ENG1 Group 2 Marlin Studios
Part 2

Architecture

SKYE FULLER

JONATHAN HAYTER

ALEXANDER MIKHEEV

MARKS POLAKOVS

FREDERICK SAVILL

HAOWEN ZHENG

Architecture

Introduction

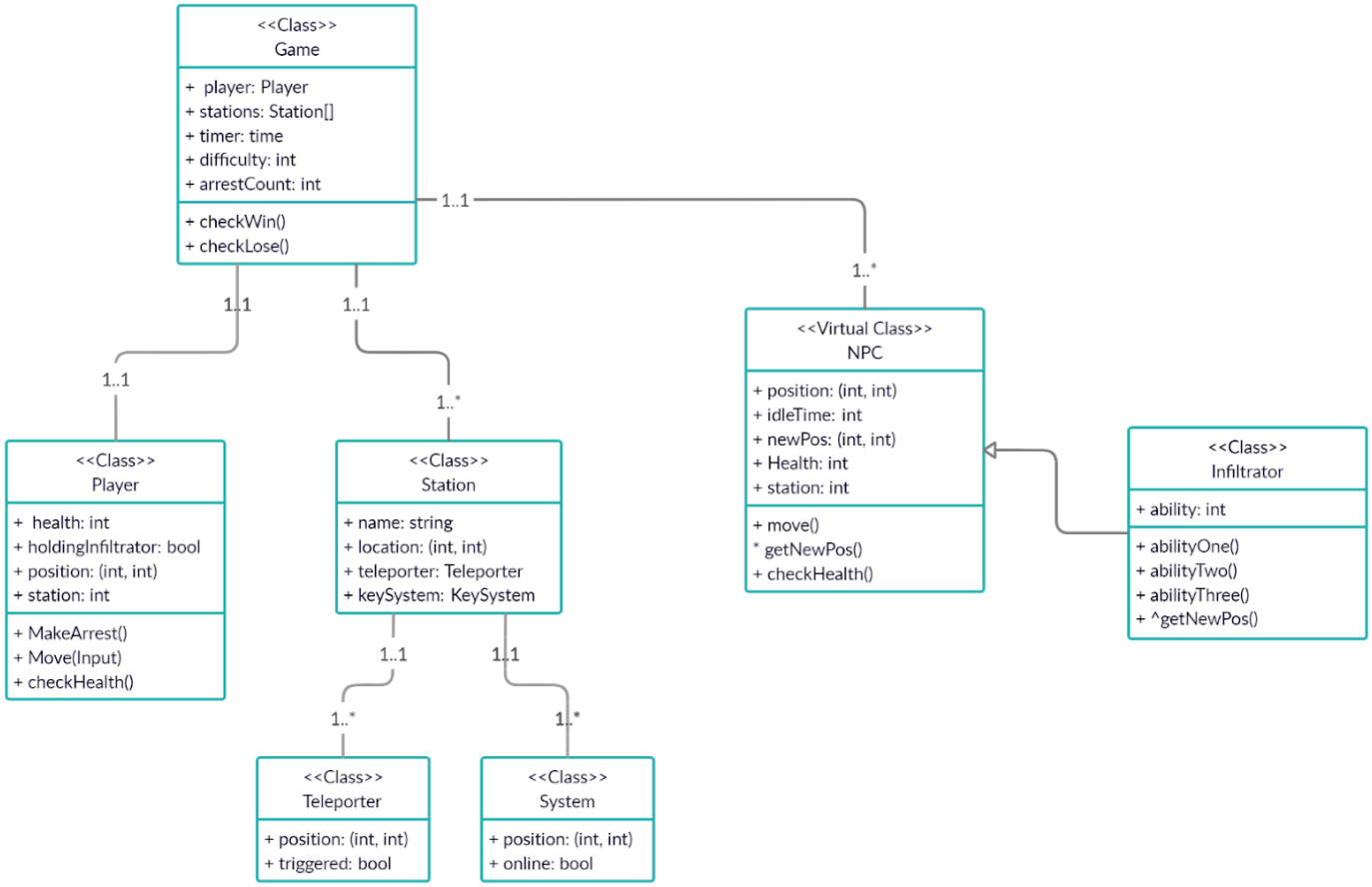
This document goes into detail about the architecture used for the implementation of the software, with justification as to why it was designed in this manner.

Generating our architecture plans

To generate our abstract and concrete architecture, we decided to utilise the Unified Modelling Language (UML). This is a standardised language consisting of diagrams which we will use to help ourselves when it comes to the implementation stage of our project. UML is well known, well documented and takes advantage of many best engineering practices, so we can be sure that the implementation process will be as seamless as possible.

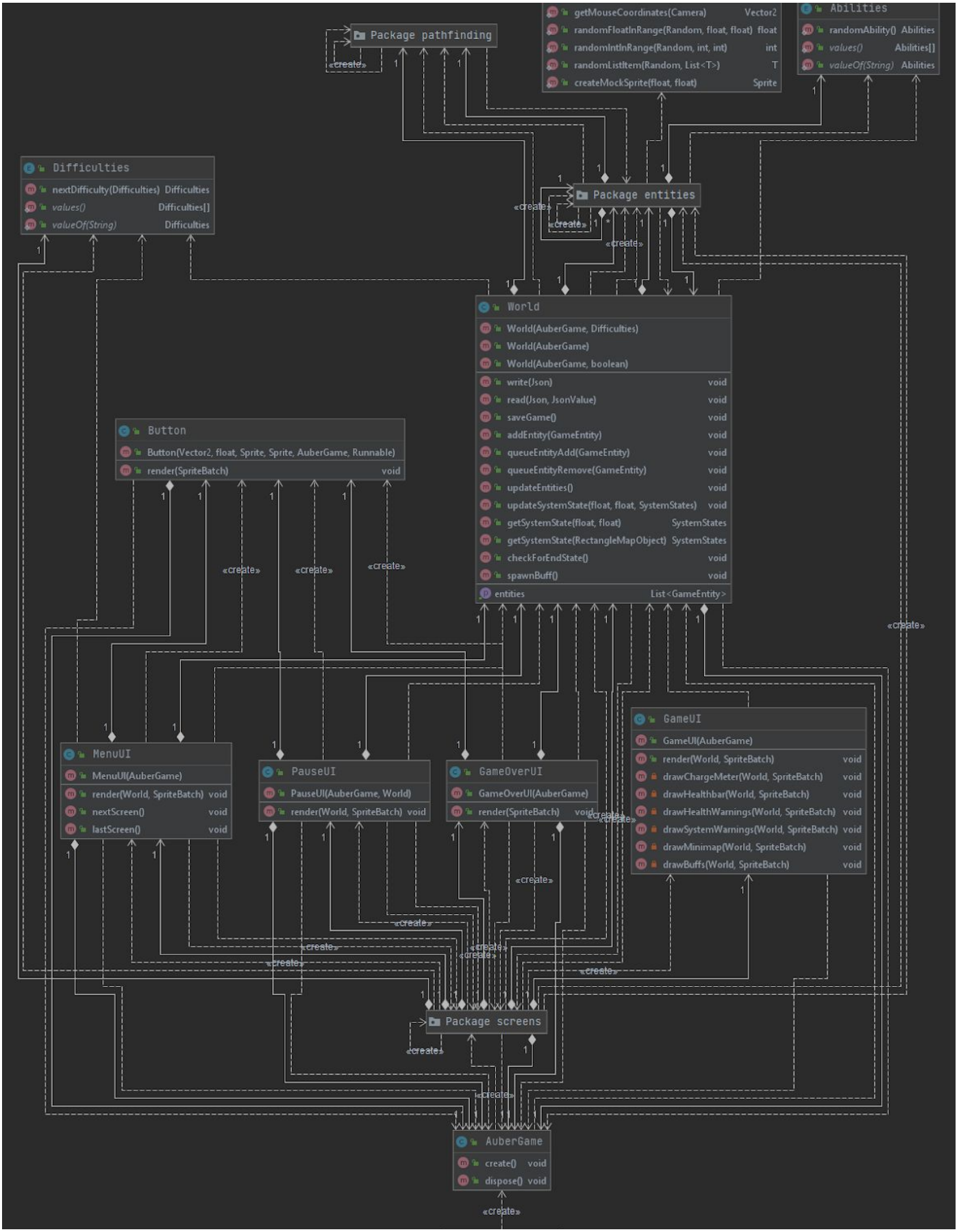
We created class diagrams to plan what classes would be needed, the attributes of each class and the operations it can perform, and the interactions required between those classes. We chose this method because of its flexibility and readability. As mentioned, it is also the most ubiquitous design language used by software developers so should be readily understood by our team. To create these diagrams, we used the websites creately.com and lucidchart.com due to their ease of use and accessibility.

Abstract Architecture

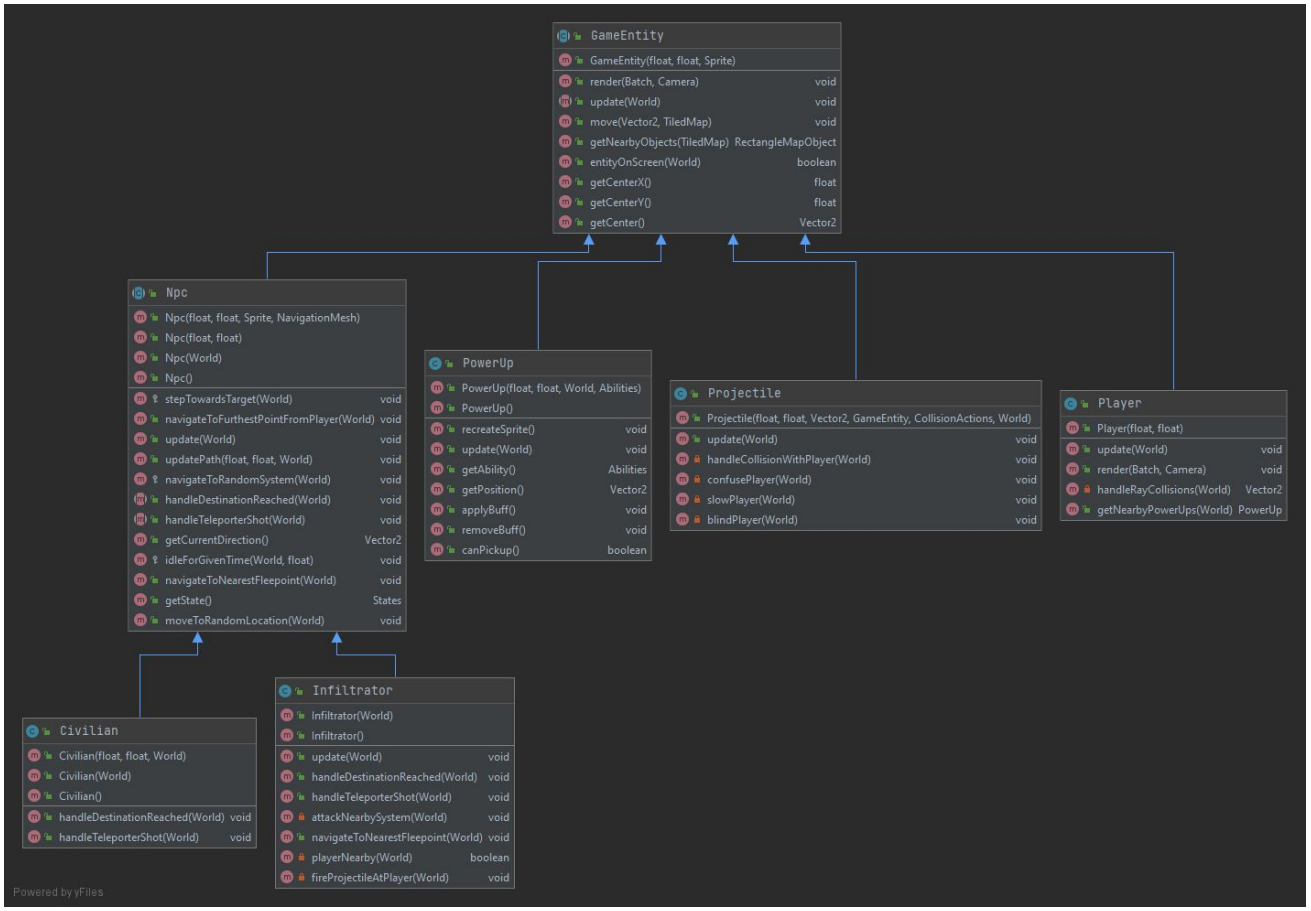


Concrete Architecture

Overview


Package pathfinding
«create»
getMouseCoordinates(Camera) Vector2
randomFloatInRange(Random, float, float) float
randomIntInRange(Random, int, int) int
randomListItem(Random, List<T>) T
createMockSprite(float, float) Sprite
Abilities
randomAbility() Abilities
values() Abilities[]
valueOf(String) Abilities
Difficulties
nextDifficulty(Difficulties) Difficulties
values() Difficulties[]
valueOf(String) Difficulties
Package entities
World
World(AuberGame, Difficulties)
World(AuberGame)
World(AuberGame, boolean)
write(Json) void
read(Json, JsonValue) void
saveGame() void
addEntity(GameEntity) void
queueEntityAdd(GameEntity) void
queueEntityRemove(GameEntity) void
updateEntities() void
updateSystemState(float, float, SystemStates) void
getSystemState(float, float) SystemStates
getSystemState(RectangleMapObject) SystemStates
checkForEndState() void
spawnBuff() void
entities List<GameEntity>
Button
Button(Vector2, float, Sprite, Sprite, AuberGame, Runnable)
render(SpriteBatch) void
MenuUI
MenuUI(AuberGame)
render(World, SpriteBatch) void
nextScreen() void
lastScreen() void
PauseUI
PauseUI(AuberGame, World)
render(World, SpriteBatch) void
GameOverUI
GameOverUI(AuberGame)
render(SpriteBatch) void
GameUI
GameUI(AuberGame)
render(World, SpriteBatch) void
drawChargeMeter(World, SpriteBatch) void
drawHealthbar(World, SpriteBatch) void
drawHealthWarnings(World, SpriteBatch) void
drawSystemWarnings(World, SpriteBatch) void
drawMinimap(World, SpriteBatch) void
drawBuffs(World, SpriteBatch) void
Package screens
AuberGame
create() void
dispose() void

GameEntity


GameEntity
GameEntity(float, float, Sprite)
render(Batch, Camera) void
update(World) void
move(Vector2, TiledMap) void
getNearbyObjects(TiledMap) RectangleMapObject
entityOnScreen(World) boolean
getCenterX() float
getCenterY() float
getCenter() Vector2
Npc
Npc(float, float, Sprite, NavigationMesh)
Npc(float, float)
Npc(World)
Npc()
stepTowardsTarget(World) void
navigateToFurthestPointFromPlayer(World) void
update(World) void
updatePath(float, float, World) void
navigateToRandomSystem(World) void
handleDestinationReached(World) void
handleTeleporterShot(World) void
getCurrentDirection() Vector2
idleForGivenTime(World, float) void
navigateToNearestFleepoint(World) void
getState() States
moveToRandomLocation(World) void
PowerUp
PowerUp(float, float, World, Abilities)
PowerUp()
recreateSprite() void
update(World) void
getAbility() Abilities
getPosition() Vector2
applyBuff() void
removeBuff() void
canPickup() boolean
Projectile
Projectile(float, float, Vector2, GameEntity, CollisionActions, World)
update(World) void
handleCollisionWithPlayer(World) void
confusePlayer(World) void
slowPlayer(World) void
blindPlayer(World) void
Player
Player(float, float)
update(World) void
render(Batch, Camera) void
handleRayCollisions(World) Vector2
getNearbyPowerUps(World) PowerUp
Civilian
Civilian(float, float, World)
Civilian(World)
Civilian()
handleDestinationReached(World) void
handleTeleporterShot(World) void
Infiltrator
Infiltrator(World)
Infiltrator()
update(World) void
handleDestinationReached(World) void
handleTeleporterShot(World) void
attackNearbySystem(World) void
navigateToNearestFleepoint(World) void
playerNearby(World) boolean
fireProjectileAtPlayer(World) void
Powered by yFiles

The abstract architecture pictured above is intended as a prescriptive tool to guide the implementation of the project. It gives an overview of what must be implemented without giving strict rules about how it must be implemented, to allow for flexibility in development.

The classes and methods illustrated in our abstract architecture reflect the requirements we have elicited from our brief and customer interviews. Design decisions such as infiltrators' special abilities are represented in the diagram, but without the decisions about the implementations of those abilities.

The diagram contains a low level of detail for a few different reasons. It allows us to capture only the most critical information at this early design stage to ensure we have the ideas for the game's core features. This allows for later expansion and additional features after the minimum viable product has been delivered. It also allows for flexibility in developing the concrete architecture if the implementation must be changed due to limitations of the engine or evolving requirements. Our abstract architecture does not contain any language or engine specific methods so that it can be adapted to the tools and methods we decide to use for the project, for example if these were to change in development.

Extrapolating from the abstract design, the developed concrete architecture is pictured above. While the abstract architecture outlines the initial thoughts regarding the implementation of the software, it is limited in its scope – it only has 7 classes. During development of the program, we add more classes and classify them into 5 main categories, GameUI, Screen, World, Infiltrators, and Pathfinding. We can find, for example, during development that we delete the Station class and change it to Pathfinding and Screen classes. NPC and Player class are all included in the Entities class. Additionally, we add the World class which include the basic information regarding the world. These are some of the main differences between the two drafts of the software architecture.

Below will be represented many of the main changes from the development of the abstract architecture into the concrete representation, and how they correlate directly with requirements outlined in our project's requirements document for traceability.

GameUI

GameUI includes 3 classes. Its functions are responsible for features such as returning the UI elements for rendering, such as buttons and title text.

PauseUI

PauseUI is responsible for rendering the pause menu UI in GameScreen. It calls functions in World, and AuberGame depending on what button is pressed.

MenuUI

MenuUI is responsible for drawing the UI on MenuScreen, it calls functions in AuberGame, and Difficulties depending on what is pressed.

GameOverUI

GameOverUI is responsible for drawing the UI on GameOverScreen, it calls functions from AuberGame in order to return the user to the menu, or start a new game.

Screens

Screens includes 3 classes.

- MenuScreen is responsible for showing what is on the menu screen (the page the player is shown upon first load of the game).
- GameScreen is used when the game is being played.
- GameOverScreen is displayed when the player wins or loses the game.

Requirement **FR_MENU** is achieved here, alongside **FR_WIN_CONDITION** and **FR_LOSS_CONDITION**.

World

The World class stores information about the game world, which is used in other aspects of the software such as the GameScreen page. General information such as the navigation mesh and charge rate for the Auber's weapon is determined here. Furthermore, the save and load functions are stored here (**saveGame(), read()** & **write()**)
This class is designed to fulfil **FR_MAP**, **FR_MENU** and **UR_SAVE**.

Entities

Entities includes many classes. It is responsible for holding information about the entities in the game.

- GameEntity contains all the information regarding the entities in a generalised way. Information such as speed, friction and position are stored here, to be obtained by classes implementing from this master class. This fulfils the **FR_HOSTILES** and **FR_SABOTAGE** requirements.
- The NPC class contains two subclasses:
    - The Infiltrator class represents the hostile infiltrators which need to be defeated by the player and attack the key systems in the station.
    - The Civilian class represents the non-hostile aliens on-board the station.

  These two classes fulfil the requirements **FR_ALIENS_COUNT, FR_ALIENS**, and **FR_HOSTILES_SPECIAL.**
- The Player class represents the entity controlled by the user.
  This class fulfils the requirements **FR_ATTACK_NOTIF**, **FR_HEAL**, **FR_ARREST**.
- The PowerUp class represents the entity that grants the player certain buffs (See Abilities)
  This class fulfills the requirements **FR_POWER_UPS**

Pathfinding

Pathfinding includes two classes and is responsible for the movement of entities around the map.

- NavigationMesh is responsible for the map section of the game - it gives the entities a reminder about the accessible path around the map, allowing them to navigate between key locations in the most efficient way possible. This fulfils the requirements **FR_TELEPORTER** and **FR_MAP.**
- PathNode is a class designed to aid in the navigation of entities around the map using heuristics and path costs. This will fulfil the requirement **FR_PLAYER_TILES**.

Difficulties & Abilities

Difficulties & Abilities were introduced in order to fulfil the requirements **FR_DIFFICULTY** and **FR_POWER_UPS_ABILITIES**. They are both enum types. Difficulties has a function that returns the next difficulty from the current.

Finally, general changes were made all over the code base in order to get testing to work, all this was, was an if statement that checked if the game was running in headless mode, if so, objects would not be rendered.