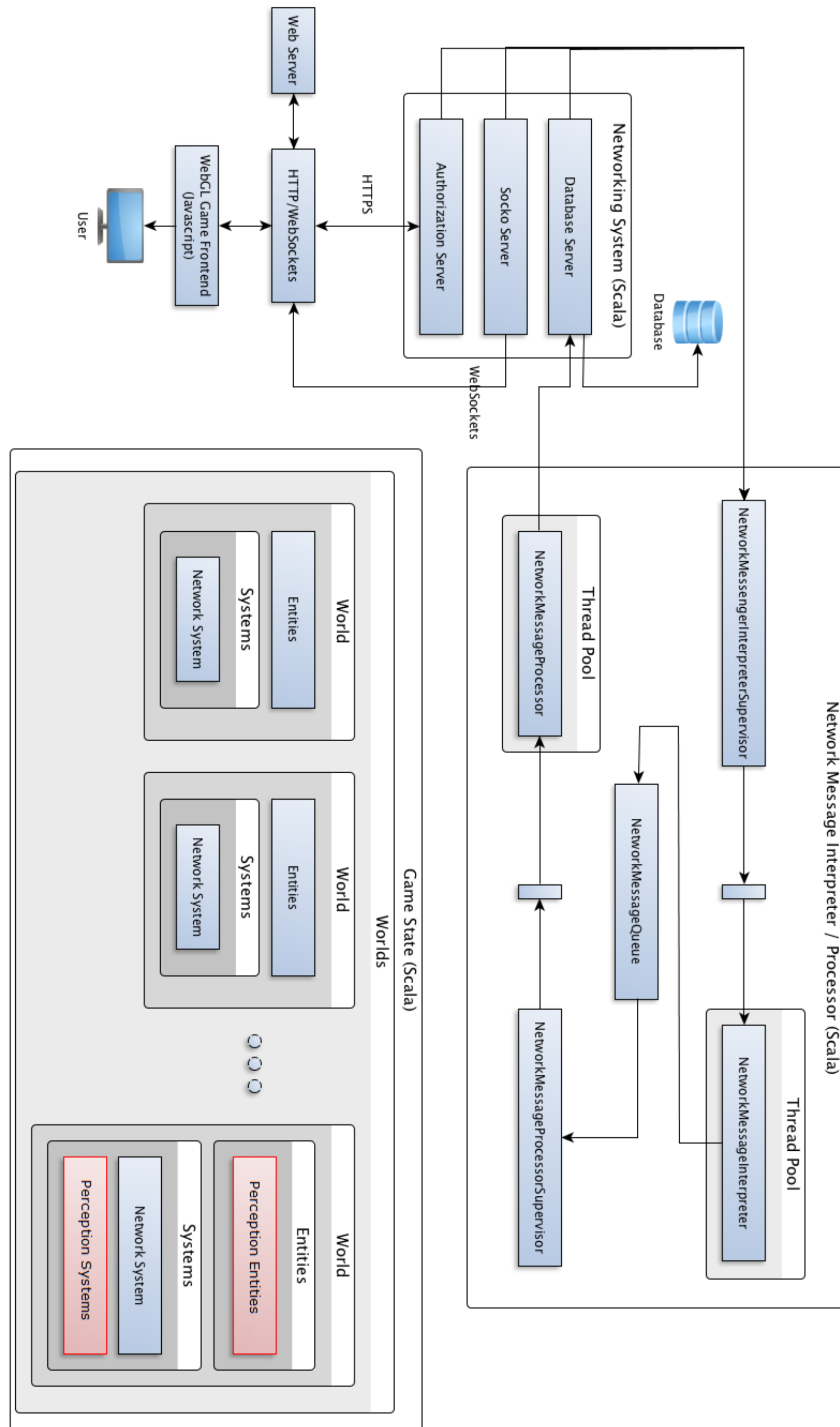

Figure3.10.4 B Impact

3.10.5 Process and Design Patterns

3.10.5.1 Sequence Diagram

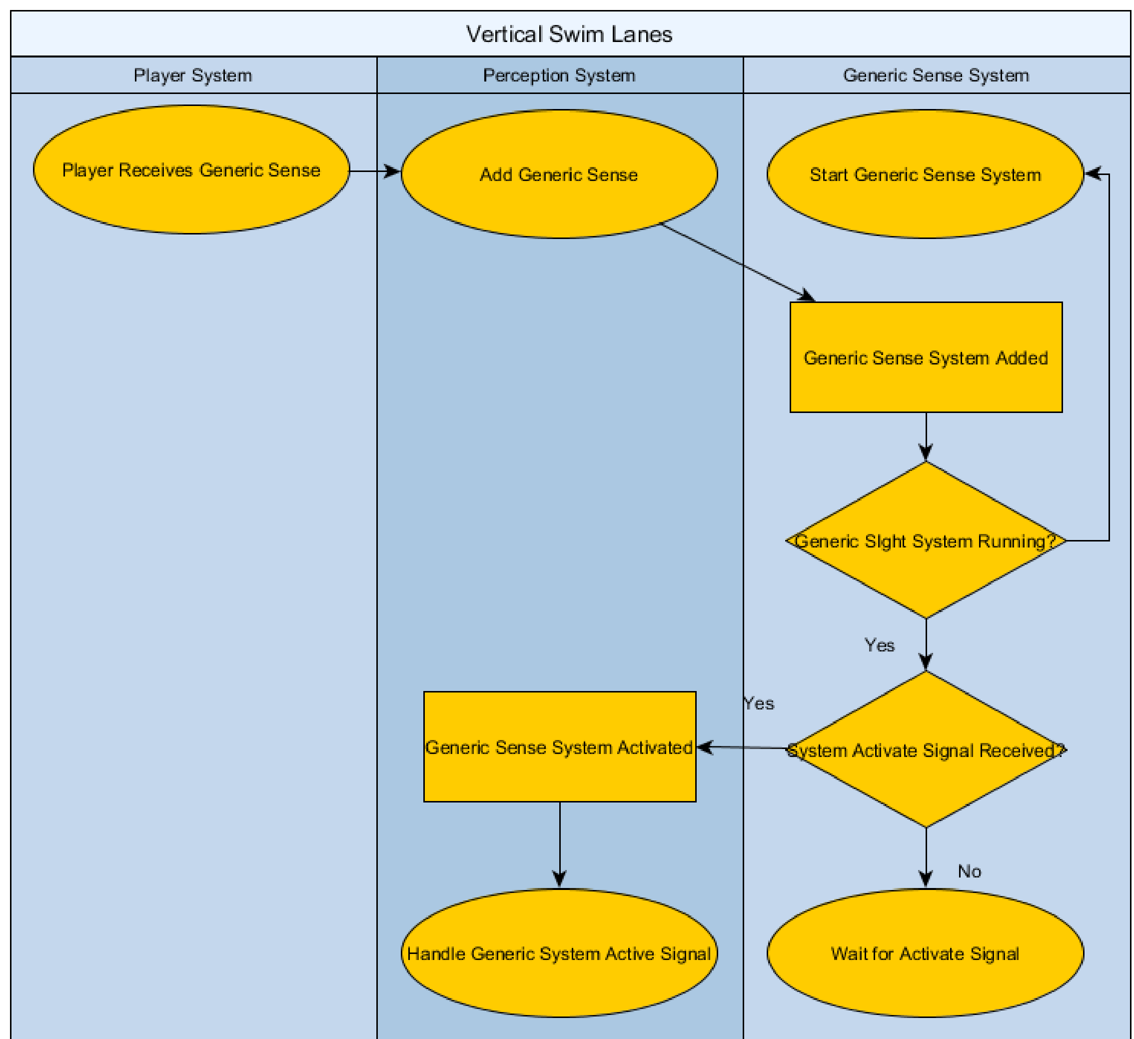

Figure 3.10.5.1 Sequence Diagram

3.10.5.2 Entity Component System

The perception framework integrates into the existing project through the Entity Component System. The perception framework uses mostly components to represent different senses and systems to process how entities use those sense.

3.10.5.3 Observer Pattern

The observer pattern is used to maintain communication between SenseComponents and the system which processes that Sense. This allows the perception framework as a whole to be more modular and extensible, rather than having a single centralized system which processes all of the senses.

3.10.5.4 Strategy Pattern

The strategy pattern is used to keep algorithm implementations separate and easily interchangeable. Rather than a Line of Sight algorithm being hard coded into the Vision System, the Vision Component can override the relevant method. The selection of algorithms would be decided by the Game Definition File.

3.10.6 Algorithms

3.10.6.1 Bresenham's line algorithm

http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

- Bresenham uses interger arithmetic to draw an approximation of the line.
- Bresenham works by dividing the change of the major axis by the change in the minor axis.
- The "error" is tracked, the separation between the real line and the approximated line.
- The error is steadily increased across the major axis by the change of the major axis by the change in the minor axis.
- If the "error" is more than .5, the rasterization of the minor axis is increased by one and the error is decreased by one.

3.10.6.2 Wu's line algorithm

http://en.wikipedia.org/wiki/Xiaolin_Wu%27s_line_algorithm

- Wu's algorithm uses anti-aliasing to draw a more precise line, but takes longer than Bresenham's algorithm.
- Wu's algorithm draws lines in a similar way to Bresenham, but draws multiple lines, with their closeness to the "true" line determining that line's gradient.

3.11 Pathfinding *

3.11.1 Components

3.11.1.1 Pathfinder

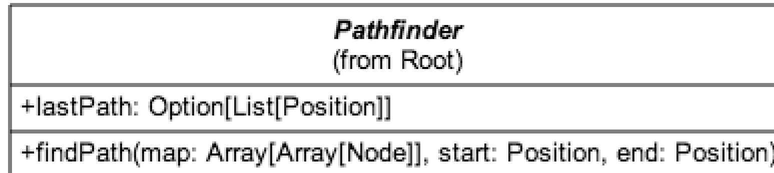

Figure 3.11.1.1 Pathfinder

Pathfinder is an abstract Component used to indicate that the attached entity utilizes pathfinding. Concrete implementations will inherit from this component and implement different pathfinding algorithms.

Attributes

Name	**Type**	**Description**
lastPath	Option[List[Position]]	Stores the last path generated for this entity for caching purposes. Is null if not path has been generated for this entity yet.

Operations

Operation: findPath(map: Array[Array[Node]], start: Position, end Postion)
Input: map: Array[Array[Node]], start: Position, end Postion
Output: returns Option[List[Position]]
Description: If defined, a list of positions the entity can traverse to arrive at its destination. If the list is empty, the entity has arrived at its destination. If null, there is no valid path from the entity to its destination position

3.11.1.2 AStar

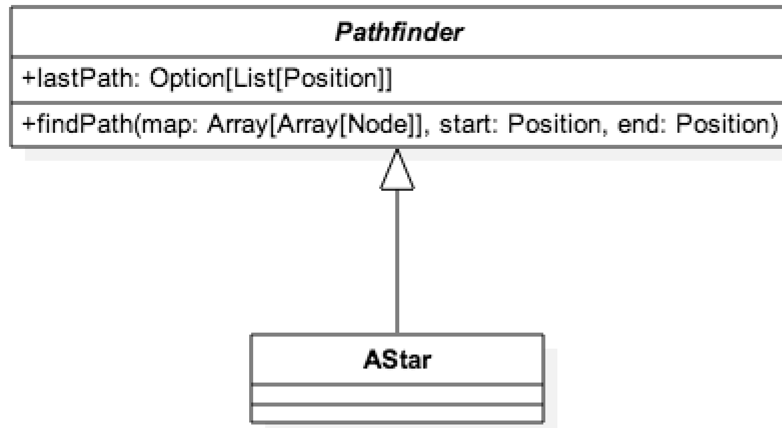

Figure 3.11.1.2 AStar

AStar is a concrete class inheriting from Pathfinder. AStar implements Pathfinder's abstract.

3.11.1.3 Dijkstra

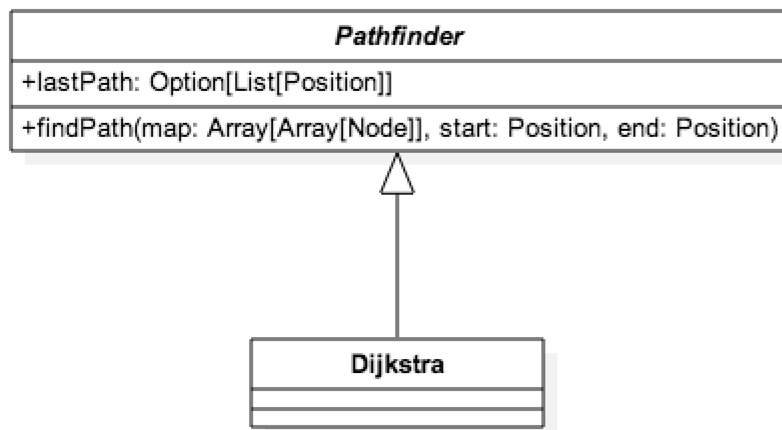

Figure 3.11.1.3 Dijkstra

Dijkstra is a concrete class inheriting from Pathfinder. Dijkstra implements Pathfinder's abstract method "findPath" with Dijkstra's algorithm to solve the single-source shortest-path problem.

3.11.1.4 DistanceHueristic

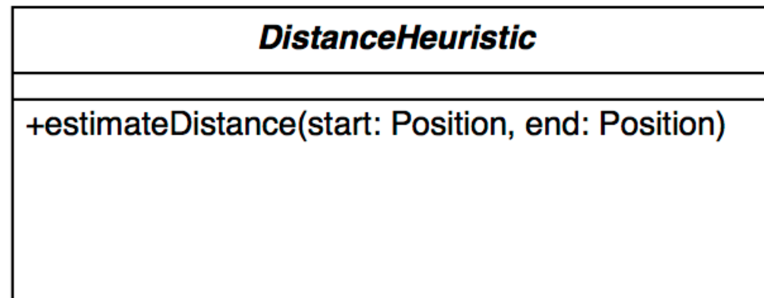

Figure 3.11.1.4 DistanceHeuristic

DistanceHeuristic is an interface which requires a single method, "estimateDistance" be implemented by inheritors. It is a dependency of the Pathfinder abstract component.

Operations

Operation: estimateDistance(start: Position, end: Position)
Input: start: Position, end: Position
Output: *Outputs:* Integer
Description: The distance, measured in Tiles, from *start* to *end*

3.11.1.5 ManhattanDistance

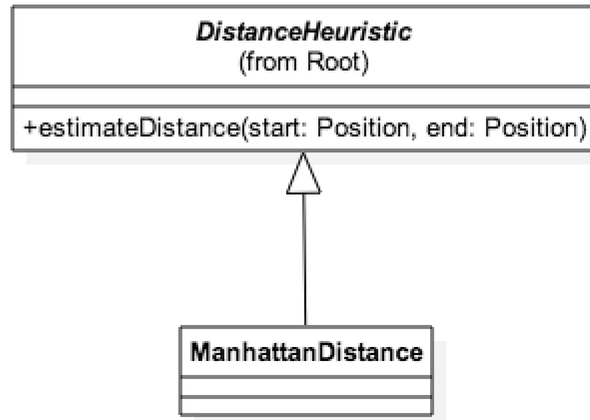

Figure 3.11.1.5 ManhattanDistance

ManhattanDistance is a realization of the DistanceHeurisitic interface. It implements the "estimateDistance" using the Manhattan Distance calculation. This algorithm is described in greater detail in the algorithms section.

3.11.1.6 DiagonalDistance

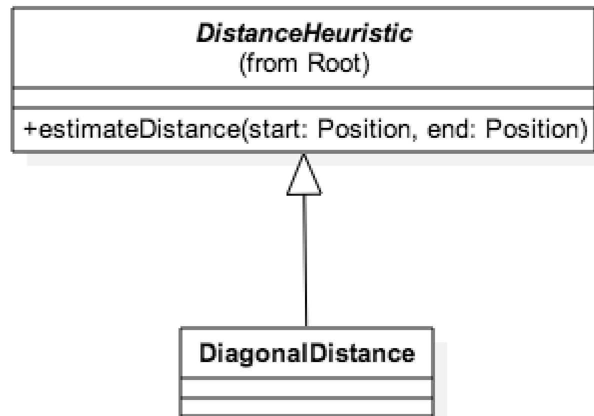

Figure 3.11.1.6 DiagonalDIstance

DiagonalDistance is a realization of the DistanceHeurisitic interface. It implements the "estimateDistance" using the Diagonal Distance calculation. This algorithm is described in greater detail in the algorithms section.

3.11.2 Systems

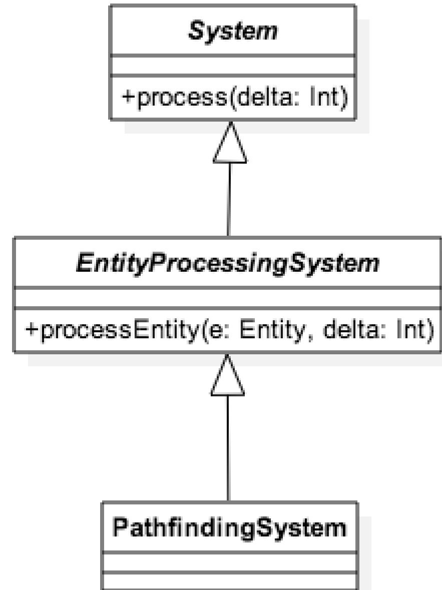

Figure 3.11.2 Systems

3.11.2.1 PathfindingSystem

PathfindingSystem is a concrete class inheriting from EntityProcessingSystem. The PathfindingSystem implements EntityProcessingSystem's "processEntity" function and selects all Entities in the current World which have the Pathfinder Component bound to them and runs the "findPath" method.

3.11.3 Design

3.11.3.1 Sequence Diagram

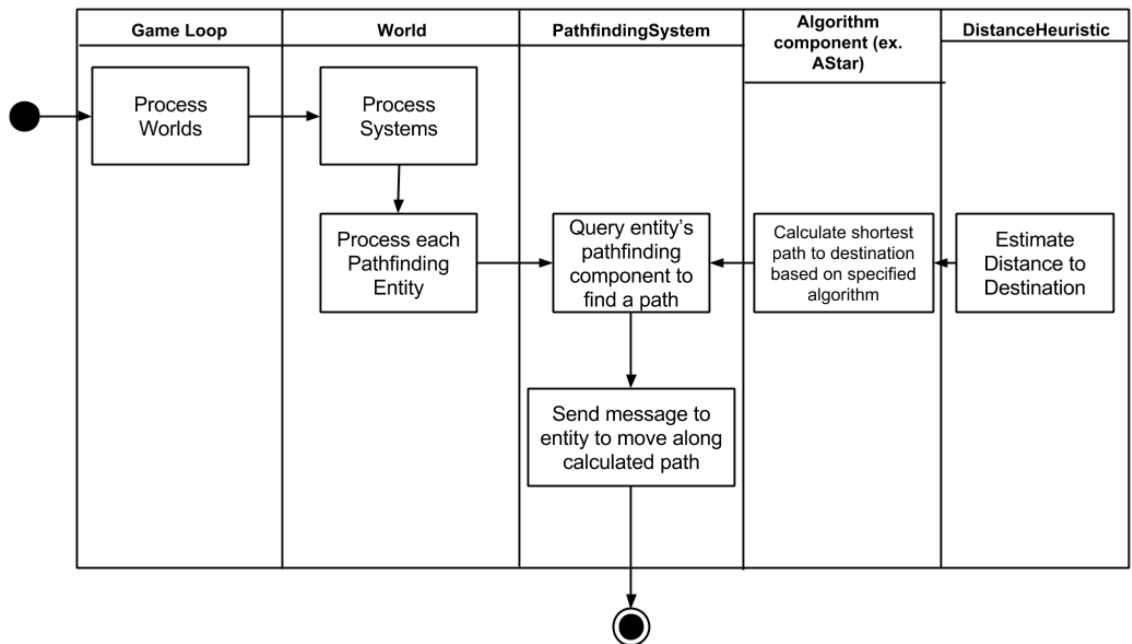

Figure 3.11.3.1 Sequence Diagram

Sequence Diagram showing the process of how paths get updated on each game tick.

3.11.3.2 Dependency Injection/Inversion of Control

The Pathfinder component uses the "Cake pattern", which is a Scala construct used to implement dependency injection. Scala traits, which are similar to Java interfaces, allow a self-type to be declared. This "self-type" is a type constraint on the Interface, mandating that it be mixed in with a class which conforms to its self-type. Pathfinder utilizes the "self-type" declaration but declaring a self-type of DistanceHeuristic. This means that whenever a class is created which conforms to Pathfinder, it must also conform to DistanceHeuristic. This method is used in favor of inheritance as it keeps the "is a" vs. "requires a" semantics correct at the code level.

3.11.3.3 Strategy Pattern

Using the "Cake Pattern" described above, we are also able to implement a version of the Strategy Pattern which enables the system to select an appropriate algorithm at

runtime. Due to the “stackable” nature of Scala traits and the Entity-Component system, different pathfinding and heuristic algorithms can be selected based on values of variables determined at runtime. This gives the programmer extreme flexibility in defining Entity behavior and can serve as the basis for more complex AI logic.

3.11.4 Algorithms/Data Structures

3.11.4.1 A* search algorithm [1]

The A* search algorithm is a type of best-first search. A* decides the order of nodes to explore by using past knowledge (the distance from the starting point of the search to the current point) in addition to using “future” knowledge, which is in the form of an admissible heuristic. A* is widely used in games as

3.11.4.2 Dijkstra’s algorithm [2]

Dijkstra’s algorithm, like A* search, is an algorithm for finding the shortest path between two nodes in a graph.

3.11.4.3 Manhattan Distance

Equation used to find the distance between two coordinates by calculating the summation of the absolute differences of their Cartesian coordinates [3].

$$\sum_{i}^{n} |p_i - q_i|$$

3.11.4.4 Diagonal (Chebyshev) Distance

Equation used to find the distance between two vectors along any coordinate dimension. On a 2-dimensional coordinate system, it is defined as:

$$max(dx,\ dy)$$

where

$$dx = |point1.x - point2.x|$$

and

$$dy = |point1.y - point2.y|$$

3.11.4.5 Binary Heap (java.util.TreeSet)

A binary heap is used to represent the queues (ex. open and closed sets in A* within the search algorithms. This data structure is used primarily for its O(logn) worst case/ O(1) average performance on insert and its O(logn) worst case performance for delete.

3.11.5 Architecture

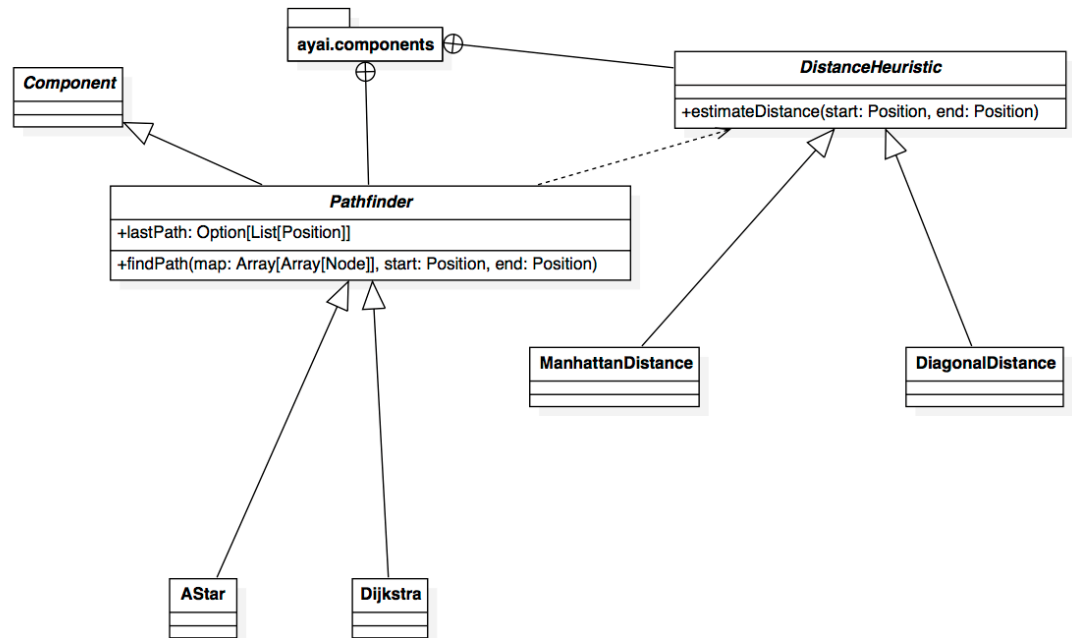

Figure 3.11.5 A Pathfinding

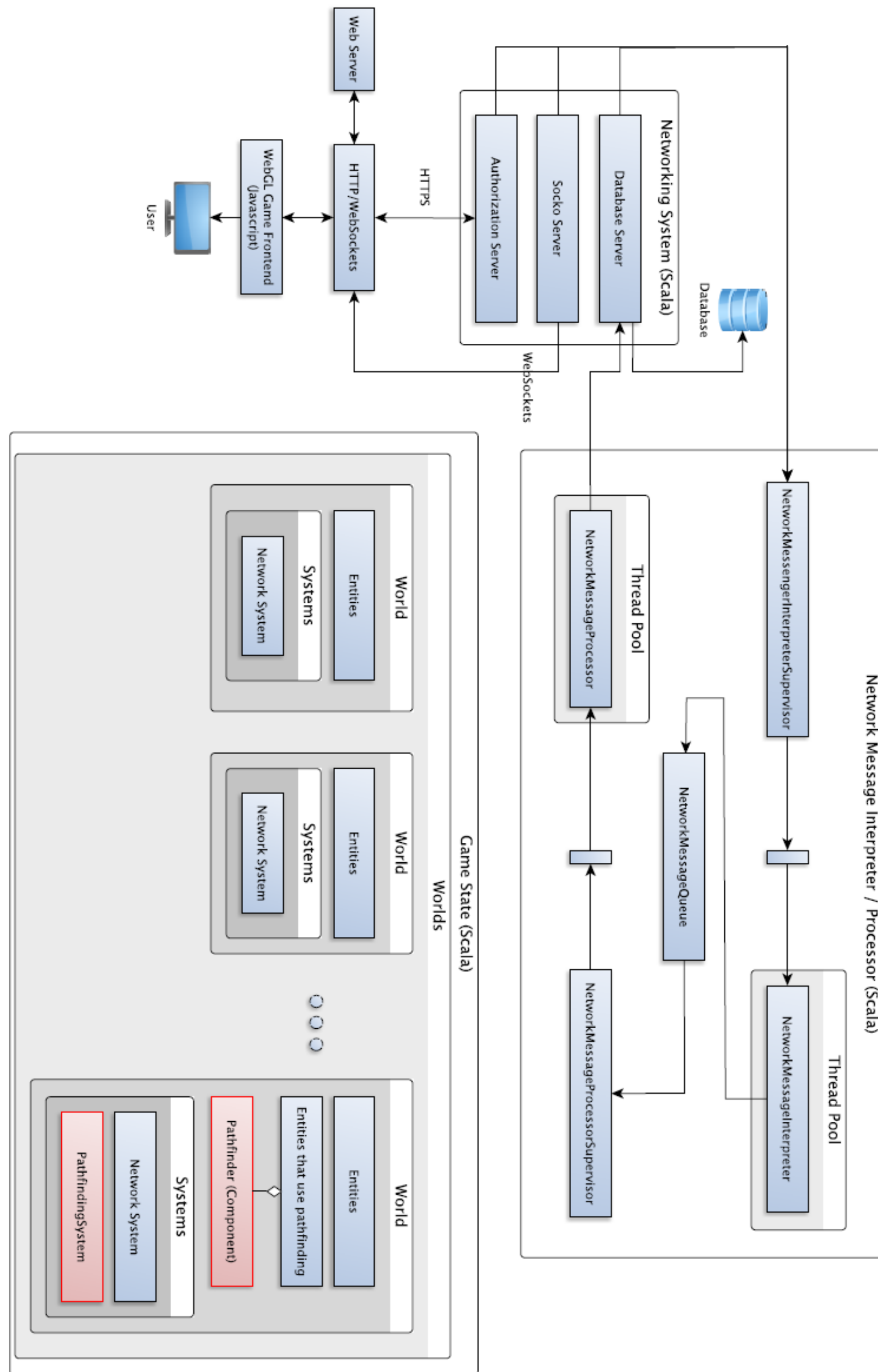

Figure 3.11.5 B Impact to Existing Architecture

3.11.6 References

[1] Hart, P., Nilsson, N., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 100-107.

[2] Dijkstra, E. (1959). A Note On Two Problems In Connexion With Graphs. Numerische Mathematik, 269-271.

[3] Eugene F. Krause (1987). Taxicab Geometry. Dover. ISBN 0-486-25202-7

3.12 Map Generation *

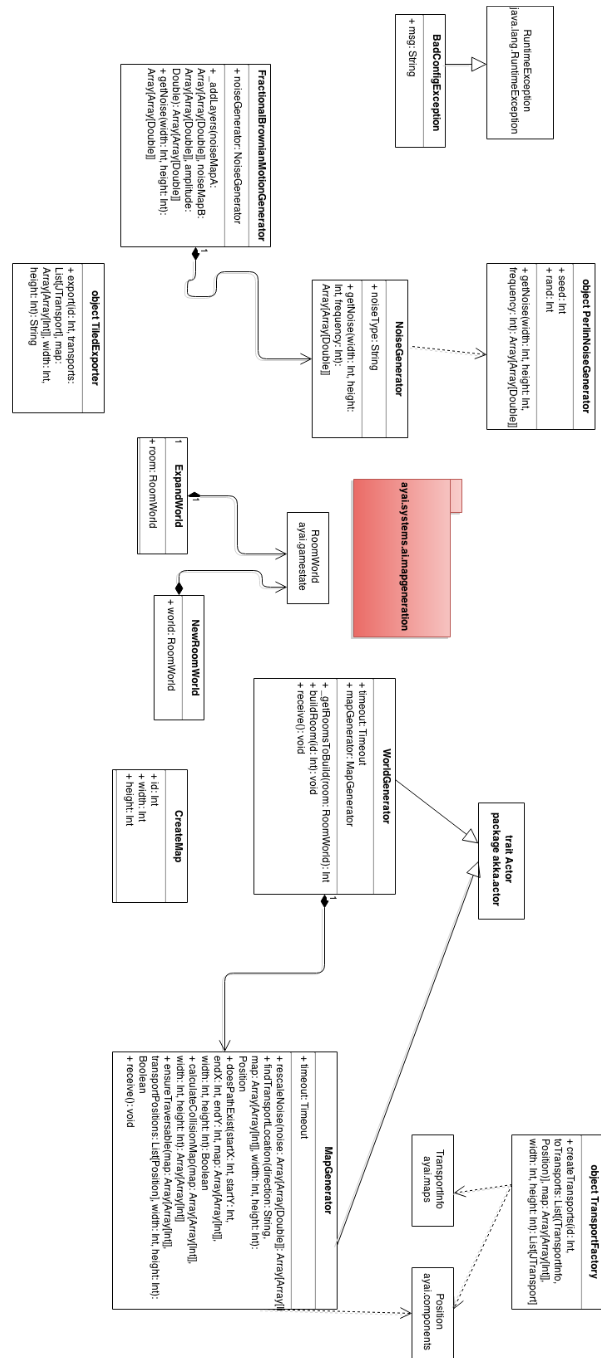

Figure 3.12 Map Generation

The Map Generation system is responsible for the construction and verification of maps in the Ayai game. Whenever a player logged on to the web server reaches the edge of a generated map, the Map Generation system constructs a new map, verifies that the map is traversable, and serves it up to any relevant players browsers.

3.12.1 WorldGenerator

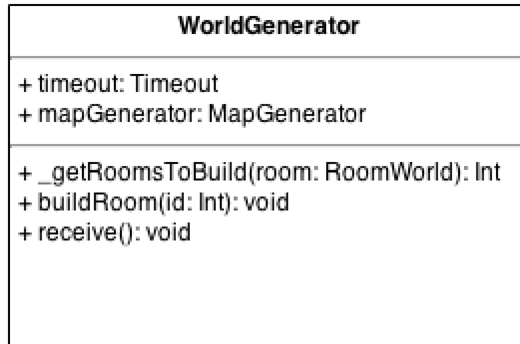

Figure 3.12.1 WorldGenerator

The **WorldGenerator** is a class that is instantiated in its own actor, and is responsible for all map generation calls.

<u>Attributes</u>

Name	Type	Description
getRoomsToBuild	Set[Int]	Fetches a list rooms (as identified by an integer) that need to be built.
buildRoom	Unit	Given a room id, build the room and add it to the world.

3.12.2 MapGenerator

MapGenerator

+ timeout: Timeout

+ rescaleNoise(noise: Array[Array[Double]]: Array[Array[I
+ findTransportLocation(direction: String,
map: Array[Array[Int]], width: Int, height: Int):
Position
+ doesPathExist(startX: Int, startY: Int,
endX: Int, endY: Int, map: Array[Array[Int]],
width: Int, height: Int): Boolean
+ calculateCollisionMap(map: Array[Array[Int]],
width: Int, height: Int): Array[Array[Int]]
+ ensureTraversable(map: Array[Array[Int]],
transportPositions: List[Position], width: Int, height: Int):
Boolean
+ receive(): void

Figure 3.12.2 MapGenerator

The **MapGenerator** class is responsible for the actual generation of maps. When the **WorldGenerator** calls for a map to be generated, all underlying map generation complexity is contained within the **MapGenerator**.

Attributes

Name	Type	Description
doesPathExist	Boolean	Given a map, determine whether or not a traversable path exists between a starting point and end point.
calculateCollisionMap	Array[Array[Int]]	Creates a map of tiles where each tile’s value is equal to how many non-collidable neighbors it has. Collidable tile’s value is 0.
ensureTraversable	Boolean	Ensures that the generated map is traversable by players.

4 Network System

This section defines the networking system which is responsible for receiving, interpreting, and processing network messages coming from the frontend. Additionally the network system is provides services for login, character creation, character selection, and the world editor. The networking system distributes workfromaNetworkMessageInterpreterSupervisorwhichsplitsJSONmessages from the

frontend to a pool of NetworkMessageInterpreters each of which individually converts the partition of messages it has received into different NetworkMessages. These NetworkMessage are then added to the NetworkMessageQueue. Once per frame rate the game loop flushes all the messages out of the queue and sends them to the NetworkMessageProcessorSupervisor. This supervisor in turn distributes them among a pool of NetworkMessageProcessors.

4.1 NetworkMessageQueue

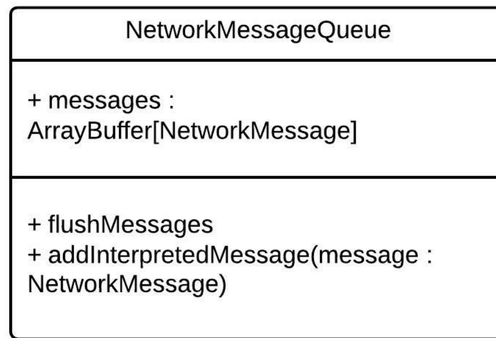

Figure 75: NetworkMessageQueue Class Diagram

NetworkMessageQueue is an actor that only accepts two types of messages. These messages are described below. The only member of this actor is an array called messages which stores case classes of type NetworkMessage.

Operation: AddInterpretedMessage(message: NetworkMessage)
Input: message: NetworkMessage - The message to be added to the queue.
Output: None
Description: Adds the message to the queue.

Operation: FlushMessages()
Input: none
Output: an array full of all the messages stored in the queue since the last flush.
Description: Returns the messages that have been stored since the last flush and empties the queue.

4.2 NetworkMessageInterpreter

Requirements met: 4.2

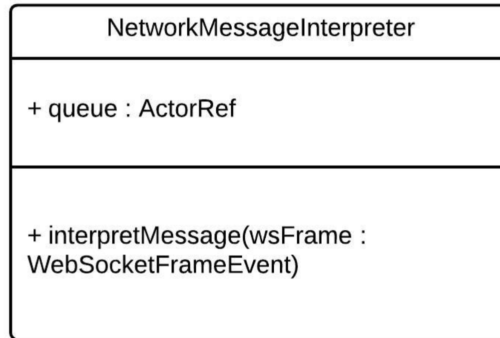

Figure 76: NetworkMessageInterpreter Class Diagram

NetworkMessageInterpreter is an actor which only accepts one type of message containing the case class InterpretMessage. InterpretMessage contains a string which is currently JSON. This may be optimized later to decrease bandwidth usage. However for now the JSON must contain an object with a field "type". This type field is then sent through a switch. The output of each case is a case class deriving from NetworkMessage which is sent to the NetworkMessageQueue actor instead of being outputted in a more traditional manner. The follow cases contain the other fields that must be specified along with the type field for each type.

Operation: interpretMessage(wsFrame: WebSocketFrameEvent
Input: wsFrame: WebSocketFrameEvent – The web socket the user connect with.
Output: Depends on the type of the message.
Description: Reads the message out of the WebSocket frame and extracts the type. It then matches on the type and handles it in the following ways.

Case: "init"(characterName: String)
Input: characterName: String - The name of the character to be added to the world.
Output: Adds AddNewCharacter and SocketCharacterMap messages to the queue.
Description: Creates an id for the character. It passes that id into the queue in the AddNewCharacter message with the characterName, the WebSocket, and starting positions. It also passes the character id and web socket to the queue via a SocketCharacterMap.

Case: "move"(start: boolean, dir: Int)
Input: start: boolean - Whether the action is starting or stopping.
dir: Int - An integer value 0-7. 0 is up and each subsequent value is 45 degrees to the right of the previous.
Output: Adds a MoveMessage to the queue containing the WebSocket, aMoveDirection, and start.

Description: Converts the dir int to a MoveDirection which is UpDirection, UpRightDirection, etc.

Case: "attack"()

Input: none
Output: Adds an attack message to the queue.
Description: The WebSocket is passed into the queue so that it can use it to look up which character issued the attack.

Case: "chat"(message: String, receiverName: String)

Input: message: String - The chat message to be sent.
receiverName: String - The name of the character the message is being sent to.
Output: Adds a ChatMessage to the queue which contains the message, the receiverName, and the WebSocket of the sender.
Description: the WebSocket is passed into the queue so that it can use it to look up which character sent the message.
Requirements met: 3.3.10.1.2

Case: "open"(containerId: String)

Input: containerId: String - The id of the container that is being opened by the character.
Output: Adds an OpenMessage to the queue.
Description: Passes the containerId and the WebSocket of the opener to the queue via the OpenMessage.

4.3 NetworkMessageProcessor

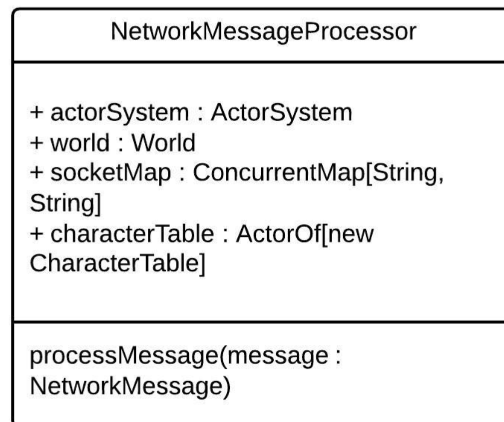

Figure 77: NetworkMessageProcessor Class Diagram

TheNetworkMessageProcessor receives NetworkMessages from the game loop and processes them in a variety of ways. Depending on the subtype of the NetworkMessage it is processed differently. The following cases show these different ways of processing NetworkMessages. These NetworkMessages store values which are essentially parameters to the case statement.

Name	Type	Description
actorSystem	ActorSystem	The Akka Actor System which stores all the actors.
world	World	World which all entities belongs to.
socketMap	ConcurrentMap[String, String]	Maps character ids to the id of their WebSocket connection.

Case: AddNewCharacter(webSocket: WebSocketFrameEvent, id: String, characterName: String, x: Int, y: Int)

Input: webSocket: WebSocketFrameEvent - TheWebSocket the character is connected to.

id: String - the entity id of the character to be added to the world. Not to be confused the id property from the database.

characterName: String - the name of the character to be added.

Output: Writes an initial message to the WebSocket so the frontend can load the game.

Description: Loads the character out of the database and creates an entity for it. Calculates level and adds the character to the world.

Case: AttackMessage(webSocket: WebSocketFrameEvent)

Input: webSocket: WebSocketFrameEvent - TheWebSocket the character is connected to.

Output: None

Description: Retrieves the character entity that is mapped to the WebSocket. It then spawns an attack entity in front of that character. Any entities who collide with that attack entity suffers the effects of that attack.

Case: OpenMessage(webSocket: WebSocketFrameEvent,containerId: String)

Input: webSocket: WebSocketFrameEvent - TheWebSocket the character is connected to. containerId: String - The id of the container entity which is being opened. Output: Writes a message to the WebSocket informing the frontend that the container has been opened.

Description: Removes the items from the container entity specified by containerId and puts them in the player’s inventory.

Case: SocketCharacterMap(webSocket: WebSocketFrameEvent,id: String)
Input: webSocket: WebSocketFrameEvent - TheWebSocket the character is connected to. id: String - The id of the character entity which is being added to the socketMap.
Output: None
Description: Adds an entry to the socketMap linking the character entity id to the WebSocket id. This allows for lookup of characters based on WebSockets.

Case: ChatMessage(webSocket: WebSocketFrameEvent, message: String, receiverName: String)
Input: webSocket: WebSocketFrameEvent - TheWebSocket the character is connected to.
message: String - The chat message that is being sent.
receiverId: String - The id of the character entity which the chat message is sent to.
Output: Writes a chat message to the receiving character's WebSocket. Also writes the message to the database.
Description: Looks up the sending character by the WebSocket id. Sends the chat message to the receiving character. Stores the message and both characters in the databse.

4.4 SockoServer

The SockoServer handles all requests from the frontend. It's only operation "run" sends handles requests differently based on whether they are an HTTPRequest or WebSocketFrame. HTTPRequests are further handled based on the path. Requirements met: 3.1, 3.2

WebSocketFrame When a WebSocketFrame comes it is simply sent to the NetworkMessageInterpreter wrapped within a InterpretMessage(wsFrame)case class where wsFrame is the WebSocket.

Path: /login Sends a LoginPost(httpRequest) message to the AuthorizationProcessor.
Path: /register Sends a RegisterPost(httpRequest) message to the AuthorizationProcessor.
Path: /chars Sends a CharactersPost(httpRequest) message to the AuthorizationProcessor.

4.5 AuthorizationProcessor

The AuthorizationProcessor is responsible for handling user actions outside of the game world. These actions are received at different routes. The AuthorizationProcessor receives different messages from the SockoServer. These messages are handled the following ways:

Case: LoginPost(request: HttpRequestEvent)

Input: request: HttpRequestEvent - The HTTPRequest which user sent.
username: String - An encrypted username for the user.
password: String - An encrypted password for the user.
Output: If the credentials are valid a HTTP 200 response with a token which can be used for further secure communication. If the credentials are invalid then a HTTP 401 response is outputted.
Description: Validates the user's credentials and either returns an authorization token or a HTTP 401 response.

Case: RegisterPost(request: HttpRequestEvent)

Input: request: HttpRequestEvent - The HTTPRequest which user sent.
username: String - An encrypted username for the user.
password: String - An encrypted password for the user.
Output: A 200 HTTP response if the username is not taken or a 401 HTTP response if it is.
Description: Registers the user within the database if the username is not taken.

Case: CharactersPost(request: HttpRequestEvent)

Input: request: HttpRequestEvent - The HTTPRequest which user sent.
token: String - The authorization token that was generated upon login. Output: A list of character data containing each character's name, level, and class. Description: Looks up all the characters associated with the user's account and returns them in a JSON list.

5 Ayai Web Application

5.1 Overview

This section covers the portion of the application that handles account details outside of the game client and world editor. The user interacts with these modules to handle character creation, character selection, and account settings.

5.2 Login Page

Requirements met:
3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.1.5, 3.1.6

The login module handles account registration and authenticating user's information. Users enter their information and choose to either login or register

with the information after being validated by the system. After this, the user is sent to the character selection screen.

5.3 Character Creation

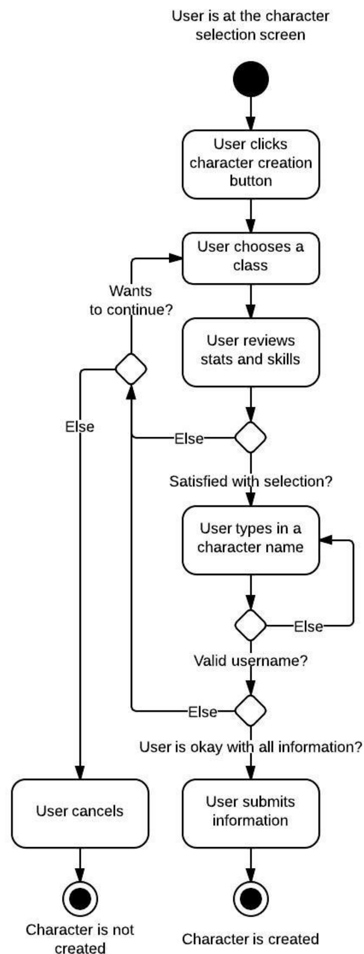

Figure 78: Activity diagram for creating a character

Requirements met:
3.2.1.6, 3.2.2.1, 3.2.2.2, 3.2.2.3, 3.2.2.4, 3.2.2.5, 3.2.2.6, 3.2.2.7, 3.2.2.7, 3.2.2.8, 3.2.2.9, 3.2.2.10, 3.2.2.11

This module allows users to create characters to play in the game. Users are presented with a list of characters and their descriptions. Once they choose their desired class, they choose an available character name and submit their preferences.

5.4 Character Selection

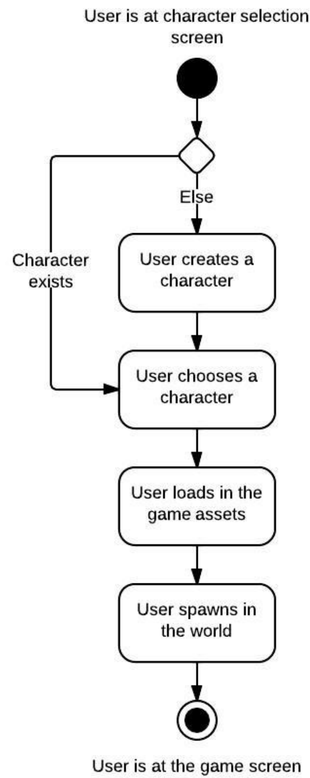

Figure 79: Activity diagram for selecting a character

Requirements met:
3.2.1.1, 3.2.1.5, 3.2.1.6

This module allows users to select their character. Once they have made their choice, they load into the game and begin playing.

5.5 Changing Settings

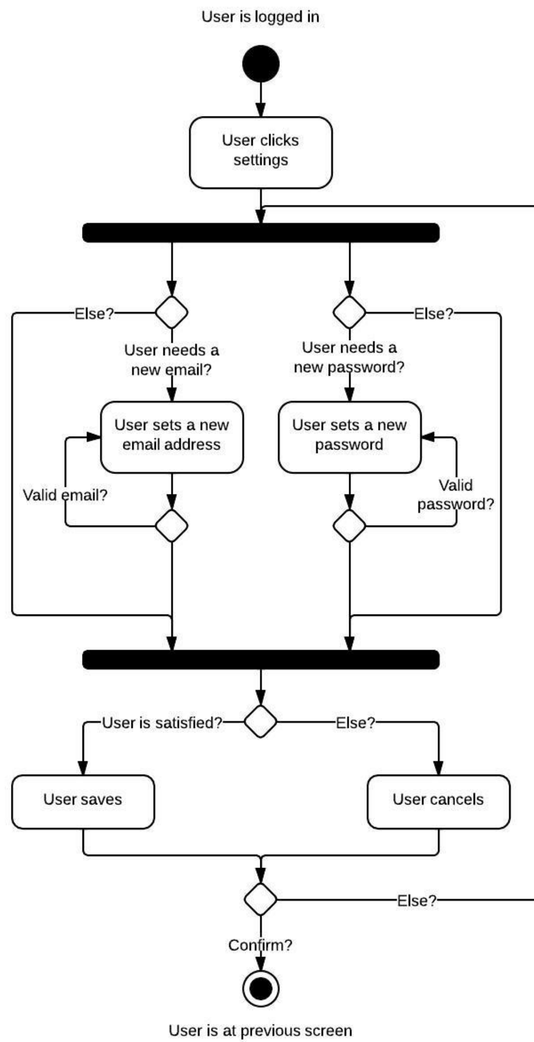

Figure 80: Activity diagram for changing account settings

Requirements met:
3.2.3.1, 3.2.3.2, 3.2.3.3, 3.2.3.4, 3.2.3.5

This module allows users to change their settings. They are presented with a form that allows them to set their email address and password.

6 Ayai World Editor

6.1 Searching

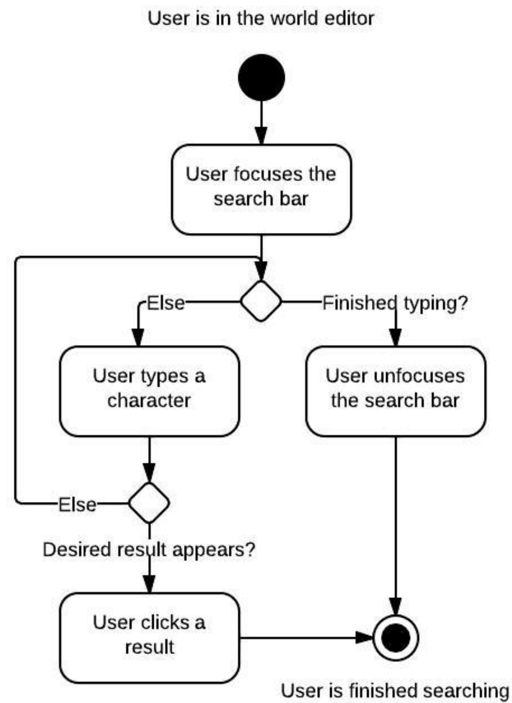

Figure 81: Activity diagram for searching the world editor

Requirements met:
2.1.1, 2.1.2, 2.1.3, 2.1.4, 2.1.5

This module allows users to search for entries within the world editor. As users type, the system makes suggestions to help them find what they are looking for.

6.2 Creating and Editing a New Entry

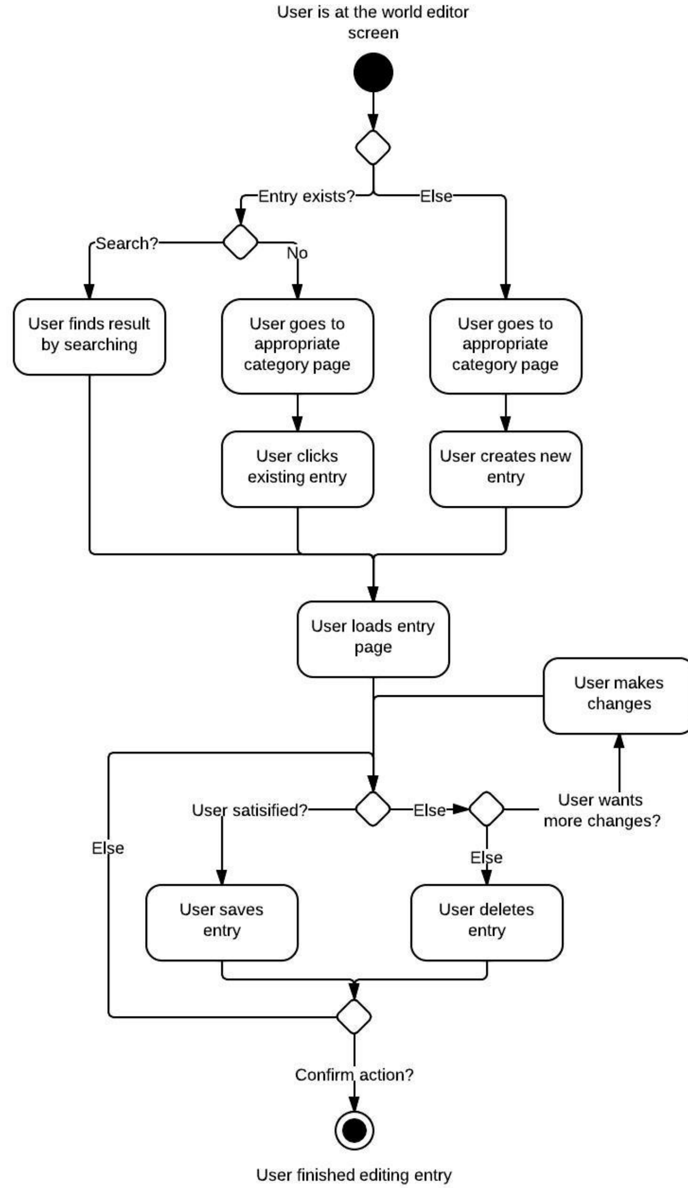

Figure 82: Activity diagram for adding an entry to the world editor

Requirements met:
2.1, 2.2, 2.3, 2.4, 2.5, 2.6

This module allows users to add the data that defines the game. Entries are objects such as classes, items, and spells. Once in the editor section, a user searches for an existing entry or uses the menu (as defined in the Ayai Software Requirements Document). To create a new entry, a user creates a new entry by going to that category's overview page where they find a button to create a new entry. At this point, they make changes that are relevant for that category. The user saves or cancels any changes after they finish.

7 Game Client

7.1 Overview

Figure 83: Game Client - UML Diagram

The Game Client is the frontend implementation of the Ayaiproject. It is a browser game leveraging WebGL,HTML5 canvas, and WebSocket technologies. The Game Client uses WebSockets to send requests to the server and receive responses, interpreting the data it receives to render the views in either WebGL canvas (Sprites and TileMaps) or HTML elements (UI Elements).

The game client makes extensive use of the open source JavaScript project Phaser.js (http://phaser.io/) in the following areas:

- Rendering the map and entity sprites in WebGL
- Mouse/Keyboard input

- Audio (Music and Sound Effects)

The full documentation for Phaser.js can be found here:
http://gametest.mobi/phaser/docs/Phaser.html

7.2 Graphics

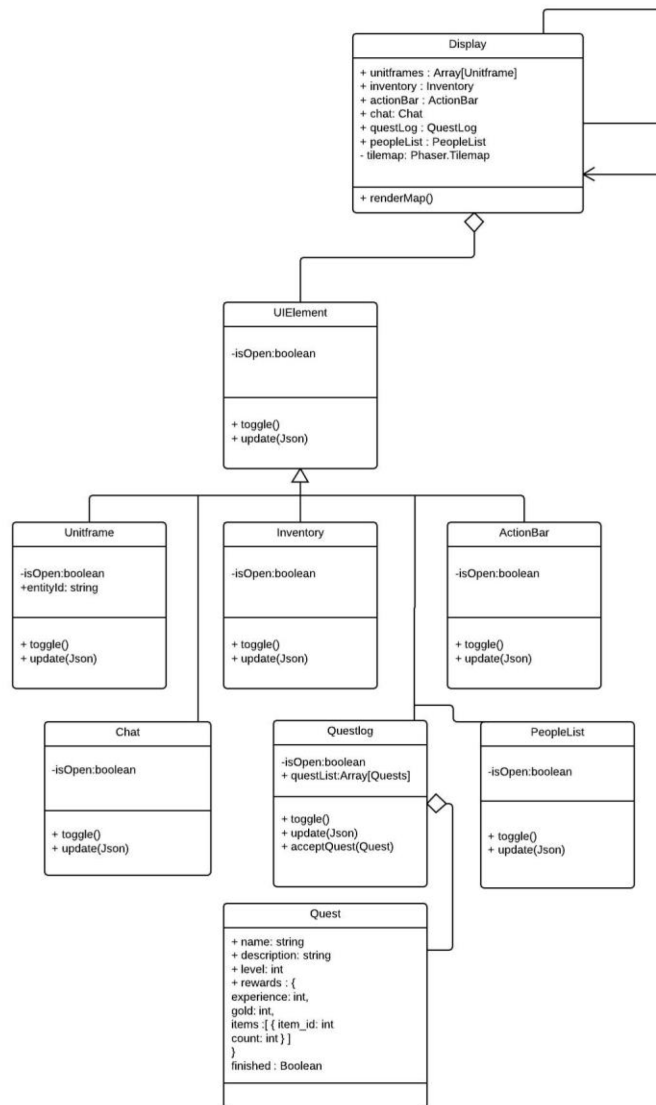

Figure 84: Game Client - UML Diagram - Graphics

7.2.1 Display

Attributes

Name	Type	Description
unitFrames	Array[Unitframe]	An array containing a reference to each Unitframe, including player, target, and group unitframes
inventory	Inventory	Reference to the singleton Inventory object
actionBar	ActionBar	Reference to the singleton ActionBar object
chat	Chat	Reference to the singleton Chat object
questLog	QuestLog	Reference to the singleton QuestLog object
peopleList	PeopleList	Reference to the singleton PeopleList object
tileMap	Phaser.Tilemap	Reference to renderable Phaser tilemap - constructed with a Tiled JSON object

Operations

Operation: renderMap(tileset: string, tilemap: string)
Input : tileset : string - name of the tileset loaded by Phaser tilemap : string - name of the tilemap loaded by Phaser
Output : None
Description : Indexes the loaded tileset and tilemap by their names, queries the browser for its dimensions and sets up the game camera/entities, then passes the tilemap to Phaser to be rendered in WebGL

7.2.2 UIElement

Attributes

Name	Type	Description
isOpen	boolean	A flag that denotes whether or not this UI Element is open and should be shown on screen

Operations

Operation: toggle()
Output : None
Description : Opens the UI element if isOpen is false and sets isOpen to true. Closes the UI element if isOpen is true and sets the isOpen to false.

Operation: update(json)
Input : json : string
Output : None
Description: Synchronizes the UI element on the given JSON, updating the view with the new values.

7.2.3 UnitFrame

Attributes

Name	Type	Description
isOpen	boolean	A flag that denotes whether or not this UI Element is open and should be shown on screen
entityId	string	The id of the entity whose vitals this unitframe is tracking

Operation: toggle()
Output : None
Description : Unused by Unitframes, these elements cannot be hidden by the player

Operation: update(json)
Input : json : string
Output : None
Description: Synchronizes the UI element on the given JSON, updating the view with the new values. Updates the health, mana, experience, and status effect views on the unitframe.

Requirements met: 3.3.2, 3.3.3, 3.3.6.1

7.2.4 Chat

Requirements Met: 3.3.10, 3.8.2

Attributes

Name	Type	Description
isOpen	boolean	A flag that denotes whether or not this UI Element is open and should be shown on screen

Operation: toggle()
Output : None
Description : Unused by Chat, this element cannot be hidden by the player

Operation: update(json)
Input : json : string
Output : None
Description: Synchronizes the UI element on the given JSON, updating the view with the new values. Updates the chat messages that have been sent to the player.

7.2.5 Inventory

Requirements Met: 3.3.6.1, 3.3.9.1, 3.3.9.4, 3.3.9.2, 3.3.9.3, 3.3.9.6.1, 3.3.9.6.2, 3.3.9.7, 3.3.9.5

Attributes

Name	Type	Description
isOpen	boolean	A flag that denotes whether or not this UI Element is open and should be shown on screen

Operation: toggle()
Output : None
Description : Unused by Chat, this element cannot be hidden by the player

Operation: update(json)
Input : json : string
Output : None
Description: Synchronizes the UI element on the given JSON, updating the view with the new values. Updates the chat messages that have been sent to the player.

7.2.6 QuestLog

Name	Type	Description
isOpen	boolean	A flag that denotes whether or not this UI Element is open and should be shown on screen
quests	Array[Quest]	The list of quests that the player has accepted

Operation: toggle()
Output : None
Description : Opens the UI element if isOpen is false and sets isOpen to true.
Closes the UI element if isOpen is true and sets the isOpen to false.

Operation: update(json)
Input : json : string
Output : None
Description: Synchronizes the UI element on the given JSON, updating the view with the new values. Updates the list of quests that the player has accepted.

7.2.7 Quest

Name	Type	Description
name	string	Name of the quest
description	string	Description of the quest
level	int	Level of the quest
rewards	Object	JavaScript object containing information for experience, gold, and items received for completing the quest
finished	boolean	Flag which indicates whether the player has completed this quest

Operation: toggle()
Output : None
Description : Opens the UI element if isOpen is false and sets isOpen to true.
Closes the UI element if isOpen is true and sets the isOpen to false.

Operation: update(json)
Input : json : string
Output : None

Description: Synchronizes the UI element on the given JSON, updating the view with the new values. Updates the list of players in the same room as the player.

7.2.8 PeopleList

Requirements Met: 3.3.6.3, 3.8.1

Name	Type	Description
isOpen	boolean	A flag that denotes whether or not this UI Element is open and should be shown on screen

Operation: toggle()
Output : None
Description : Opens the UI element if isOpen is false and sets isOpen to true. Closes the UI element if isOpen is true and sets the isOpen to false.

Operation: update(Json)
Input : json : string
Output : None
Description: Synchronizes the UI element on the given JSON, updating the view with the new values. Updates the list of players in the same room as the player.

7.2.9 Settings Menu

Name	Type	Description
isOpen	boolean	A flag that denotes whether or not this UI Element is open and should be shown on screen
Controls	Keys[Quest]	The list of keys bound to their functions.

Operation: toggle()
Output : None
Description : Opens the UI element if isOpen is false and sets isOpen to true. Closes the UI element if isOpen is true and sets the isOpen to false.

Operation: update(json)
Input : json : string
Output : None
Description: Synchronizes the UI element on the given JSON, updating the view with the new values. Updates the list of keys and their bindings.

7.3 Game

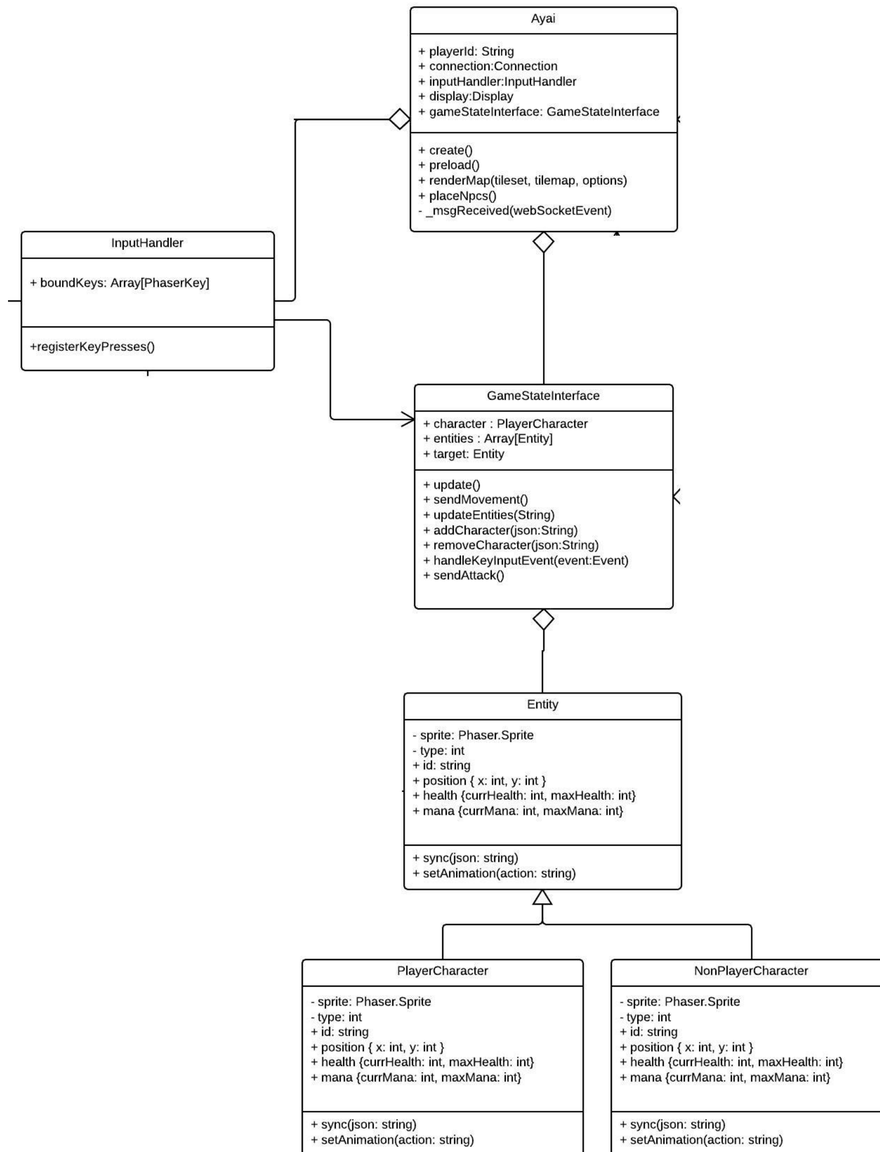

Figure 85: Game Client - UML Diagram - Game

7.3.1 Ayai

Attributes

Name	Type	Description
playerId	String	The players ID given by the server
connection	Connection	Games connection object
display	Display	Games display object
gameStateInterface	GameStateInterface	Games singleton copy of the gameStateInterface object.

Operation: preload()
Output : None
Description: Starts preloading all the assets. Calls create when the assets are finished loading.

Operation: create()
Output : None
Description: Creates all the UI elements for the game after the assets are loaded by preload.

Operation: _msgReceived(msg:Event)
Output : None
Description: Called when a message is received on the websocket connection. Dispatches the message to the correct location based on the type of message received.

7.3.2 GameStateInterface

Requirements met: 3.3.4, 3.3.5, 3.3.6, 3.3.7, 3.3.8

Attributes

Name	Type	Description
character	PlayerCharacter	The sessions current character
entities	Array[Entity]	Sprite given to phaser for rendering
target	Entity	Current entity which is selected in the game

Operations

Operation: update()

Output : Void

Description : Calls Phaser.JS to rerender the stage.

Operation: sendMovement()

Output : Void

Description : Use phaser.js to detect which keys are down and send the correct movement messages to the message sender.

Operation: updateEntities(json:String)

Input: json : JSON representation of entities to be updated in string format.

Output : Void

Description : Update the position of entities. Also handle the creation and deletion of entities.

Operation: addCharacter(json:String)

Input: json : JSON representation of character to be added.

Output : Void Description : Add character entity to GameStateInterfaces list of entities.

Operation: removeCharacter(json:String)

Input: json : JSON representation of character to be added.

Output : Void Description : Remove character entity to GameStateInterfaces list of entities.

Operation: handleKeyInputEvent(inputEvent:InputEvent)

Input: json : JSON representation of character to be added.

Output : Void Description : Handle keyboard inputs and send corresponding messages to the message sender based on which keys are pressed.

Operation: sendAttack()

Output : Void

Description : Send attack message to message sender.

7.3.3 InputHandler

Requirements Met: 3.5

Attributes

Name	Type	Description
boundKeys	Array[PhaserKey]	List of bound keys.

Operation: registerKeyPresses()
Output : Void
Description : Iterates over the bound keys and register them with the phaser keypress detection functions.

7.4 Net

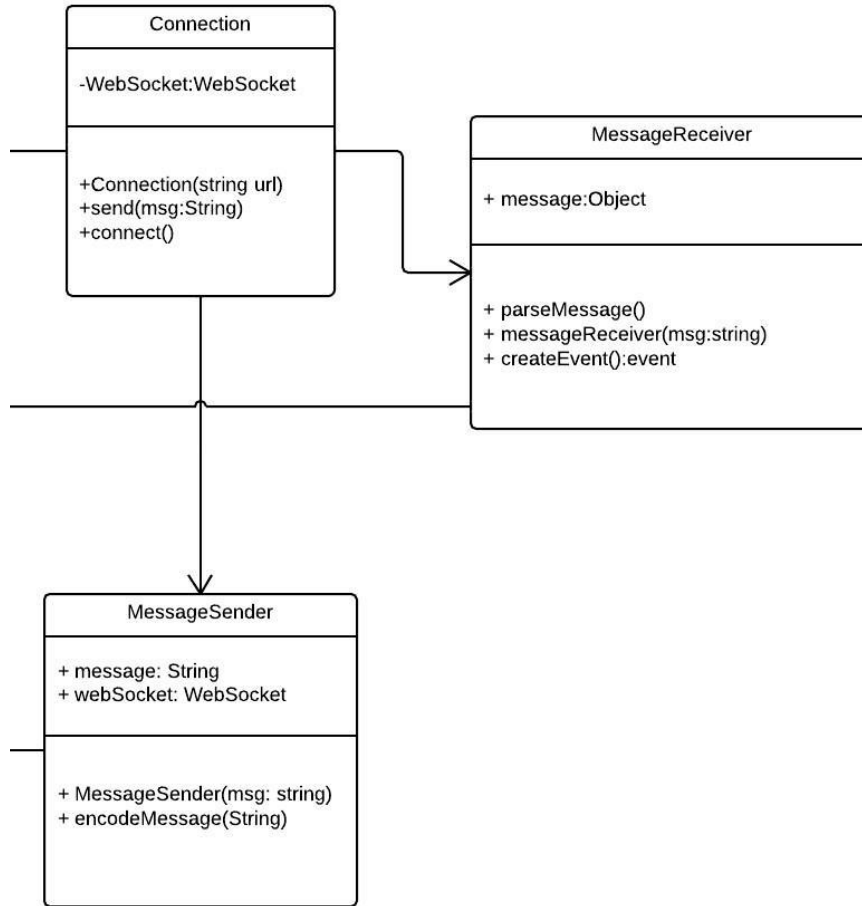

Figure 86: Game Client - UML Diagram - Net

7.4.1 Connection

Attributes

Name	Type	Description
webSocket	WebSocket	The websocket object for the connection to the backend

Operation: Connection(urlString: String)
Input : urlString : String : string of the url of the backend server
Output : Void

Description : Constructor for this class which takes the url of the backend server.

Operation: send(msg:String)
Input : msg : String : string of the message to be sent.
Output : Void
Description : Sends the message through the websocket to the backend.

Operation: connect()
Output : Void
Description : Creates the websocket object and starts the connection.

7.4.2 MessageReceiver

Attributes

Name	Type	Description
message	Object	Javascript Object version of the message after parsing.

Operation: MessageReceiver(message: String)
Input : message : String : JSON string representation of the message.
Output :Void
Description : Constructor for this class which calls parse on the passed in message string.

Operation: parseMessage(msg:String)
Input : msg : String : text to parse Output : Void
Description : Parses the passed in message and sets the class attribute message to the parsed object.

Operation: createEvent()
Output : Event
Description : Creates a message received event based on the message which has been parsed.

8 Database Design

The following is a UML style database diagram. It uses standard conventions. The only exception is the tag EK. EK stands for entity key. An entity key refers to an entity defined in the game files.

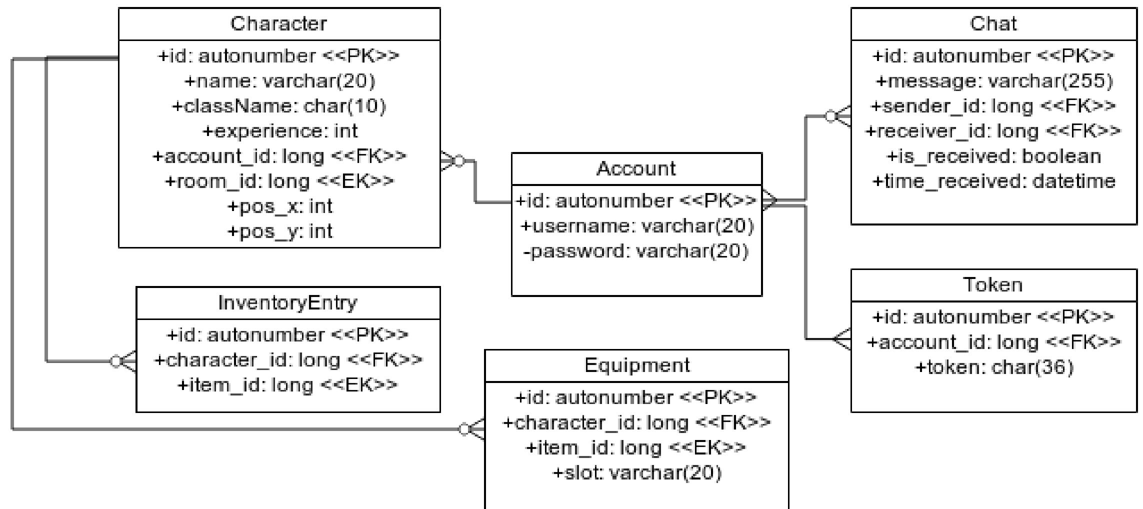

Figure 87: Database Diagram

Table: Account

id: autonumber - A unique id for each account.

username: varchar(20) - The account’s username. Must be 6-20 characters. email: varchar(20) - The user’s email address. Must be standard email formatting. password: varchar(20) - The user’s password. Must be 8-20 characters. Description: Each user of the system creates one account. This account is used for authentication and linking all the user’s data.

Table: Token

id: autonumber - A unique id for each token.

account_id: long - The account the token belongs to. This is a foreign key referencing the id field of the account table.

token: char(36) - The authentication token that is created when the user logs in. This is always 36 characters long.

Description: Each time a user logs in a token is created, sent to them, and stored in the database. The client uses this token to verify they are still the same user.

Table: Chat

id: autonumber - A unique id for each chat.

sender_id: long - The account that sent the chat message. This is a foreign key referencing the id field of the account table.

message: varchar(255) - The chat message that was sent.
receiver_id: long - The account chat message was sent to. This is a foreign key referencing the id field of the account table.
is_received: boolean - Indicates whether or not the chat has been received by the receiver account.
time_sent: datetime - The time the message was sent.

Table: Character

id: autonumber - A unique id for each character.
account_id: long - The account the character belongs to. This is a foreign key referencing the id field of the account table.
name: varchar(20) - A unique name for the character.
className: char(10) - The class of the character.
experience: int - The character's progress towards a certain level. The level attribute is calculated from this number using the experience array from the config files.
room_id: long - The id of the room the character is in. This id references the config files from which all the game content is loaded.
pos_x: int - The x position of the character within the room.
pos_y: int - The x position of the character within the room.

Table: InventoryEntry

id: autonumber - A unique id for each inventory entry.
character_id: long - The character the item belongs to. This is a foreign key referencing the id field of the character table.
item_id: long - The item that belongs to the character. This id references the config files from which all the game content is loaded.

Table: Equipment

id: autonumber - A unique id for each equipment entry.
character_id: long - The character the item belongs to. This is a foreign key referencing the id field of the character table.
item_id: long - The item that belongs to the character. This id references the config files from which all the game content is loaded.
slot: varchar(20) - This is the slot that the item is equipped in. When an item is equipped it is removed from the inventory table and added to the equipment table. When it is unequipped this process is removed.

9 Game Configuration File *

9.1 Purpose

Ayai-AI supports a Game Configuration file used to configure the application's artificial intelligence components. This file is used for determining which algorithm is used for a specific AI module. For example, if a developer writes a new Map Generation algorithm, that developer is able to configure AI to use this new algorithm through this file.

9.2 Design

The Game Config file is located at /src/main/resources/gameconfig.ayai

This file takes the form of a JSON object, with the following (example) structure

```
ai: {
  MapGeneration: "WorldGeneratorAlgorithm",
  Pathfinding: "RandomPathfindingAlgorithm",
  ...
}
```

This file is handled entirely by GameConfiguration.scala. In order for a developer to add support for a new category of AI configuration, they must edit that file and add support there. In the current iteration, only "MapGeneration" is supported.

If, while the webserver is running, one of the values in the file is changed, the system will automatically and immediately reflect that change.

This file is built to be highly extensible, and is not meant to be limited to AI-related configuration alone. Future support could include tile set configuration, database configuration, etc.

Glossary

A* a pathfinding algorithm that finds the most efficient path between 2 points.

ACID compliant A set of properties that guarantee that database transactions are processed reliably (Atomicity, Consistency, Isolation, Durability).

Action A spell or ability a character or an item can perform.

Administrator User with ability to ban users or give access to certain players.

Algorithm a step by step procedure for calculations and data processing.

Animation rapid display of static pictures based on certain player movement and commands.

ArrayBuffer A mutable list.

Authentication verify the user's credentials on the server to give access to game and characters.

Backend any processing that takes place remotely from the player's location.

Breadcrumb A navigation aid which allows users to keep track of their locations within the program.

Character A single entity in an MMORPG game world which can interact with the game world.

Character Level Measures the overall effectiveness of a character. As the character's level increases, so does the value of their statistics.

Character Statistics (Stats) Measure how effective a given character is at certain tasks. Example: Strength, Agility, Intellect.

Class a method of differentiating game characters that have different sets of abilities and statistics.

Component A structure of data which is held inside of an entity.

Cooldown after an attack has been down, there is a time based countdown before the player can do that same attack again.

Damage A reduction in a character's health.

Damage Type The type of damage that is being dealt to a character. Examples: fire, physical, etc.

Database organized collection of data and supports processing of information.

Effect A magical component which applies a status to its target.

Entity A list of components.

Entry An instance of content that defines the objects and actions that make up the game world.

Experience A value that measures a character's progress to the next Character Level.

Faction An organization within the game which NPCs may belong to.

Frontend any processing that takes place on the player's computer/application.

Game State The complete knowledge of everything contained within the game at a current point in time.

Game World The collection of all rooms, or zones and the characters they contain which are managed by the server(s).

Health A statistic which measures how much damage a character can sustain before the character dies.

HTTP Secure (HTTPS) An implementation of http with enhanced security..

HyperText Mark Up Language (HTML) The latest revision of a markup language used to organize content for the web.

HyperText Transfer Protocol (HTTP) An application protocol for distributed, collaborative, hypermedia information systems.

Java Virtual Machine (JVM) Java Virtual Machine.

Latency time delay experienced by a system.

Mana a resource that a character can expend to use different abilities.

Massively Multiplayer Online Game (MMO) An online video game in which there is a central game world managed by one or more servers to which many players, or clients, can connect in order to interact with one another.

Melee a short range attack that is only limited to the area immediately around a character.

Non-Player Character (NPC) Non-playable characters whose actions are processed by server(s) of an MMORPG.

Player A player is a person who controls an avatar.

Player Character (PC) The representation of a player in the MMORPG game world.

PostgreSQL An open-source object-relational database management system (ORDBMS) with an emphasis on extensibility and standards-compliance.

Prop A purely aesthetic visual element which has no impact on game play. (Example: a bush.).

Quest A mission with one or more objectives, usually resulting in a reward and/or story advancement when all objectives are complete.

Role Playing Game (RPG) A game in which players control characters intended to represent themselves.

Room One piece of the game world. Rooms will be connected by portals which will be the only way to enter or leave a room.

Scala A JVM programming language incorporating object oriented and functional programming paradigms.

Scala Build Tool (SBT) A tool used to compile and run Scala projects.

Sprite A small image which is used to represent a game entity.

Spritesheet A file which has multiple images representing different stages of animations for a game entity.

Status Effect An effect on a character/player/enemy that increases or decrease a statistic from the normal amount.

SuperUser a user with access to all abilities and moderation functions of an application.

Tilesheet A list of sprites for use in building a map.

Web Graphics Library (Web GL) is a javascript API for rendering interactive 3D graphics and 2D graphics within a compatible browser.