Arch1
Cohort 3 Group 5

Members:

Amity Van Rooyen
Cassian Kanhukamwe
Dhruv Madan
Gilda Grimes
Jerry Anish
Matt Ritchie
Oakley Fiddler
Ruby Brown

Architecture

Description

For the project, the architecture was used as a guideline for the software development team to refer to rather than a strict instruction, allowing for the development of the game to run smoothly, instead of trying to follow every component to the letter.

To represent and model the software architecture, Unified Modelling Language (UML) was used as it was the most widely used language and supported Java and object-oriented programming, allowing for the representation of inheritance. In order to document and display the UML architecture, PlantUML was utilised as it helped display a diagram of the UML code. All images can be found on the following website under the Arch1 pdf:

https://jsg551-cell.github.io/ENG1/

PlantUML was useful as it indicated what various classes interacted with or inherited from others. In the diagram shown in Figure 1, a white-headed arrow indicates that the class it is pointing from inherits from the class it is pointing to.

The architecture followed the criteria provided by the customer, i.e. 5 events to hinder or help the user, shown by a key required to open a door, or an NPC that may or may not be friendly.

The following section references the “Initial design” PNG.

As there were plans for having multiple interactable objects in the project, we decided to plan for the future by having the door class inherit from the Interactable class. This meant that going forward implementing further interactable objects would be more straightforward. As the game would be having a tile-based map, the tile class was made to represent each tile the player would be walking on. The plan was to have further classes inherit from this tile and to be stored in a 2D array to represent the map of the game.

The player would have its own class to store its location, with a parameter representing how many tiles the player can see. This was going to be used either with a fog of war system, or a zoom adjusted to the vision of the player. The player also had a parameter of an instantiation of the inventory class, allowing us to track what current items the player has. Putting the inventory as its own separate class allows us to implement NPCs having inventories in the future, if the need arises. Although the inventory class is quite simple, only having the functionality of a basic list, there is room for it to be expanded in the future.

The NPC class was created to allow us to create multiple NPCs for the player to interact with. The purpose of the interactable boolean parameter is so that some NPCs are able to be talked to in some sort of dialogue, whereas others wouldn’t have any interaction you could do with them, for example, if there was an enemy that chased you, making it interactable wouldn’t be the right decision. Furthermore, we have the hostile boolean for if the enemy would do damage to you upon contact.

Class: Interactable	
- Returns x,y coordinates	- Door is an *interactable*

- Have an action when interacted with - Returns if the player can interact with the interactable or not	

Class: Tile	
- Returns boolean value based on whether the player can walk on it or not - Returns x, y coordinates	No collaborators

Class: Player	
- Handles input of wasd keys to control movement - Returns x,y coordinates	- Player picks up *Item* - Player has an *Inventory* - Player can interact with *NPCS*

Class: Item	
- Item can be collected and added to player inventory - Player can use the item - Returns x,y coordinates	- *Door* requires a key which is a subclass of Item - *Player* picks up item

Class: Inventory	
- Add items - Use item - Get name of item	- *Player* has an inventory

Class: NPC	
- Interacts with player - Returns x,y coordinates	- *Player* can interact with NPC

Evolution and Adjustments

As mentioned in the previous section, not everything in the initial architecture's UML code was followed to the letter. This may have been due to disagreements with the original design, a problem with over-thinking a given feature or just for simplicity's sake, for example, the variables being of data type "double" were not needed and were changed to type "int". The reason for these simplifications is that the base game was intended to be a very barebones first implementation. Once the first final implementation of the game had been verified and tested, the UML was updated for documentation purposes.

The following section references the "Updated Item" PNG.

One part of the implementation that was changed was the way the Item class works. After changing the x and y coordinates to int, as previously mentioned, the next thing we did was to add a parameter with a list of possible names. This would be initialised with the names of the classes that inherit from the item class, so that when a new instantiation is created, we can ensure that the name provided for it is one of the possible names provided. There are two ways to initialise an item, one with coordinates and one without, as some items may not have a location decided for it on initialisation. We have 4 different classes inheriting from Item, which will be developed in the future: Key, FreshersFlu, EnergyDrink and Assignment.

Class: Item	
- Can be removed from map - Needs to be displayed on map - Return name	- *Key, FreshersFlu, EnergyDrink and Assignment* are subclasses

The following section references the “Updated NPC” PNG.

An additional part that was changed is the NPC class. The sprite string was replaced with name, for the name of the inheriting classes (“e.g. professor). The updateAI and interact methods were removed. updateAI was replaced by functionality in the main class, as it didn’t make sense to update the NPC from the class itself. However, much of the functionality that would have been included in this method was placed in the walkCycle method instead. Interact was removed as it didn’t make sense for the NPC to have a way to interact with the player, instead having the player interact with them or if the NPC is an enemy then have a check for the NPC’s proximity to the player and then run the code needed. As there were to be multiple classes inheriting from the NPC class, most methods are left blank, with only the getting and setting methods having functionality.

The professor class inherits from the NPC class and fulfils 2 of the obstacles needed, being able to block the door and needing to be interacted with to move, as well as slowly following you and making you stop for 3 seconds when caught. Professor overrides the walkCycle method to update the x and y values in an oval walk cycle, using Math.PI, based off the speed parameter input. It also uses the pathSize parameter, which just represents the length of the path taken during the walkCycle.

Class: NPC	
- Needs to be displayed on the map - X,y coordinates can be set - Check if its hostile or not - Hostility can be set - Can choose whether interactable or not - Check if its interactive or not	- *Professor* is a subclass

The following section references the “Updated Player” PNG

A number of parameters were removed and added to the player class. The vision parameter was removed altogether as we decided to just have everything visible on the screen, at least

in the prototype. 3 read-only variables have been added (type final in java), representing the size diameter and speed of the player. These have been made read-only as they are not meant to change whilst the program is running as it could cause many errors. We decided to represent the direction a player is moving using four boolean values. This way we can check these values every player update and adjust the players position based on that. We considered just representing these as four boolean values in an array but decided that four separate boolean variables, whilst longer to type, would be much easier going forward to work with as you wouldn't have to keep checking which index represents which direction. The update method which has been added is the method that runs this check, using the previously mentioned booleans as well as calling the movementCheck function, which ensures that the player won't move out of bounds. The majority of the rest are simple getter and setter functions, as to ensure that the other parameters can be easily edited but not by mistake outside of the class.

The connection between Player and Inventory represents that Inventory is a nested class within Player, so that it can only be accessed through the Player class. This is a form of encapsulation which we used as the inventory class is only useful in relation to the player class, and its functionality isn't useful to any other part of the program. It should be noted that using the final modifier on a list means that you can still add and remove items from it; it just prohibits you from creating another list with the same name. This prevents accidental reassignment of the list and signals that this name will always refer to the same list in the program. All the other functions in Inventory have practically the same functionality as the first draft, just including a way to see how many items the player has (using size), and a way to clear the list using clear.

Class: Player	
- Check inventory - Update the map - Displayed on map - Can move up, down, left, right - Return diameter of player for collision calculations - Check for collisions	- *Inventory* is a nested class within Player

The following section references the "Updated Main" PNG

The Main class has been added and has arrows pointing from it to Map and TileManager. Most of the attributes in this class are to initialise components of the software such as the map, player, time, etc. There are also attributes to set the screenWidth, screenHeight, maxScreenCol, and maxScreenRow to support the Swing library that we are using for the GUI. Swing is part of Java's standard library. All attributes other than the ones relating to the timer, tileManager and Professor have been set to read-only. This is to eliminate the risk of accidental changes that can occur during runtime. It creates a safer program. The attributes that are related to the timer cannot be read-only since we need to constantly update the timer and make sure it hasn't finished. We decided to do the UML for the main class last, since it needs to control the other components, which needed to be defined first. It is also the class that controls graphic,s and we needed more discussions with our customer on how the GUI should look.

Another addition is the Map class. It defines the parameters needed for Swing to generate the GUI. The isColliding method checks if the player is touching the borders of the map or not. There are also numerous functions that return the boundary and center values.

Class: Main	
- Has to listen to keyboard input - Checks if the player is in the hitbox of an interactable and whether the game is finished or not - Display map and components	- *Map and TileManager* are subclasses

Relation to requirements:

Requirement Number	**ID**	**Priority**
1	UR_CREDITS	When checkInteractions method from Main class declares that the game is ended, credits will be shown
2	UR_PERSPECTIVE	TileManager and Main classeswill display the map as a 2D/birdseye view
3	UR_MAP_LENGTH	The map is 16x10 tiles. Short enough for players to finish the game
4	UR_FINAL_GOAL	Within checkInteractions, there is a block of code that checks if the player has the key once they reach the end. If so, they finish the game. The interactables spread throughout the map make the game fun.
5	UR_SPEED_BOOST	EnergyDrink gives the player a speed boost. It is a subclass of Item
6	UR_IMMUNITY	Assignment protects the player from the professor. It is a subclass of Item
7	UR_TELEPORT	
8	UR_DOOR	The final exit is blocked without the key
9	UR_KEY	Key is a subclass of Item. T
10	UR_PROFESSOR1	These kinds of professors can be implemented as a subclass of NPC
11	UR_PROFESSOR2	Professor is initialised at the start and the checkInteractions method checks if the player has been caught or not
12	UR_FRESHERS_FLU	Freshers Flu is a subclass of Item

13	UR_WETFLOOR_SIGN	Wet floor sign is a subclass of Item
14	UR_DUCK	Can be implemented within the checkInteractions method
15	UR_LAKE_TOKEN	Player can interact with the lake and ducks. It is implemented within the checkInteractions method
16	UR_TIMER	Timer is initiated and controlled within the main function. Pause functionality can be implemented as its own method
17	UR_EVENT_COUNTER	The interactionCounter variable within checkInteractions method