Architecture

Cohort 3
Group 11
Team Aubergine
Joshua Wainwright, Piotr Koziol, Sarvesh Sridhar, Harrison Barrans, Daniel Thwaites, Callum Newton, Arnav Jamidar, Harry Turner

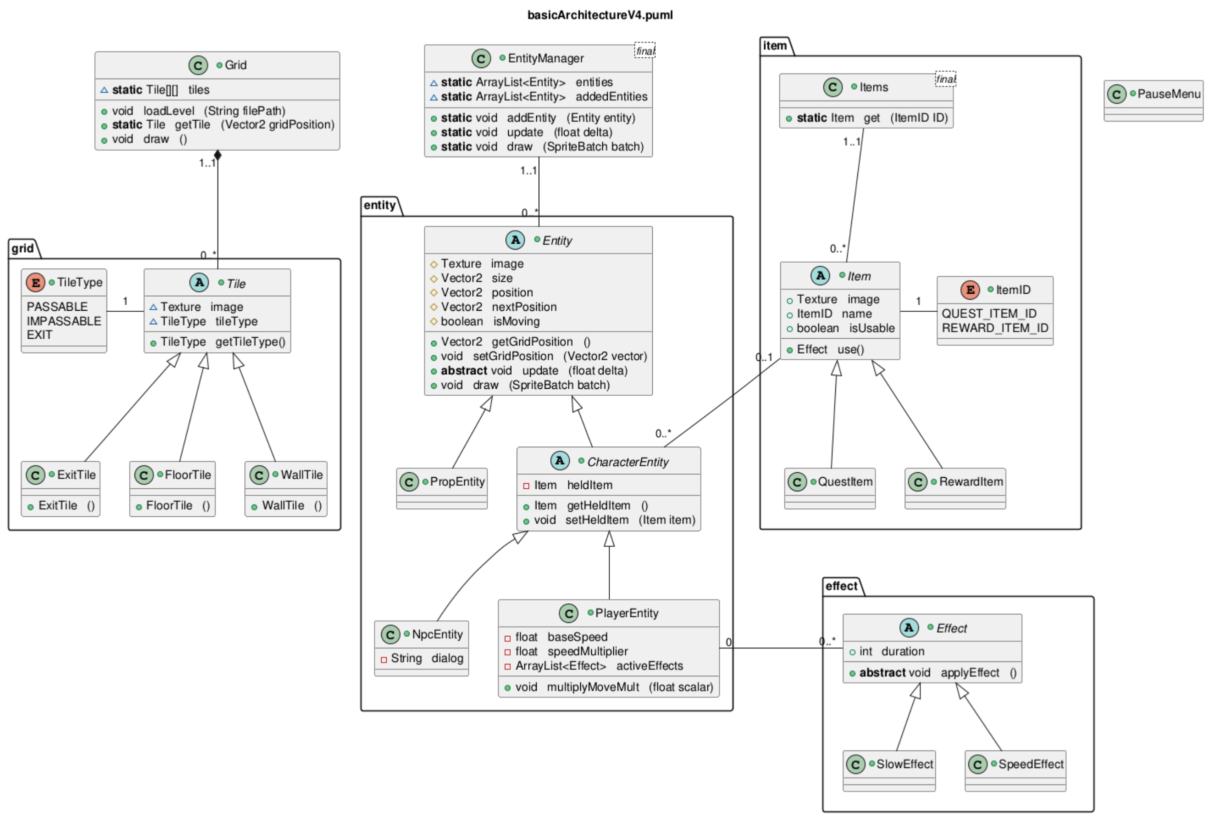
Methods and tools

To design the architecture of our system we will use several types of UML diagrams, particularly focusing on class diagrams and activity diagrams. When viewed as a whole, these diagrams will describe both how our code should be structured, and how it should behave.

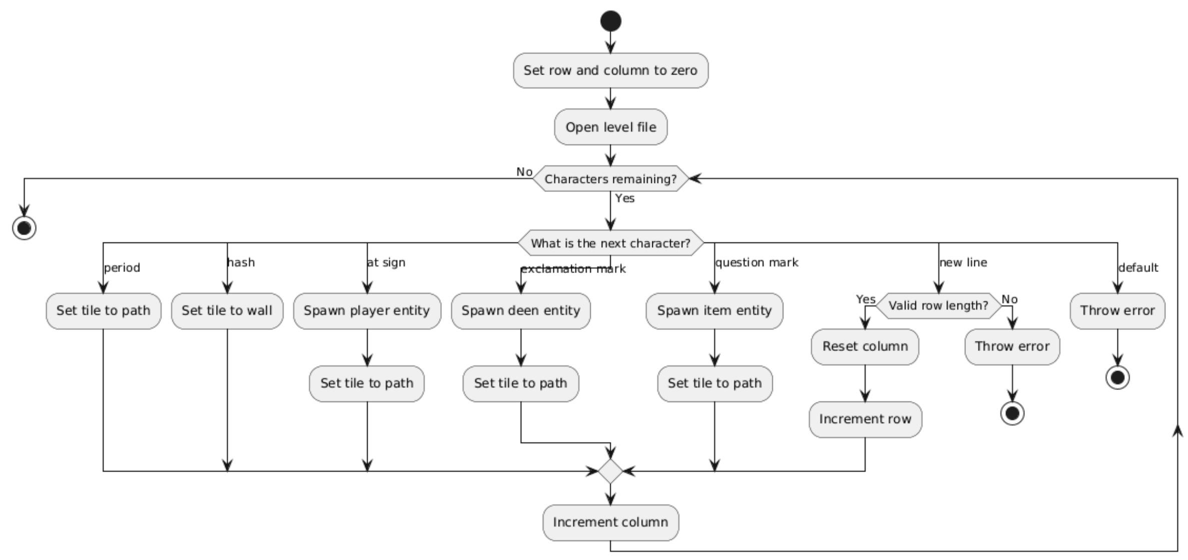
To create the diagrams more easily, we will use the PlantUML language to automatically generate them from text. This is beneficial because it means we can quickly iterate on ideas without spending a lot of time drawing. The text can also be version controlled alongside our code on GitHub, to keep a record of previous iterations.

Diagrams

Overall class diagram:



Algorithm for loading a level file:



<u>Current Architecture Overview</u>

The final architecture layout was created after many iterations in order to suit all functional, non functional, and user requirements to the maximum possible capacity. This has been accomplished through iterative development based on the requirements document created in the early stages of this project.

To begin with, the Grid class is responsible for loading the maze from existing data, rendering all tiles, and communicating which tile exists at any given coordinate. This is an isolated system, therefore changes to internal functionality will not affect any unrelated classes, therefore supporting NFR_SCALABLE_CODE. This is because external classes will interact with the grid solely through the members it provides instead of direct access to fields.

The Grid utilises tiles in a 2D array in order to convey the map layout. Interactions that occur when a specific coordinate on the grid is interacted with will be handled custom by the interacting object itself. This is because any object (mainly entities) that will interact with the grid will have to handle interaction with tiles uniquely per class. In order to assist with this however, the enum TileType has been created in order to define the properties present for each tile.

Next, the package “entity” is used to hold all entities. The content of this package is interfaced with through the class “EntityManager”. “EntityManager” is used to handle all active entities logic and rendering, accomplished through the “update” and “draw” methods respectively. These methods have been separated in order to facilitate the requirement FR_GAME_PAUSED, as the game should still be able to render whilst all simulation logic remains in stasis. Another result of this is that the camera will always be able to focus on the player entity, as its position will always be calculatable at all times, suiting the requirement FR_CAMERA. All members and fields within the EntityManager class are static, as only one entity manager will exist at any given time (because only one grid will ever exist at any given time). The entity manager holds 2 arrays for entities it is storing, one holding entities to be simulated and drawn, and the other acting as a buffer for inserting new entities to the prior entity array. The buffer array exists to ensure data is kept consistent throughout each entity's simulation logic, as otherwise entities may act on data that wasn't previously available for prior entities. This intuitive design ensures that logical errors between entities will not occur due to the order entities are stored in the entity manager (as the entity manager has an undefined priority order for interfacing with each entity).

The class EntityManager has been named as such to provide intuitive traceability, as its functionality is immediately evident, this being its functionality of managing the logic and rendering of entities within the game. This will therefore assist with debugging and developing the program.

Entities are objects used to interface with the grid and entity manager to provide content to the maze that can exist, move, and interact with grid tiles and other entities. All Entities are derived from the "Entity" class. This class provides methods to determine where a given entity exists or should exist on the grid, as well as the "update" and "draw" functions which purposefully share namesakes with the relevant EntityManager members to assist with intuitive design supporting code understandability (required by NFR_DOCUMENTATION). Multiple derivatives of the "Entity" class exist in order to facilitate specific functionality required by the customer. Derivatives such as "CharacterEntity" exist in order to provide groupings of entities with similar reusable methods and fields that would be specific to the entities in question. In the case of the "CharacterEntity" class, held items are required by all derivatives, therefore the 2 new methods "getHeldItem" and "setHeldItem" have been introduced. This will also provide the ability to override the draw method to display any held items the entity may have. This segmentation of code displays a compliance with the requirement NFR_SCALABLE_CODE because of this.

The Entity "PlayerEntity" is a child of "CharacterEntity" that is responsible for handling its own movement through user inputs. The logic required to accomplish this is unique to the player entity, implemented through overriding the "logic" method. This implementation completes the requirement FR_MOVEMENT. The method "getTile" provided by the grid will also provide the ability to view if the Entity has interacted with a tile with the attribute "Exit", therefore allowing the game to end (if all requirements are met). This therefore fulfils the FR_GAME_COMPLETION requirement.

To provide UR_REWARDS, the package "effect" has been designed. This package contains the main class "Effect", which is used to start a specified effect for a given duration of time. The result of the effect will be designed in a derivative class. This has been implemented in order for certain interactables to provide the player with unique results, adding relevant mechanics to the game. These effects are justifiably stored as objects, as their effects could apply to any applicable entity through them being added to the target's array of active effects. This will also allow for an effect to interact with the entity with unique results based on potential settings configured on a field before the effect was applied.

In order to facilitate the UR_EVENTS and UR_BONUS_OBJECTIVES requirement, the package “item” was created. This package provides items to the grid that entities are able to pick up and interact with. The class “Items” is used to store all items available in the maze game, obtained through their associated “ItemID” (a value stored as an enum). As every item placed on the grid will have an associated unique item ID, the “Items” class holds only static methods, ensuring that relevant items will be able to be accessible at all times throughout the code of the game without the risk of duplicate items existing with the same ID. The class “Item” is used to provide an interactable object that returns an effect on use.

Design Iteration 1

This project was initially designed to solely facilitate grid and entity functionality, as shown in this class uml diagram. To reflect this, the classes “Grid” and "Entity" were created. Because the maze game was set to exist on a singular map, the Grid class employed the use of static methods and variables to hold its data.

Entities were developed as a singular class that would hold their individual images and positions to uniquely identify each other. This quickly proved to be an insufficient implementation, as in order to compute logic for each individual entity, all varieties of logic would be implemented into the base entity class, with use determined by a switch case statement. Fields unique to individual entities were also required to be included within the Entity class, heavily bloating any objects with unused data. This did not conform to the requirement NFR_SCALABLE_CODE. Because of this, later iterations employed the use of polymorphism to separate logic between the different possible types of entity.

All created entities were also required to have their logic and rendering methods executed manually. This, again, was a massive issue as it heavily decreased scalability of the code, as any modifications to this process (e.g. adding a new entity) would require manual edits to all areas it had been implemented prior. To resolve this issue, the “EntityManager” class was introduced to handle the execution of all entities logic, as well as the storage of each.

Design Iteration 2

This iteration was created in order to expand the usability and intuitiveness of the Entity class, as shown in this class uml diagram.

Originally, the class “Entity” held no abstract methods, and instead existed as its own fully functional class. This was an issue largely specific to the “logic” method, as the developer would not know the default implemented logic used, therefore being unable to appropriately implement the class. To remedy this issue, the logic method was made abstract to force developers to inherit derived classes with defined logic methods, ensuring the developer knew how their entity would function by default (e.g. is it non moving, using a generative path finding algorithm, responding to user input, etc).

Derived classes of the Entity class are all suffixes with the word “Entity”. This is to ensure traceability throughout the code, as each entity's name will display its association to its original parent class. This will therefore assist with future debugging and assist with code documentation.

It was also noted that the Tile class could involve the usage of child extensions in order to increase the ease of implementing each tile onto the grid. This would allow for each derived tile to set their own dedicated texture and ID without forcing the grid to manually assign this for each tile. This would therefore segment the code further into more manageable and readable chunks, increasing the versatility and readability of the code.

Design Iteration 3

This iteration was created to provide an intuitive layout for the creation of unique entities with reusable functionality, as shown in this class uml diagram.

In order to further increase the reuse of code, The class “CharacterEntity” has been introduced. This class will be responsible for handling character specific interactions, such as holding and rendering items (that will be implemented in the next iteration). This class also provides the ability to add many custom attributes and methods specific to characters further on in development, increasing code scalability (NFR_SCALABLE_CODE). In the given example, CharacterEntity is further derived into “PlayerEntity” and “QuestGiverEntity”.

Final Design

This iteration was created to further organise classes, and to provide a framework for user requirements goals to be developed for, as shown in the class uml diagram present within this document.

One of the most noticeable additions present within this iteration is the usage of packages. A majority of classes present within the diagram has been put into an associated package in order to ensure organisation throughout future development.

Another edition is the introduction of the ExitTile class and its associated ID. This has been introduced in order to provide an endpoint in the maze that will trigger the completion of the game once all requirements have been satisfied. This addition accounts for the UR_MAIN_OBJECTIVE and FR_GAME_END requirements, as each states that an end point on the maze should exist in order for the game to finalise.

A major introduction to the architecture is the implementation of items. These items exist to be picked up and used by Character entities throughout the program, as described in the “Current Architecture Overview” segment. Similarly Effects have also been introduced in this iteration, and are also justified above.

The final introduction is the PauseMenu class. This has been introduced in this iteration in order to satisfy the FR_GAME_PAUSED requirements, as these classes will be responsible for displaying UI information on a user pause prompt at any point throughout the program's lifetime. This class is manually displayed as there is no other class used within the program that provides similar functionality. This prevents automated management (e.g. as seen in the EntityManager class), however the functionality offered by these classes do not necessitate such automation.