ARCHITECTURE REPORT

NP Studios – Team 8

Team Members
Lucy Ivatt, Jordan Spooner, Alasdair Pilmore-Bedford, Matthew Gilmore, Bruno Davies, Cassandra Lillystone

Concrete Architecture & Structure of Code

Languages & Tools Used

We chose to describe our architecture with UML (Unified Modelling Language). UML was chosen so that it was consistent with our abstract UML diagram, allowing us to compare them easily and identify any changes between the two diagrams. It was easy to understand and use when we created the abstract architecture, therefore we felt it was best to use it again rather than any other language.

We chose to use a class diagram because this diagram can be used in many stages of the software engineering process. It is used in the analysis phase to create an abstract representation of what we want the game to be like. It allows us to capture key features and ideas we initially come up with. It is also used in the implementation phase as a foundation for generating the code, to then be built upon. They are based very much on object orientation, which is a key concept for this project, making it very appropriate.

We have shown inheritance, composite relationships and associations between the classes in our UML diagram to ensure we completely capture the structure of the game and how each class and object works with each other.

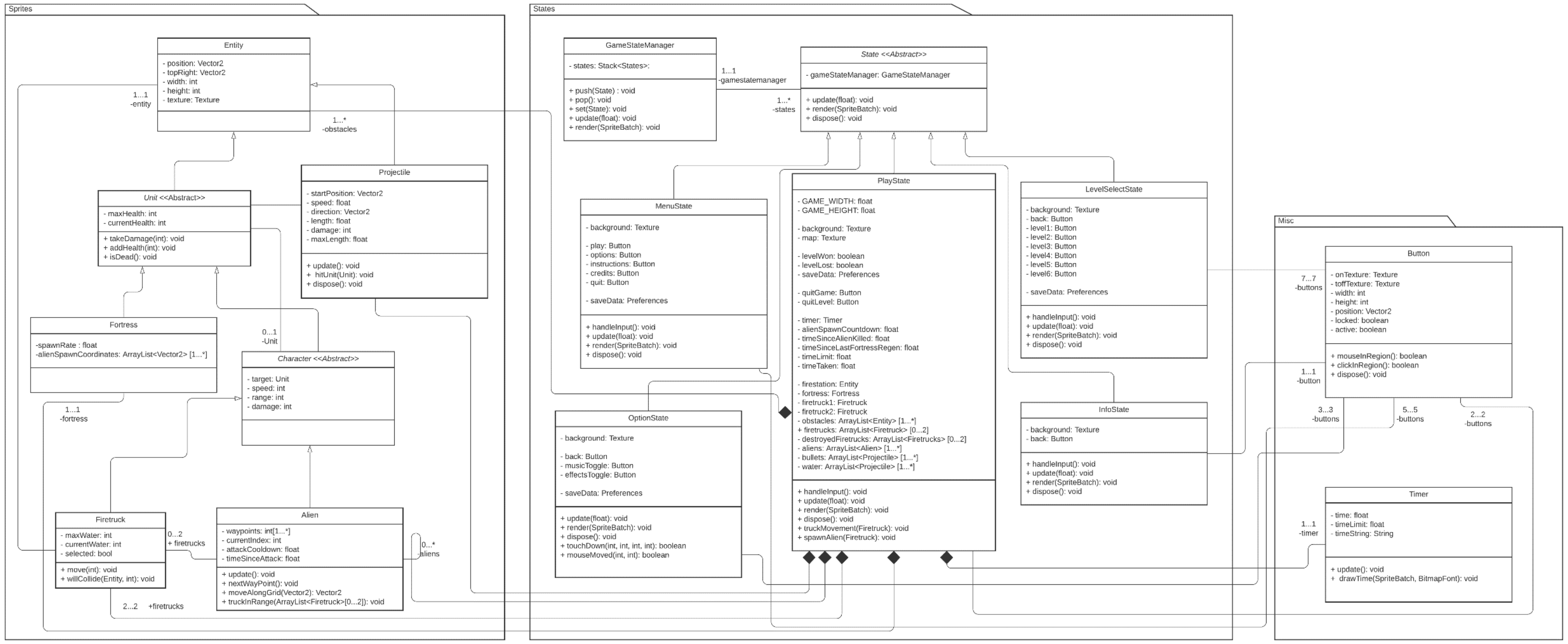
We used Lucidchart as a tool to create the concrete architecture. We decided to switch from draw.io to this because some of the team members said they had used Lucidchart in the past and found it easier to use than draw.io. Like draw.io, it can be integrated with Google Drive and Google Docs, which is our main tool for storing and putting together our documentation for the project. This makes it very suitable for us to use. As well as this, Lucidchart's interface is a lot easier to use and understand. When using draw.io we had a few issues with exporting the diagram and creating relationships between classes, however when we used Lucidchart we didn't have these issues.

Class Diagram – Website Link

We appreciate that having the diagram in this document makes it difficult to view, due to the size of it. If you would like to view it separately, it is on the website at:

https://npstudios.github.io/files/concrete_arch.png

Class Diagram - Image



Justification

The concrete architecture builds on the abstract class diagram by including new and modified classes, as well as showing the more complex detail within each class.

Class	Justification
Button	For the buttons on the State screens such as MenuState, LevelSelectState etc. Handles collision between the mouse and the button, plays its animation on hover and calls the function on a click. It's required for important actions such as starting the game. ***Related Requirements: UR_start_screen and UR_select_level***
Timer	Keeps track of the game time. Calls end-state if the time limit has been reached. Used to draw the time remaining to the screen and update it and to warn when the fire station is 15 seconds from being destroyed. ***Related Requirements: UR_attack_notification & FR_display_timer***
Projectile	Used for projectiles in the game (water drops and bullets). maxLength limits the range of projectiles so fire truck must be near fortress to attack. The damage attribute is inherited from the Character who created the projectile and calculates new health of its target. Projectile inherits Entity because the projectiles need Entity's attributes for them to appear on the screen. ***Related Requirements: UR_instruct_engines & FR_enemies_destroyed***
State	Abstract class. Defines methods *update()* (updates the game logic), *render()* (draws the updated elements to the screen) and *dispose()* (garbage collector for textures and sounds) which are implemented differently depending on the State. They have different implementations depending on the State, so we included each of these methods in the inherited Classes on our UML diagram. MenuState: Needed to control buttons and play background music. OptionState: Needed to control back button, music/sound effect toggle buttons and save their state in a preferences file as well as playing background music. InfoState: Used for the controls screen and credits screen. Controls back button. LevelSelectState: Shows the buttons for the levels and their correct colour. Loads the saveData file, checks if the user has completed level n (and sets button n to green) and then unlocks level n + 1 (and sets button n + 1 to blue).
PlayState	Controls game logic and calls required functions from the sprite and misc. packages when the appropriate conditions within PlayState have been reached.

Modifications to Originally Included Classes

We made some changes to the attributes in **Entity.** We created a Vector2 position attribute to store the Entity's position instead of using our original positionX and positionY values to make it simpler and more efficient. We also removed the active Boolean attribute as the way we implemented game States made this redundant. Two extras attributes we added were texture and Vector2 *topRight* (this is the top right coordinate that is calculated using width, height and position). The texture is what will be rendered to the screen (e.g. Firetruck image) and the *topRight* coordinate is used to check hitbox collision.

In the **Unit** class, we moved the range attribute into the Character class as we decided that fortresses don't need a range, they will instead spawn aliens at specified coordinates in the ArrayList *alienSpawnCoordinates*. We also added the *isDead()* method and the *addHealth(int)* method. The isDead() method is required so that we can call the end of game functions when the fortress is destroyed or all fire trucks are destroyed, as well as removing the aliens and trucks from the game screen when killed. This helps meet the requirement **FR_end_game** that says that the player wins if the fortress health is depleted or they lose if all the fire trucks are destroyed. We also wanted the fortress to be able to recover their health gradually, therefore requiring *addHealth(int).*

In the **Fortress** class, the *produceAlien()* method has been replaced with an ArrayList<Vector2> *alienPositions* containing the spawn coordinates. We instead put *produceAlien()* in the PlayState class as it was more intuitive to initialise the aliens here.

In the **Character** class, we removed the *destinationX*, *destinationY* and bearing attributes. Originally, the player would click a desired point on the map to move to, however, we have now decided we want the player to move the fire truck by using the WASD keys, partially fulfilling the requirement **UR_instruct_engines**. We decided it would be more engaging than using the mouse for both aiming and moving the fire trucks.

In the **Alien** class, the waypoints attribute was changed to type Vector2 to increase efficiency and ensure it remains consistent with the rest of the program. The update() method was added to call the code required to update the alien in each tick of the game in the PlayState. The *nextWayPoint()* method and the *moveAlongGrid(Vector2)* method are called in this update(), allowing the aliens to move up and down in the game (this replaced guardRoute() from our original UML). In the future these can be extended to allow the aliens to follow a patrol path. The truckInRange(ArrayList<Firetruck>) is used to check if any truck is within the range of the Alien so that they can attack them. (this replaced *seePlayer()* in our original UML). Additionally, we added the *attackCooldown* attribute and the *timeSinceAttack* attribute. These attributes are to control when the aliens can attack the fire trucks. We didn't want the aliens to be able to attack the fire trucks continuously, as this would make it near impossible for the player to dodge and complete the level.

The GameStateManager class was originally called Manager. The functionality of the methods originally mentioned in the abstract diagram have been moved to the PlayState class, occurring as part of the update function loop. We did this because the methods and attributes, such as *onWin* and *onLose* were about the game's logic, which is all handled in the PlayState class. This class initialises the game stack, has methods to *push(State), pop()* and *set(State)* the stack, updates the game logic for the State on top of the stack while also rendering the screen for the current state. We believe this was an intuitive model to understand, especially for future groups who may inherit our project. It also allowed us to easily create a pause menu as the OptionState is pushed onto the stack and then popped when the user goes back to the game, allowing them to continue their progress and not have to restart the level as the original PlayState is still on the stack.

New Classes

We originally thought we would only control damage through the Alien, Firetruck and Fortress classes. However, we decided that we needed a new class, called Projectile, to handle collisions between the bullet/water drop and the Character instances (done using the *hitUnit(Unit)* method), as well as calculating its distance as explained in the table above.

Button and timer were too specific implementations to include in the original UML. We need Button to check for collision between the mouse and the button and call the function / button animation. Timer is used to keep track of how long the level has been played for so that the *timeLimit* in PlayState can be implemented

Excluded from Original UML:

Our abstract architecture included a Pipe class for our minigame (which relates to **UR_minigame**), however assessment two didn't require this to be implemented, so we excluded it from the concrete architecture.