Architecture

Cohort 3 Team 4

Kiran Kang

Hannah Rooke

Ben Slater

Abualhassan Alrady

Cassie Dalrymple

Charles MacLeod

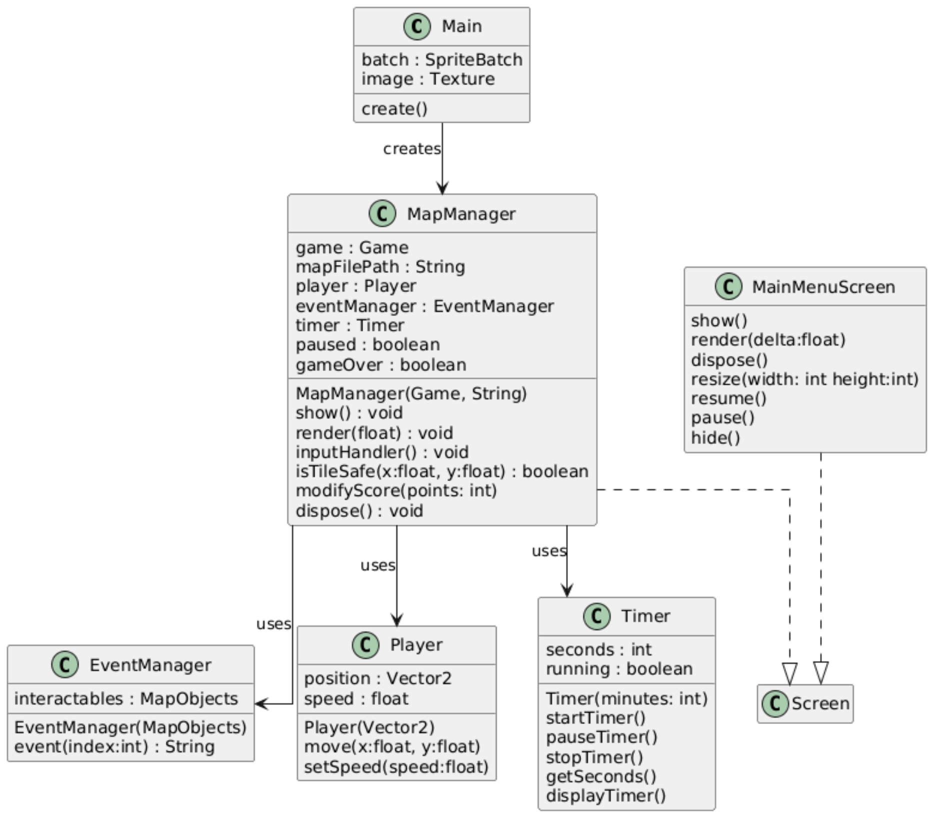
Dash Ratcliffe

Harley Donger

Introduction

Our team's 2D game is based on the user's requirements and is implemented using the LibGDX framework which is written in Java. IntelliJ will be used as our Java IDE. Development and version control will be managed through GitHub, this will manage branches, track changes and overall makes it easier to meet our goals.

Architecture (class diagram): Figure 1a - designed with plantUML



The class diagram above is the architecture of the game's design. In *PROCESS.md* github repository our method chosen was going to be an agile and scrum framework, where our main goal was to have a new playable piece of the game by the end of each meeting.We planned the game to be coded using an objected orientated paradigm approach.The diagram will allow relationships between classes to be shown through the implementation.

Figure 1a the **Main clas**s extends the **MapManager** class (also known as the **Game class** in LibGDX). The role of the Main class is to launch the application by initialising the assets and the main menu screen, which fulfils the relationship of *FR_MAIN_MENU*, as when the game launches the main menu will appear and transition to the maze map when playing. Moreover, the **MapManager** class is used to bring all of the classes together, it holds the player input, rendering and the transition between all game states. It does this by using the main components **Player**,**EventManager** and **Timer**. These components will operate simultaneously when the game is currently being played. MapManager and Player fulfils the *FR_PLAYER_MOVEMENT* since the MapManager would need to process the W,A,S,D inputs in the method *inputHandler()* and need to constantly update the player's position whilst checking for collisions using the method *isTilesafe()*.

In order for all of the function requirements in *REQUIREMENTS.md* to be fulfilled the **MapManage**r would need to include a maze map, a user controlled Player sprite, a not controlled Dean sprite, main menu, pause menu, a timer of 5 minutes and 3 types of events.

Implementing these components which ensure that the game will align with the customer's functional and non-functional requirements.

The **Screen class** brings all of the visual states together within the game. This fulfils the functional requirement *FR_UI_SCREENS*, as it maintains a clear structure on how the screens transition and interact with each other.

The use of the **Screen class** provides modularity for the team members who will be implementing screens within the game as they can choose to modify or replace different screens without affecting the rest of the code. The screen class also handles its own rendering meaning the memory usage will remain low, this fulfils the Non functional requirement *NFR_HARDWARE* as it makes sure there will be optimal performance across all standard PCs.

The **MainMenuScreen class** links very closely with the **Screen class**. The **MainMenuScreen** implements the Screen's interface in the game which allows the game to show different scenes. For example it can show the menu, game over, victory screen, pause and the map of the game all in one system. This fulfills the Functional requirement as it makes sure all visual components of the game are represented in a consistent medieval theme, where a medieval theme was necessary in REQUIREMENTS.md. It also fulfills the *FR_MAIN_MENU* functional requirement since the player would need to interact with the screens to play the game, like clicking play for the main menu to activate all of the game's classes or showing a victory or game over screen once the game is finished. (More info on this class in the Timer class section)

The **Timer class** keeps track of the time with two private attributes: seconds and running. Where seconds is the amount of time that is currently left ,which is then displayed in a format of minutes and running checks to see if the game is active or paused. The methods *startTimer()*, *pauseTimer()*, *stopTime()* and *displayTimer()* handle the different states of the gameplay's time. In the class diagram, the Timer is managed and implemented by the MapManger to make sure that the game is synchronised with the countdown.This fulfils the *FR_TIMER* because game starts by having a 5 minute countdown system so when the countdown reaches zero, the game must automatically stop.

The **Timer class** also links with the **MainMenuScreen class** as it controls the screens transitions based on the current time's state. Hence when the user inputs *'esc',* the pause menu will appear and the timer will stop at its current state, so seconds will remain the same and when resumed, the timer will continue to countdown until the game ends or if the user decided to pause the game again. Additionally, if the player completed the game within the 5 minute time limit, a victory screen will appear and when the timer is equal to zero then the game over screen will appear. This demonstrates that the Timer influences the screen interface changes depending on the current state of the Timer class and is needed to fulfill part of the *FR_MAIN_MENU* requirement.

The **Timer class** also aligns with the Agile Scrum methodology, meaning that it is easy to adjust to our team's weekly sprint iterations such as changing the timer's behaviour when the pause screen appears or just simply changing the design of the timer to make it have a more clean look.

The **Player class** is controlled by the user and represents the player sprite during gameplay. It's responsible for movement, interaction with events within the map. The main attributes include the position, which stores the player's current location in the map as a Vector and speed stores how fast the player moves along x and y directions. The setSpeed() method can update the speed during gameplay by increasing or decreasing the speed.

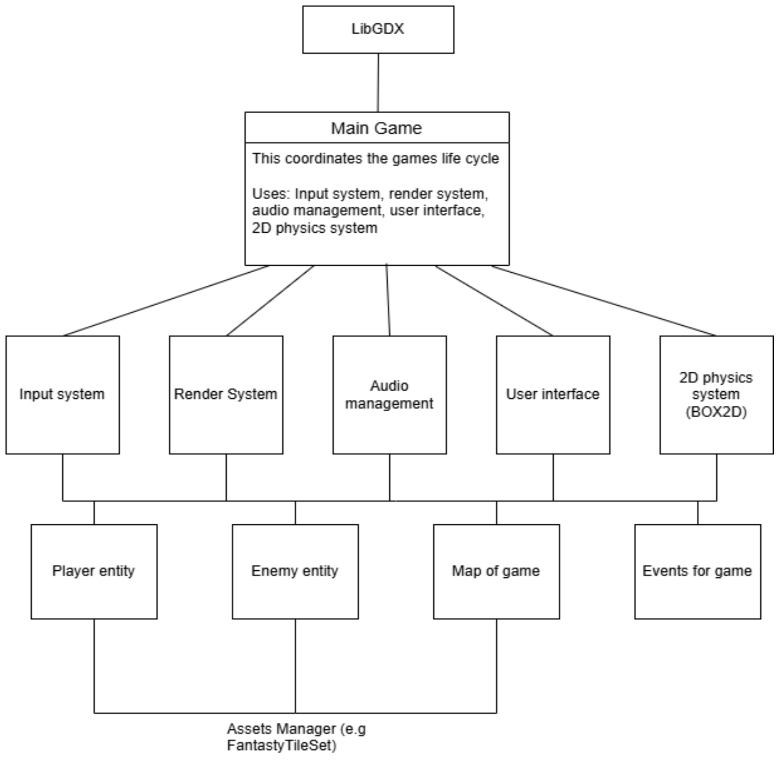
In the class diagram, the **Player class** is managed by the **MapManager class**, which handles the input, detecting collisions and rendering. This fulfils the functional requirement *FR_PLAYER_MOVEMENT* as the player is able to move up,down,left and right within the game's path, meaning it is restricted from passing through any map barriers or walls.

The **Player class** also links with the **EventManager class** to trigger events when moving around the map. This fulfils the functional requirement *FR_INTERACT_EVENTS*, for example if a player moves around the map and accidentally interacts with a hidden, negative or positive event, it will update the **EventManager class** to process the appropriate event. In addition to this, the Event() method should be able to count the amount of events the player has interacted with during a game session, which contributes to tracking the game's progress.

The **EventManager class** is in charge of handling all of the game's events that happen during gameplay. These events are sorted into three categories: positive, negative and hidden. Each event that has been completed, will be incremented by one in the event counter. This class holds the MapObjects, which is loaded from the tile map in LibGDX, and takes it as an input to pass into the constructor and stores them in the Interactables attribute The MapObjects holds the information such as the event type and what action is to be used.

The **EventManager class** is integrated in the **MapManager clas**s which is called whenever the Player sprite interacts with a tile or specific area. This fulfils the *FR_INTERACT_EVENTS* as it allows the player to discover positive, negative or hidden events during the game which could contribute to potential victory triggers, which could then also support the *FR_MAIN_MENU*, where the game will transition to a new victory or a game over screen depending on how many events were completed or if specific events were completed. This also supports the FR_SCORE as when the positive and negative events are triggered the score will increase or decrease.

<u>Structural architecture</u>

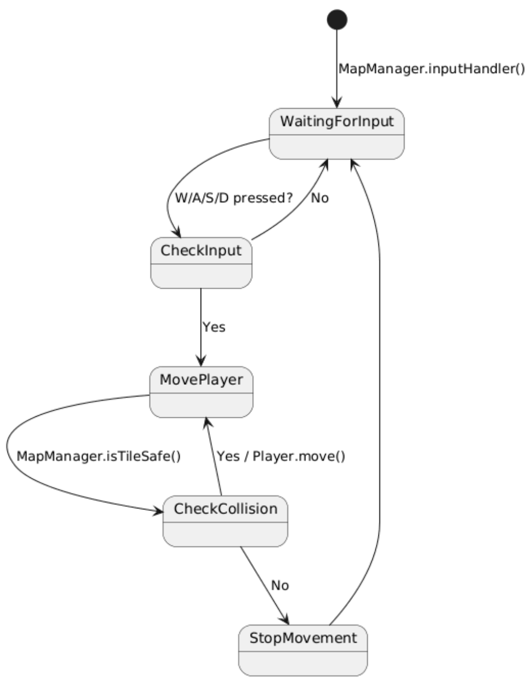

LibGDX
Main Game
This coordinates the games life cycle
Uses: Input system, render system, audio management, user interface, 2D physics system
Input system
Render System
Audio management
User interface
2D physics system (BOX2D)
Player entity
Enemy entity
Map of game
Events for game
Assets Manager (e.g FantastyTileSet)

MainGame diagram: (Figure 1b) - designed with draw.io
The structural architecture is needed to show how all the components in the game interact and depend on each other. IntelliJ provides the foundation for structuring all the code and makes sure that game behaviours like movement, events and timer are correctly implemented and worth simultaneously within the game engine, LibGDX. This structural architecture fulfils the *NFR_HARDWARE* non functional requirement as it has its own layers within the map, player and events to improve performance for the players so the game can run smoothly on start PCs.

Process of designing architecture

Iteration 1: Player Movements & Collisions Figure 2a - designed with plantUML

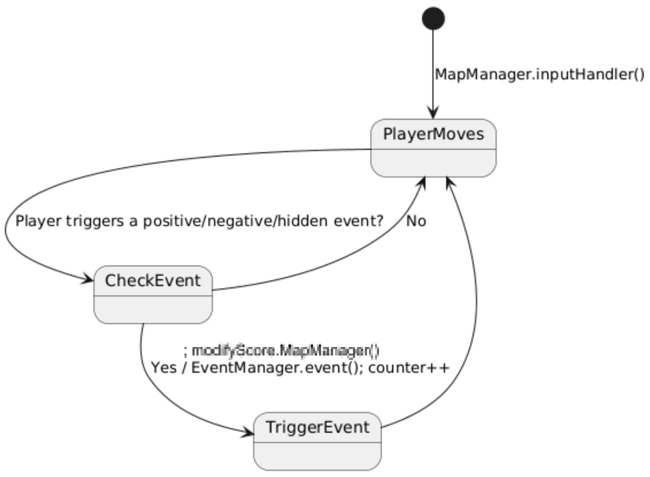
Figure 2a: Player Movement & Collisions



In Figure 2a, The need for player movement and collision flowchart is because in our GitHub's *REQUIREMENTS.md*, the team discussed additional ideas to fill in any missing requirements. One of these was accessibility and input methods. The movement of WASD is needed so the player can move smoothly throughout the game and the collisions are needed to prevent unrestricted roaming and make sure the player follows the game's path.This behaviour flowchart design demonstrates accessibility by providing easy movement that supports the game and enforces collision boundaries so the player will not get lost or get out of the game's map. This behaviour structure links with the structural architecture as it is implemented through the input system, player entity and physics systems components. This shows the consistency between the systems's design and the behavioural diagrams.

Iteration 2: Events + counter: (one negative, positive and hidden) - Figure 2b designed with plantUML

Figure 2b: Event Interaction & Counter



MapManager.inputHandler()
PlayerMoves
Player triggers a positive/negative/hidden event?
No
CheckEvent
; modifyScore.MapManager()
Yes / EventManager.event(); counter++
TriggerEvent

The need for Event interaction and Event Counter is because in order to complete the requirement *FR_INTERACT_EVENTS* it is necessary for the game to have a simple counter to keep track of how many times a player interacts with a negative, positive or a hidden event during gameplay. This behaviour flowchart demonstrates that whenever the player moves to an area where the event is located. The event will trigger the EventManger.event(), which updates the event counter by incrementing 1. Otherwise, the EventManager.event() will not be triggered nor incremented by 1. It will also trigger the modfiyScore.MapManager() which increases or decreases the score depending on which event was triggered.

Iteration 3 & 4: Timer of 5 minutes + menu mani interaction with pause,victory and game over screens Figure 2c and Figure 2d- designed with plantUML

Figure 2c: Timer of five minutes with victory/gameover screens

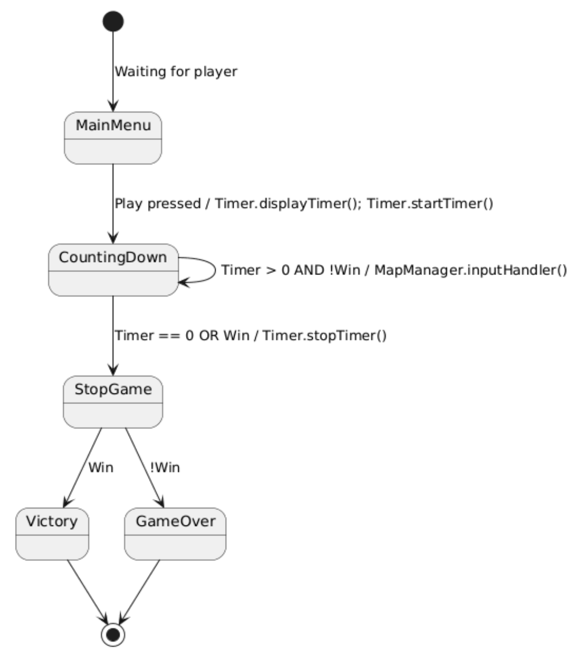
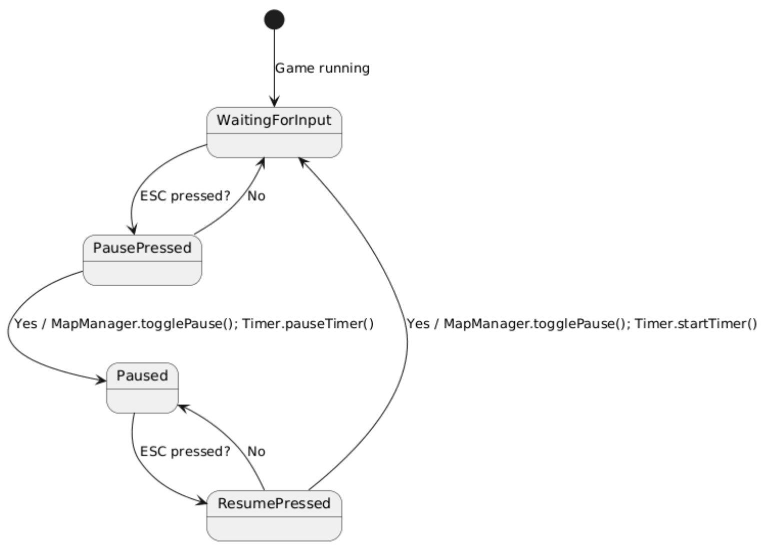

Waiting for player
MainMenu
Play pressed / Timer.displayTimer(); Timer.startTimer()
CountingDown
Timer > 0 AND !Win / MapManager.inputHandler()
Timer == 0 OR Win / Timer.stopTimer()
StopGame
Win
!Win
Victory
GameOver

Figure 2d: Timer Pause & Resume


Game running
WaitingForInput
ESC pressed?
No
PausePressed
Yes / MapManager.togglePause(); Timer.pauseTimer()
Yes / MapManager.togglePause(); Timer.startTimer()
Paused
ESC pressed?
No
ResumePressed

In figure 2c, a timer of 5 minutes with the victory/gameover screens is needed because it completes the *FR_TIMER* as it is necessary to enforce time constraints.This behaviour flowchart demonstrates that whenever **Timer** is updated it interacts with MapManager and will trigger the correct screen such as when the Timer reaches zero and Player has not completed the game, then a 'GameOver' screen will appear. Whereas, when the 'Win' condition has been met within the time limit then the 'Victory' screen will appear. In figure 2d, the Timer pause and resume is needed to implement the functional requirement *FR_MAIN_MENU* where it is required to have a pause screen by clicking 'esc', which stops the game and when resumed, the gameplay will continue so it meets the requirement.This behaviour flowchart demonstrates that the **MapManager** handles the pause input by using the togglePause() method and the Timer class calls the pauseTimer() when this happens so it does not affect the Timer's accuracy.

Traceability (table showing how architecture has met assessment 1 requirements)

Requirement ID	Requirement description	Architecture elements used	Reference to diagrams	Justification/Notes
FR_PLAYER_MOVEMENT	Player must be able to move up,down left, right and not past through solid objects	Input System, Physics System and Player Entity,MapManager	Figure 1a,1b, 2a	Input of W,A,S,D is used and then processed to update the player's movement in Physic system
FR_INTERACT_EVENTS	Player must interact with a negative, positive and hidden event. Each event interacted by the player adds 1 to the event counter	Player, EventManager, MapManager,	Figure 1a,2b	MapManager checks player's tile position to see if the Player has interacted with an event. Event() is then updated
FR_TIMER	Game will start when timer starts counting down, game will stop when timer is 0.	Timer,MainMenuScreen,MapManager, UI	Figure 1a,1b, 2c	Timer is controlled by the current game state in MapManager and updates the current Screen depending on the time.
FR_MAIN_MENU	Player will click start to play the game. Pause menu, when 'esc' is pressed the game has stopped. When resumed game will run normally. If player completes game within 5 minutes - victory screen will pop up. If timer is 0 'game over' screen will pop up.	UI, Input System,MainMenuScreen,Map Manager	Figure 1a,1b, 2c,2d	Initialises the Main Menu screen. MainMenuScreen shows the menu whilst MapManager handles the pause logic screen
FR_UI_SCREENS	The screens must visualise all components and the aesthetics of the game. Such as having a medieval theme and being able to transition to other screens smoothly, links to the FR_MAIN_MENU functional requirement.	UI, Render system,MapManager,MainMenuScreen	Figure 1a,1b, 2c,2d	All rendering is handled by the Render system which draws the medieval map player and events. MapManager links
FR_SCORE	Score will increase or decrease depending on if the player interacts with a positive/negative event. Links to the FR_INTERACT_EVENTS functional requirement.	MapManager	Figure 1a,2b	Score is dependent on how many times the user interacts with events. Score is controlled by MapManager class.
NFR_HARDWARE	Must run on standard PCs	All of Architecture system	Figure 1b	Layers the architecture and reduces memory usage so it can meet the performance on running on standard PCs