Architecture

Group 14

Tecch Titans:

Bradley Mitchell

Daniz Hajizada

Ellie Gent

Joel Crann

Keela Ta

Leo Crawford

Lukas Angelidis

Diagrammatic Representations:

To create these diagrams, we have used PlantUML as it's easy to use and edit a UML diagram.

Structural

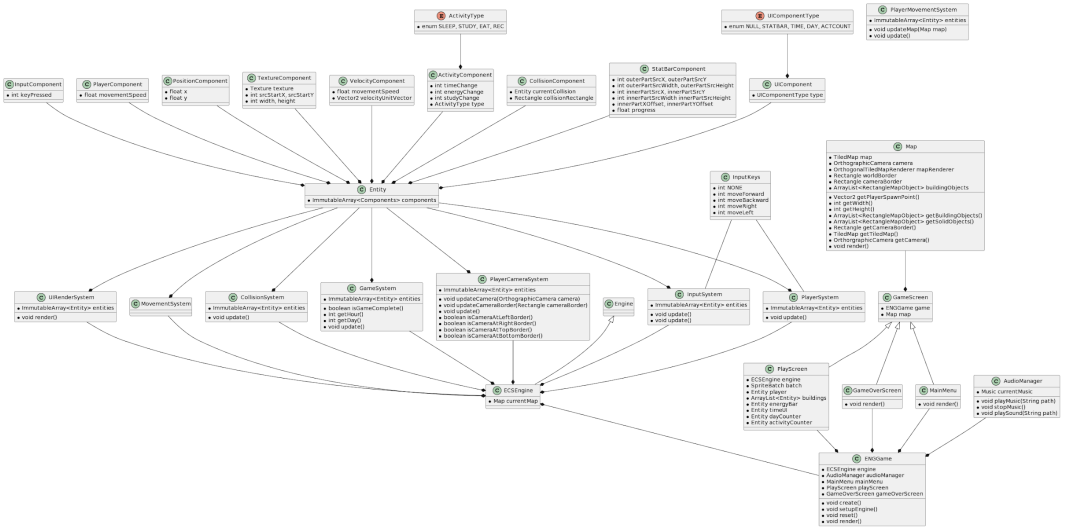
Message-Based Architecture *(Visible on the website https://danizhajizada.github.io/team14.github.io/)*

Basic architecture for a game involving a player that can move around a map, using messages to communicate input data with the display.

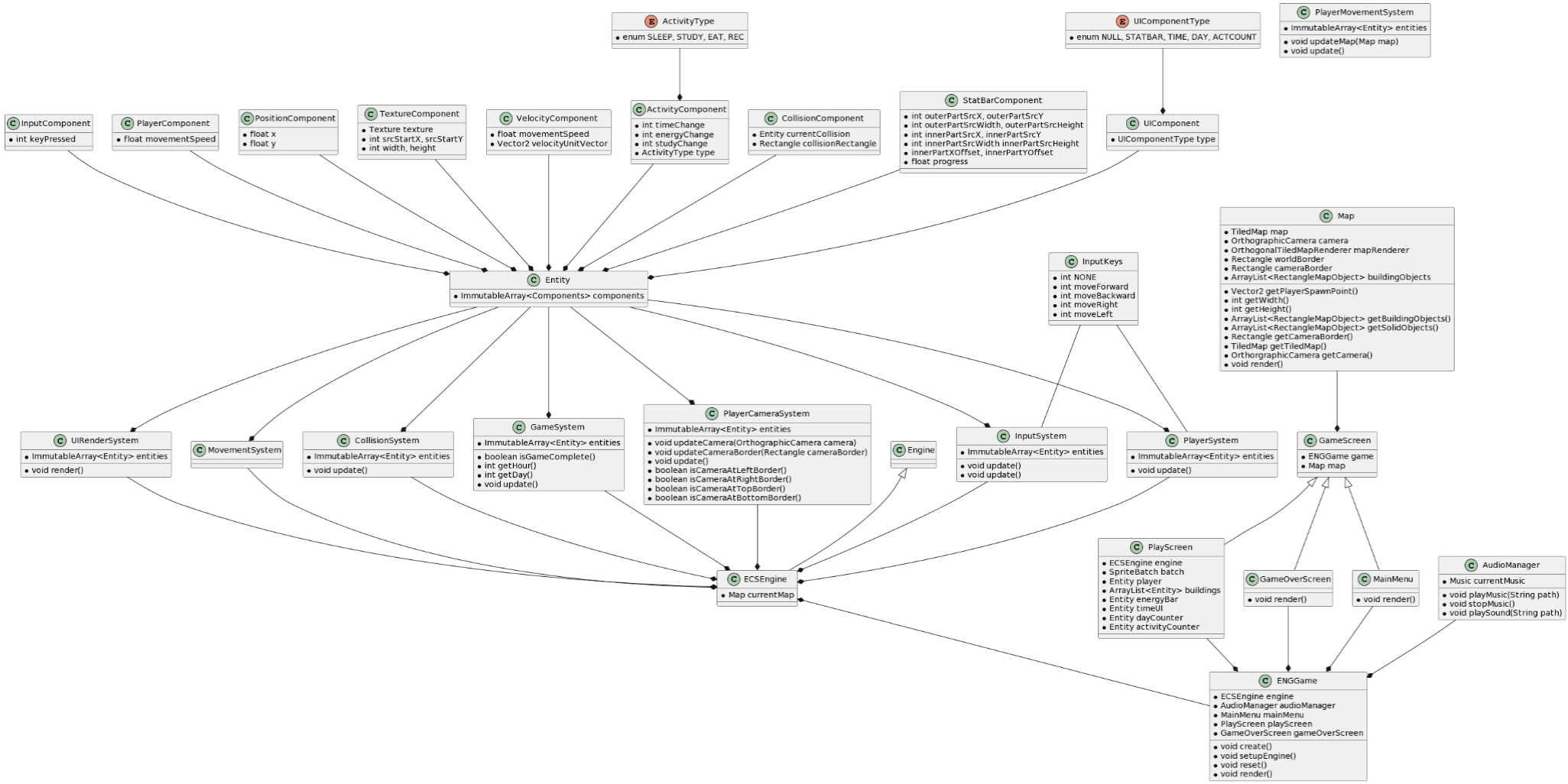
We've decided on using the LibGDX framework upon reflecting on this UML diagram, utilising a message based architecture to communicate inputs with a game screen, allowing the user to interact with the program as they desire. This will help us satisfy the user requirement [UR_USER_EXPERIENCE].

Entity-Component System

This is the finalised Entity-Component System (ECS) diagram of our project. The previously implemented version of it can be found on our website.



(Larger view next page)

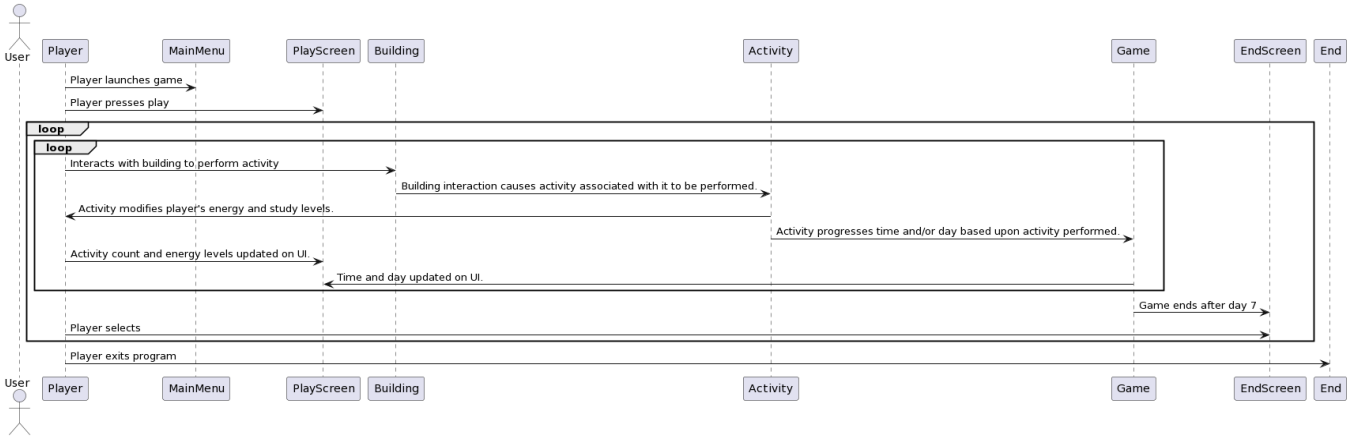

ActivityType
• enum SLEEP, STUDY, EAT, REC
UIComponentType
• enum NULL, STATBAR, TIME, DAY, ACTCOUNT
PlayerMovementSystem
• ImmutableArray<Entity> entities
• void updateMap(Map map)
• void update()
InputComponent
• int keyPressed
PlayerComponent
• float movementSpeed
PositionComponent
• float x
• float y
TextureComponent
• Texture texture
• int srcStartX, srcStartY
• int width, height
VelocityComponent
• float movementSpeed
• Vector2 velocityUnitVector
ActivityComponent
• int timeChange
• int energyChange
• int studyChange
• ActivityType type
CollisionComponent
• Entity currentCollision
• Rectangle collisionRectangle
StatBarComponent
• int outerPartSrcX, outerPartSrcY
• int outerPartSrcWidth, outerPartSrcHeight
• int innerPartSrcX, innerPartSrcY
• int innerPartSrcWidth innerPartSrcHeight
• innerPartXOffset, innerPartYOffset
• float progress
UIComponent
• UIComponentType type
Map
• TiledMap map
• OrthographicCamera camera
• OrthogonalTiledMapRenderer mapRenderer
• Rectangle worldBorder
• Rectangle cameraBorder
• ArrayList<RectangleMapObject> buildingObjects
• Vector2 getPlayerSpawnPoint()
• int getWidth()
• int getHeight()
• ArrayList<RectangleMapObject> getBuildingObjects()
• ArrayList<RectangleMapObject> getSolidObjects()
• Rectangle getCameraBorder()
• TiledMap getTiledMap()
• OrthorgraphicCamera getCamera()
• void render()
Entity
• ImmutableArray<Components> components
InputKeys
• int NONE
• int moveForward
• int moveBackward
• int moveRight
• int moveLeft
UIRenderSystem
• ImmutableArray<Entity> entities
• void render()
MovementSystem
CollisionSystem
• ImmutableArray<Entity> entities
• void update()
GameSystem
• ImmutableArray<Entity> entities
• boolean isGameComplete()
• int getHour()
• int getDay()
• void update()
PlayerCameraSystem
• ImmutableArray<Entity> entities
• void updateCamera(OrthographicCamera camera)
• void updateCameraBorder(Rectangle cameraBorder)
• void update()
• boolean isCameraAtLeftBorder()
• boolean isCameraAtRightBorder()
• boolean isCameraAtTopBorder()
• boolean isCameraAtBottomBorder()
Engine
InputSystem
• ImmutableArray<Entity> entities
• void update()
• void update()
PlayerSystem
• ImmutableArray<Entity> entities
• void update()
GameScreen
• ENGGame game
• Map map
PlayScreen
• ECSEngine engine
• SpriteBatch batch
• Entity player
• ArrayList<Entity> buildings
• Entity energyBar
• Entity timeUI
• Entity dayCounter
• Entity activityCounter
GameOverScreen
• void render()
MainMenu
• void render()
AudioManager
• Music currentMusic
• void playMusic(String path)
• void stopMusic()
• void playSound(String path)
ECSEngine
• Map currentMap
ENGGame
• ECSEngine engine
• AudioManager audioManager
• MainMenu mainMenu
• PlayScreen playScreen
• GameOverScreen gameOverScreen
• void create()
• void setupEngine()
• void reset()
• void render()

The reason we had decided upon using an Entity Component System architecture is because it allows a more flexible and modular design, allowing entities to easily reuse components to create new entities, giving us the ability to satisfy the user requirement [UR_AVATARS_ON_MAP] with relative ease. This architecture builds upon the previous architecture of being purely message based, with messages being sent between systems via the components which contain data related to the game. Each system can find particular entities with specified components, allowing the systems to interoperate while also retaining independence from each other, as they do not need to directly interact to function. This allows a player to make any choice of their own leading to their own individual outcome - [UR_ACTIVITIES].

One particular instance of this can be seen in the input system. When a player interacts with an interactable entity on the map, i.e. the piazza building, the player component's current activity is updated to the desired activity. The game system independently constantly checks the entities with a player component for an activity to perform. When one arises, the game system handles the appropriate logic for the activity, i.e. progress the day and reset the time and energy when sleeping. The systems remain independent while also sharing information in a message based way. Overall, ECS has led to a more manageable and versatile codebase.

Behavioural

Sequence Diagram

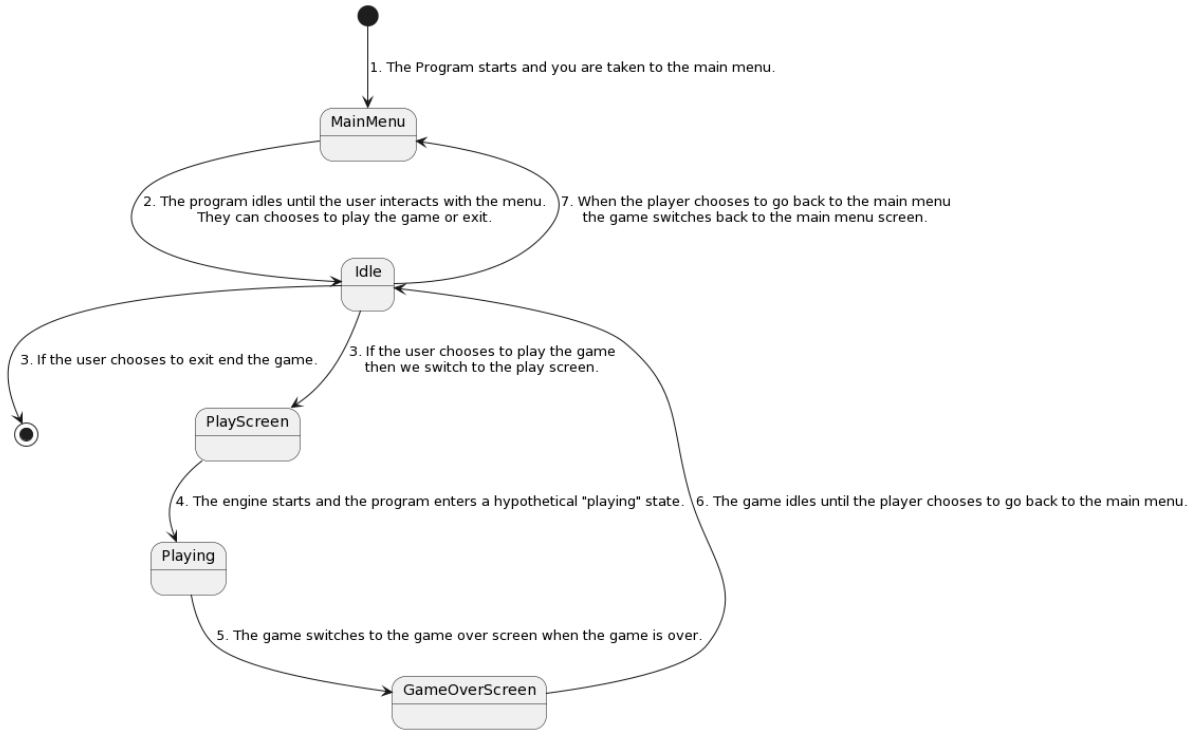


Using an UML Sequence diagram, we've illustrated the sequence of interactions a player can make, as well as the flow of events which could happen throughout the game. This

diagram includes components such as the player, the screen(s), objects and actions which interact with each other.

This behavioural diagram shows a loop which represents a quick and simple way of playing the game. Satisfying the [UR_GAME_PLAY] requirement, ensuring the game is easy to pick up and play.

State Diagram



This state diagram shows the transition between states and screens as the program runs and how it reacts to user input. User input is received and processed so that the user can choose what happens in the game, fulfilling the requirement [UR_USER_EXPERIENCE]. The transitions between states are quick and seamless, fulfilling the requirement [NFR_TIMING]. This also makes the game more engaging for people to use, as long loading times can decrease immersion.