ARCHITECTURE

Cohort 1 group 11

Freddie Aberdeen
Mutaz Ghandour
James Given
Jasper Owen
Jungwan Park
Joe Reece
Ivan Shestakov

Structural and Behavioural Diagrams

Languages and Tools Used:

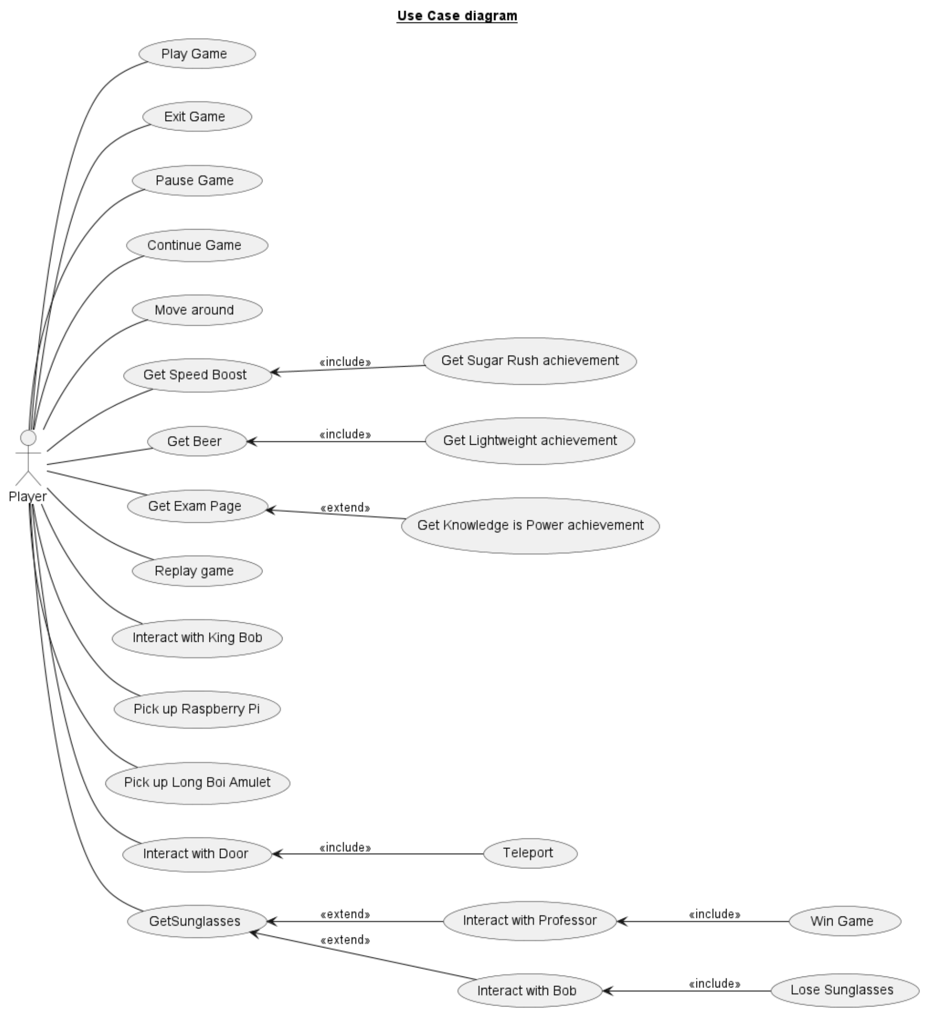
The architectural representations of the system were modelled using the Unified Modeling Language (UML). Some of the interim diagrams were produced using UMLet, which was chosen for its compatibility with standard UML notation. However, later diagrams were produced using plantUML, which was also compatible with standard UML notation.

- Structural aspects: The Structural Aspects of our architecture were shown with a class diagram, which focused on the game's interfaces and classes. This perspective was chosen as it shows the contents of the classes and how they interact with each other, without the addition of implementation-level detail.

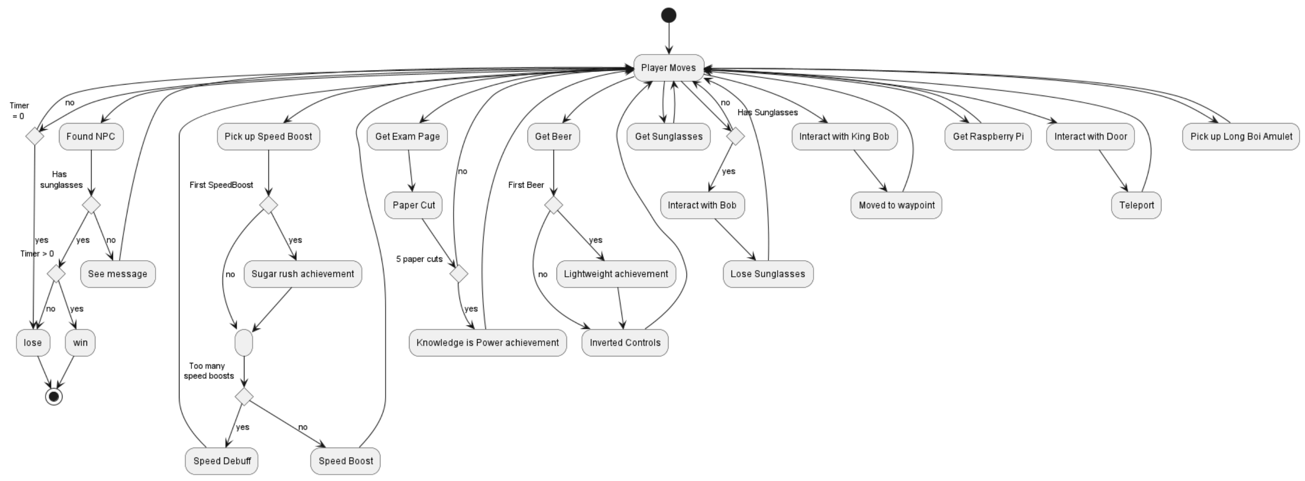
- Behavioural aspects: Illustrated using Use Case and Activity Diagrams, with a user perspective and workflow perspective respectively. These diagrams demonstrate how a user can interact with the system, as well as an example of what will happen when they play the game

Behavioural Diagrams:

Use-Case Diagram:

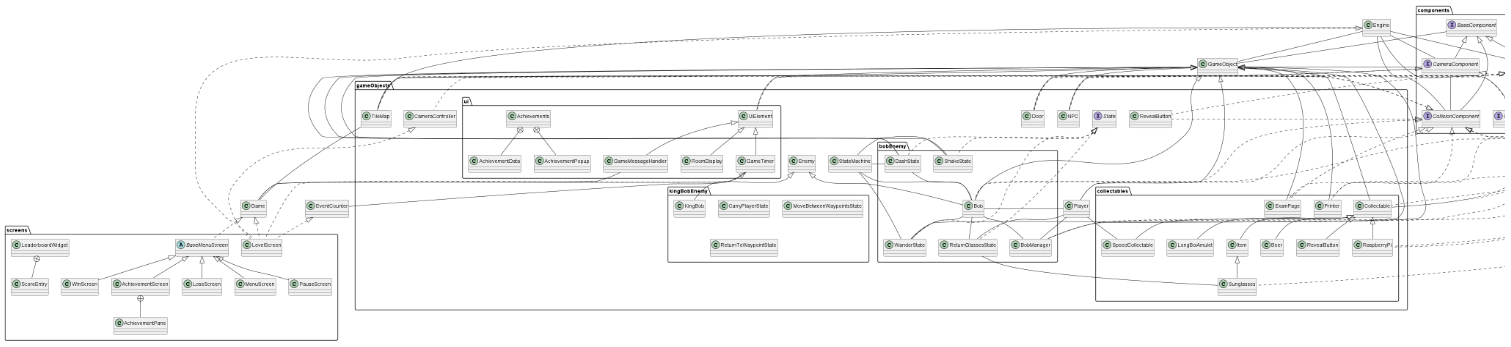


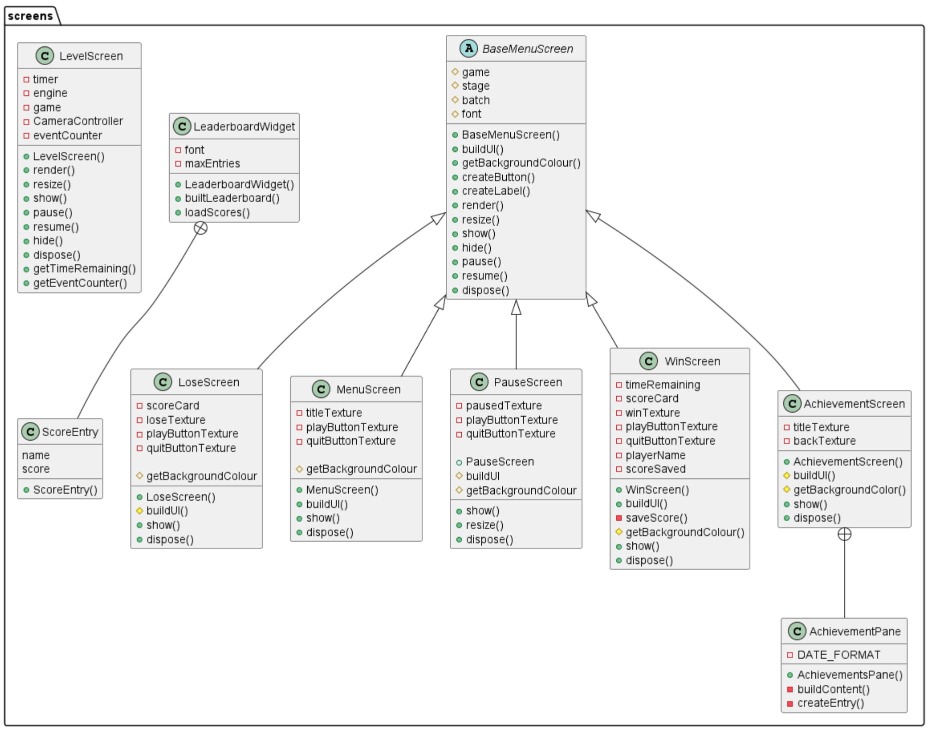
Activity Diagram:

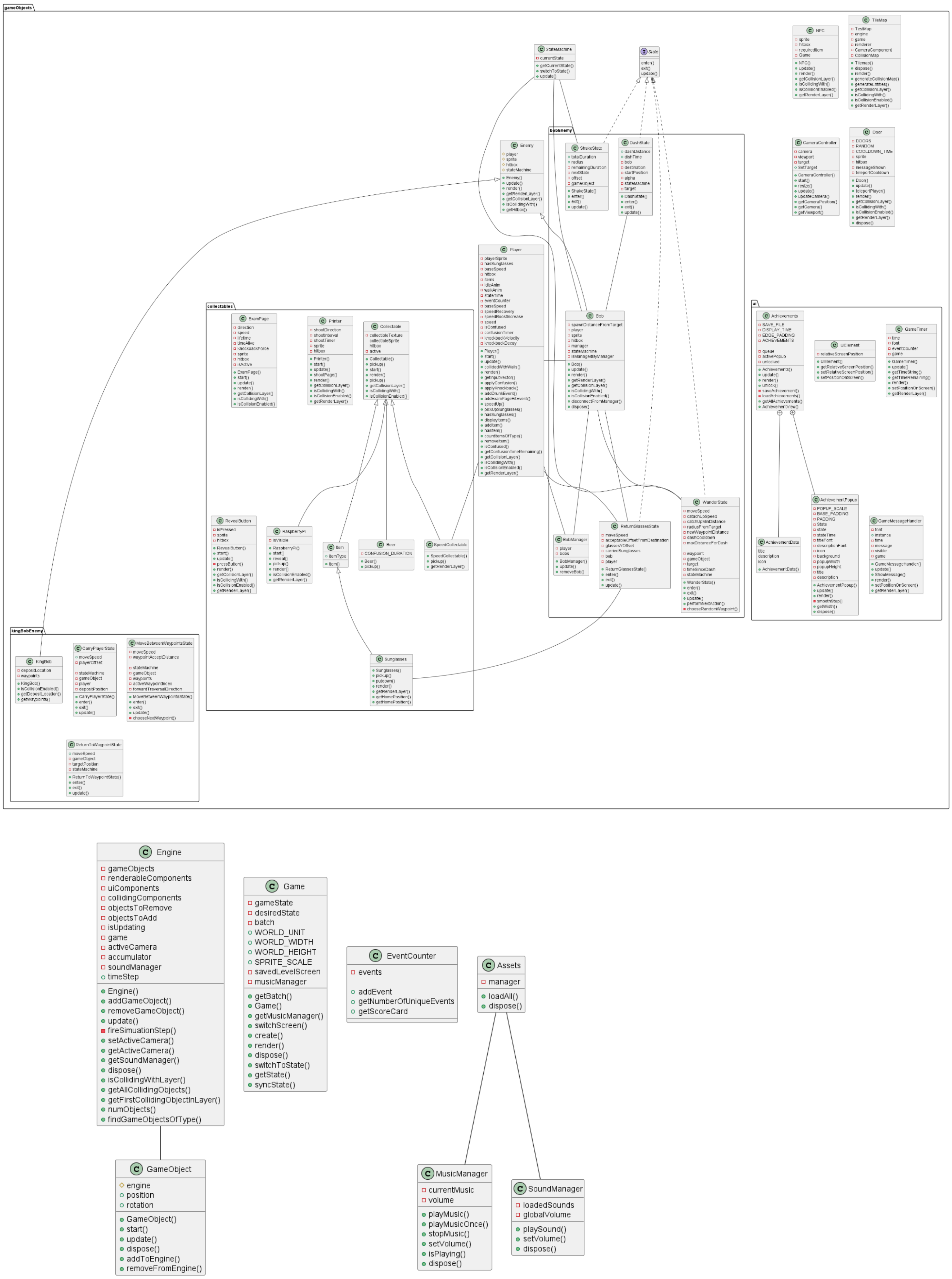


Structural Diagrams:

The following classes show all the classes of our game, with sub-diagrams for specific packages:






gameObjects
StateMachine
currentState
getCurrentState()
switchToState()
update()
State
enter()
exit()
update()
NPC
sprite
hitbox
requiredItem
Game
NPC()
update()
render()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
getRenderLayer()
TileMap
TestMap
engine
game
renderer
CameraComponent
CollisionMap
Tilemap()
dispose()
render()
generateCollisionMap()
generateEntities()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
getRenderLayer()
bobEnemy
ShakeState
totalDuration
radius
remainingDuration
nextState
offset
gameObject
ShakeState()
enter()
exit()
update()
DashState
dashDistance
dashTime
bob
destination
startPosition
alpha
stateMachine
target
DashState()
enter()
exit()
update()
CameraController
camera
viewport
target
SetTarget
CameraController()
start()
resize()
update()
updateCamera()
getCameraPosition()
getCamera()
getViewport()
Door
DOORS
RANDOM
COOLDOWN_TIME
sprite
hitbox
messageShown
teleportCooldown
Door()
update()
teleportPlayer()
render()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
getRenderLayer()
dispose()
Enemy
player
sprite
hitbox
stateMachine
Enemy()
update()
render()
getRenderLayer()
getCollisionLayer()
isCollidingWith()
getHitbox()
Player
playerSprite
hasSunglasses
baseSpeed
hitbox
items
idleAnim
walkAnim
stateTime
eventCounter
baseSpeed
speedRecovery
speedBoostIncrease
speed
isConfused
confusionTimer
knockbackVelocity
knockbackDecay
Player()
start()
update()
collidedWithWalls()
render()
getInputVector()
applyConfusion()
applyKnockback()
addDrunkEvent()
addExamPageHitEvent()
speedUp()
pickUpSunglasses()
hasSunglasses()
displayItems()
addItem()
hasItem()
countItemsOfType()
removeItem()
isConfused()
getConfusionTimeRemaining()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
getRenderLayer()
Bob
spawnDistanceFromTarget
player
sprite
hitbox
manager
stateMachine
isManagedByManager
Bob()
update()
render()
getRenderLayer()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
disconnectFromManager()
dispose()
collectables
ExamPage
direction
speed
lifetime
timeAlive
knockbackForce
sprite
hitbox
isActive
ExamPage()
start()
update()
render()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
Printer
shootDirection
shootInterval
shootTimer
sprite
hitbox
Printer()
start()
update()
shootPage()
render()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
getRenderLayer()
Collectable
collectibleTexture
collectibleSprite
hitbox
active
Collectable()
pickup()
start()
render()
pickup()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
ui
Achievements
SAVE_FILE
DISPLAY_TIME
EDGE_PADDING
ACHIEVEMENTS
queue
activePopup
unlocked
Achievements()
update()
render()
unlock()
saveAchievement()
loadAchievements()
getAllAchievements()
AchievementView()
UIElement
relativeScreenPosition
UIElement()
getRelativeScreenPosition()
setRelativeScreenPosition()
setPositionOnScreen()
GameTimer
time
font
eventCounter
game
GameTimer()
update()
getTimeString()
getTimeRemaining()
render()
setPositionOnScreen()
getRenderLayer()
RevealButton
isPressed
sprite
hitbox
RevealButton()
start()
update()
pressButton()
render()
getCollisionLayer()
isCollidingWith()
isCollisionEnabled()
getRenderLayer()
RaspberryPi
isVisible
RaspberryPi()
start()
reveal()
pickup()
render()
isCollisionEnabled()
getRenderLayer()
Item
itemType
Item()
Beer
CONFUSION_DURATION
Beer()
pickup()
SpeedCollectable
SpeedCollectable()
pickup()
getRenderLayer()
BobManager
player
bobs
BobManager()
update()
removeBob()
ReturnGlassesState
moveSpeed
acceptableOffsetFromDestination
glassesYOffset
carriedSunglasses
bob
player
ReturnGlassesState()
enter()
exit()
update()
WanderState
moveSpeed
catchUpSpeed
catchUpMinDistance
radiusFromTarget
newWaypointDistance
dashCooldown
maxDistanceForDash
waypoint
gameObject
target
timeSinceDash
stateMachine
WanderState()
enter()
exit()
update()
performNextAction()
chooseRandomWaypoint()
AchievementData
title
description
icon
AchievementData()
AchievementPopup
POPUP_SCALE
BASE_PADDING
PADDING
State
state
stateTime
titleFont
descriptionFont
icon
background
popupWidth
popupHeight
title
description
AchievementPopup()
update()
render()
smoothStep()
getWidth()
dispose()
GameMessageHandler
font
instance
time
message
visible
game
GameMessageHandler()
update()
ShowMessage()
render()
setPositionOnScreen()
getRenderLayer()
kingBobEnemy
KingBob
depositLocation
waypoints
KingBob()
isCollisionEnabled()
getDepositLocation()
getWaypoints()
CarryPlayerState
moveSpeed
playerOffset
stateMachine
gameObject
player
depositPosition
CarryPlayerState()
enter()
exit()
update()
MoveBetweenWaypointsState
moveSpeed
waypointAcceptDistance
stateMachine
gameObject
waypoints
activeWaypointIndex
forwardTraversalDirection
MoveBetweenWaypointsState()
enter()
exit()
update()
chooseNextWaypoint()
Sunglasses
Sunglasses()
pickup()
putdown()
render()
getRenderLayer()
getHomePosition()
getHomePosition()
ReturnToWaypointState
moveSpeed
gameObject
targetPosition
stateMachine
ReturnToWaypointState()
enter()
exit()
update()
Engine
gameObjects
renderableComponents
uiComponents
collidingComponents
objectsToRemove
objectsToAdd
isUpdating
game
activeCamera
accumulator
soundManager
timeStep
Engine()
addGameObject()
removeGameObject()
update()
fireSimuationStep()
setActiveCamera()
getActiveCamera()
getSoundManager()
dispose()
isCollidingWithLayer()
getAllCollidingObjects()
getFirstCollidingObjectInLayer()
numObjects()
findGameObjectsOfType()
Game
gameState
desiredState
batch
WORLD_UNIT
WORLD_WIDTH
WORLD_HEIGHT
SPRITE_SCALE
savedLevelScreen
musicManager
getBatch()
Game()
getMusicManager()
switchScreen()
create()
render()
dispose()
switchToState()
getState()
syncState()
EventCounter
events
addEvent
getNumberOfUniqueEvents
getScoreCard
Assets
manager
loadAll()
dispose()
GameObject
engine
position
rotation
GameObject()
start()
update()
dispose()
addToEngine()
removeFromEngine()
MusicManager
currentMusic
volume
playMusic()
playMusicOnce()
stopMusic()
setVolume()
isPlaying()
dispose()
SoundManager
loadedSounds
globalVolume
playSound()
setVolume()
dispose()

Justification and Evolution of Architecture

Justification:

The project required a flexible and modular game design to handle multiple entity types with varying behaviours and methods, including unusual features such as the speed boost mechanic, where using boosts too quickly results in temporary slowness.

Therefore, Object-Oriented architecture was chosen, because it contains these features:

- Reusable classes: In our game there are a series of re-usable classes containing data and systems that can be used by multiple objects, including when one class inherits from another, reducing the need for repeated code
- Maintainability: Isolated classes reduce the risk of cascading bugs and makes debugging or updating individual components easier.
- Scalability: New objects or classes can be added without affecting existing objects or classes, allowing the game to expand easily beyond the first iteration.
- Alignment with Agile Development: Objects and classes can be developed and integrated incrementally, supporting parallel work and iterative development.

Overall, the Object-Oriented architecture provides a flexible, maintainable, and scalable framework - making it the most suitable choice to support this project.

Packages

As shown in the class diagram, several of the classes in the game are assigned to separate packages. This allows us to group classes with similar code and/or purpose together to make our classes easier to manage and easier to locate when we wish to edit a specific class. This also allows us to assign new classes a specific package to make it easier to find them in future development.

The classes outside of any package, such as Game and Engine, are central classes that are essential to the game and are often used by multiple classes stored in the packages. For example, the class GameObject is used, sometimes through inheritance, by every class in the gameObjects package. Also, the class Engine is responsible for the core logic of the game, including updating the game, meaning there would be no logical package to assign it to.

Design Evolution:

The design process for the game followed an iterative and evolutionary approach, starting from a simple object-oriented structure and gradually moving toward a flexible, component-based model as the project complexity increased.

Initial Design Phase

- At the beginning of development, the design was considerably more straightforward, with elements being hardcoded in the base controlling class first with the intention to refactor in the future.
- The main class had a reference to each individual game object and all game objects (eg. Player, Map, Key) extended a common GameObject class

Addition of interfaces

As we added more classes to the game, this approach became difficult to maintain. Therefore, we extracted behaviours shared by multiple classes, such as rendering and collision, to interfaces.

Each class then only implemented the interfaces it needed. We did this because it gave us more flexibility in which classes could use which interface, and allowed us to re-use more of our pre-existing code.

Further development

- We decided to create an Engine class that is responsible for creating and handling the objects in our game. This allowed us to remove some logic from the game class which made it easier to test, as well as making it easier to test the engine class.

On the team's website (https://eng1-c1g11.github.io/escape-from-uni-website) there are interim architectural diagrams, as well as CRC cards which provide evidence of the design process followed.

Architectural Refinements

.

The project was reorganized into packages with a small structural change. This made it easier to keep track of what each class was responsible for and improved navigation of the project as a whole. In addition, variables were made private to improve encapsulation and make the code less surprising. Previously, public variables were modified from multiple classes, which made it difficult to achieve predictable behaviour, as any class could modify a variable at any point.

The player class had too many responsibilities, which made the logic complicated. As a result, the player contained logic related to events and direct interactions with UI elements. This was refactored to separate responsibilities and introduce an EventCounter that contained the event logic. This resulted in smaller units that were easier to test independently.

Initially, we had a Game class which contained a large loop that handled all the different states the game could be in, such as menus, gameplay, and paused states. This resulted in a large Game class that handled a lot of disparate behaviour and had too many responsibilities. To address this, the logic was split into smaller classes responsible for different screens, in order to simplify the logic and improve the modularity.

In addition, the LevelScreen class contained logic related to resizing, instantiating game objects, and running the simulation. This meant that the level configuration, including all objects in the level, was connected to the logic of how to run a simulation with any assets.

As a result, it was not easy to create other scenes with different objects, such as different levels or simplified scenes for testing. To support this, an Engine class was created to contain the logic on how to run a game with certain GameObjects, but we could change the objects we put into it. The Engine class is used by LevelScreen as a service.

The codebase previously contained a large number of static variables in random places. This was problematic for expandability, as it made it difficult to create more than one object of a certain type. For example, a new level with new objects should be created rather than retaining the previous data. However, when using static variables, we must manually reset everything, which is undesirable. This was even worse for unit tests, as running the game multiple times across different unit tests would result in static data persisting. We refactored to remove the practice of static variables and passed references to relevant classes instead. The code got easier to understand, test, and expand, and we could now avoid problems related to data being shared in situations where we may not be expecting it, such as strange bugs.

In addition, we refactored to have a centralised Assets system which allowed us to keep file paths in one place for easier changes. As this system was also used in the previous project, further explanation is omitted. For assets loading animation, each enum used row, frames, duration and path. This helped keep the code clean, descriptive and readable. We used constants such as SPRITE_SIZE and WORLD_WIDTH. We also added getRenderLayer for all RenderableComponents, using an enum sorted from rendering top to bottom for convenience.

Relation to Requirements

From the earliest stages of development, the design of the game was shaped by key requirements identified during the elicitation process. The need for a short, single-player, event-driven game set in a university environment established the functional boundaries, while the non-functional requirements directly informed the system's architecture and its evolution.

Extensibility and Maintainability:

Our decision to use an inheritance-based architecture as a modular design was to ensure that in future development, new components and entities could easily be added without rewriting the pre-existing code. This corresponded to our requirement NFR_SCALABILITY

In our current code, core systems such as Player, TileMap, and GameTimer were each given their own class, containing all the methods and attributes they would need to run in the game, with references to other classes being added when needed, such as Engine or Game. Also, additional classes could be added if we ever wanted to expand our game.

This corresponded to our requirements NFR_MAINTAINABILITY

Event Management and Gameplay Interaction:

- The requirement for multiple event types necessitated an event-driven design early in the development process. The architecture incorporated an event system capable of handling generic game events, such as collisions and item pickups which allows further events to be introduced later without changing the event infrastructure. This allows us to meet our event-based requirements such as FR_EVENT_INTERACTION or FR_VISIBLE_HINDRANCES.

User Interface and Accessibility:

- The inclusion of the UI Component interface and a dedicated UI system addressed the requirement for a clear, consistent, and accessible interface

- By isolating UI logic from gameplay components, the team ensured that menus, timers, score counters, and tutorials could be developed and iterated independently, supporting both ease of use and accessibility goals. This allowed us to meet our requirements FR_USER_INTERFACE, NFR_ACCESSIBILITY, NFR_USABILITY

Performance, Reliability, and Portability:

- The streamlined update process helps meet the requirement for maintaining stable performance on standard desktop systems and ensures the game would remain stable and crash-resistant through isolated error handling in each system, allowing us to meet our requirements NFR_PERFORMANCE and NFR_RELIABILITY

- Lightweight rendering also ensured portability across desktop operating systems, allowing us to meet our requirement NFR_PORTABILITY

Scoring:

- The modular component approach allows scoring to be calculated during gameplay based on time and events, and supports the final result being shown on the ending screen. This allowed us to meet our requirement FR_SCORING_SYSTEM