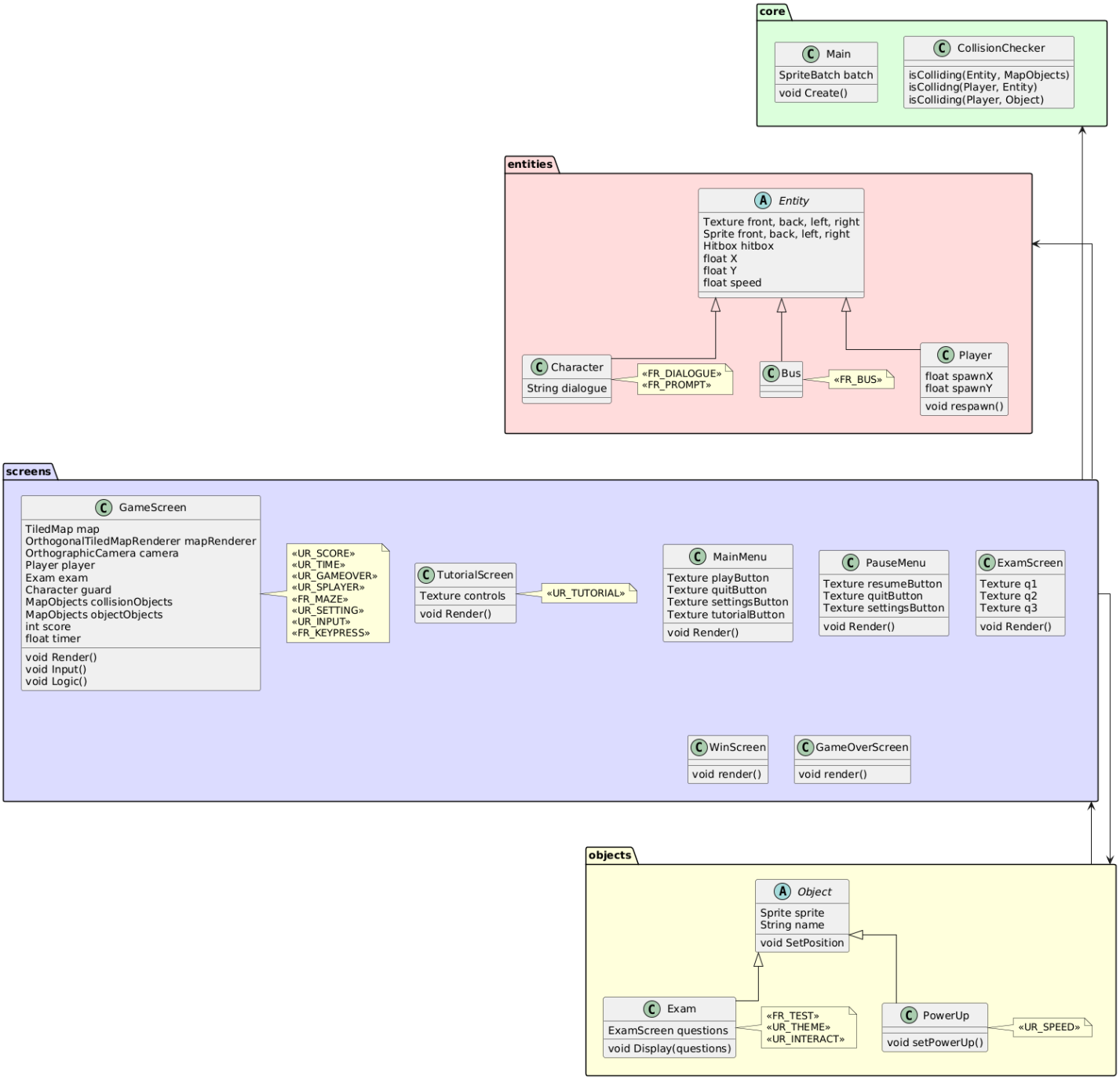
## Architecture

## Structural and Behavioural Diagrams, and CRC cards

The structural diagrams, as well as the use case diagrams, were designed using PlantUML. Our state diagrams were designed using LucidChart.
**Our behavioural diagrams can be found on our website under appendix D.**



Our CRC cards are displayed below:

| **Class: Entity** | |
|---|---|
| Responsibilities | Collaborators |
| • Load and store textures and sprites<br>• Track entity position<br>• Update sprite position during movement<br>• Create common movement and sprite accessors<br>• Dispose of textures when done | • Sprite<br>• Texture<br>• Player (subclass)<br>• Guard (subclass) |

| **Class: Player** | |
|---|---|
| Responsibilities | Collaborators |
| • Manage player spawn point<br>• Reset player position when respawning<br>• Update sprite position after movement<br>• Maintain sprite direction (front/back etc) | • Entity (superclass) |

| **Class: CollisionChecker** | |
|---|---|
| Responsibilities | Collaborators |
| • Detect collisions between entities and objects<br>• Build hitboxes for entities<br>• Determine overlaps<br>• Handle collision exceptions | • Entity<br>• Player<br>• Object |

| **Class: Main** | |
|---|---|
| Responsibilities | Collaborators |
| • Set and manage screens<br>• Provide accessors to game dimensions (width, height)<br>• Initialise and manage a SpriteBatch | • Game (LibGDX superclass)<br>• SpriteBatch<br>• MainMenu |

| **Class: Object** | |
|---|---|
| Responsibilities | Collaborators |

| **Class: Object** | |
|---|---|
| • Load and store texture and sprite for objects<br>• Store name identifier for map matching<br>• Position object based on map data<br>• Provide accessors to sprite and name<br>• Dispose of textures when done | • CollisionChecker<br>• Texture<br>• Sprite<br>• Exam (subclass) |

| **Class: Exam** | |
|---|---|
| Responsibilities | Collaborators |
| • Represent an exam object<br>• Track whether an exam has been completed<br>• Check and update completion<br>• Inherit sprite name and data | • Object (superclass) |

| **Class: GameScreen** | |
|---|---|
| Responsibilities | Collaborators |
| • Manage and render main gameplay screen<br>• Handle player input<br>• Control camera and viewport<br>• Update game logic (movement, collisions etc)<br>• Detect and handle collisions via CollisionChecker<br>• Render map, entities, and objects<br>• Track and display score and time<br>• Manage pause and resume | • Main<br>• Player<br>• Guard<br>• Exam<br>• CollisionChecker<br>• PauseScreen |

| **Class: MainMenu** | |
|---|---|
| Responsibilities | Collaborators |
| • Display menu screen<br>• Draw and update menu buttons<br>• Detect mouse hover and click<br>• Start new GameScreen on input<br>• Exit application on input<br>• Maintain button textures<br>• Dispose of textures when done | • Main<br>• GameScreen |

| **Class: PauseScreen** | |
|---|---|
| Responsibilities | Collaborators |
| ● Display pause menu overlay<br>● Draw and update menu buttons<br>● Detect mouse hover and click<br>● Return to GameScreen at current state on input<br>● Exit application on input<br>● Maintain button textures<br>● Dispose of textures when done | ● Main<br>● GameScreen |

| **Class: TutorialScreen** | |
|---|---|
| Responsibilities | Collaborators |
| ● Display tutorial instructions and control key images<br>● Detect mouse hover and click<br>● Return to MainMenu or GameScreen on input<br>● Manage viewport and scaling<br>● Dispose of textures when done | ● Main<br>● MainMenu<br>● GameScreen |

**Systematic Justification of the Architecture**

Each decision that we made towards the architecture was made to fulfill more of the requirements. To keep track of this, each class diagram is clearly labelled with the requirement that each component helps to achieve. This allowed us to keep track of our priorities, ensuring that components that were needed for requirements with “Shall” priority were designed first. The architecture supports a single player (UR_SPLAYER), in a maze designed in Tiled (UR_MAP and FR_MAZE), that can interact with exams and university themed characters (UR_THEME).

Our architecture is inspired by the Model-View-Controller (MVC) pattern. In this case, the views are the screens, the models are the entities and the objects, and the controllers are in the core package. Using this structure means that the code will be more maintainable, so it will be easier to alter aspects of the system or add new features without interfering with the other components. This design pattern also means that the implementation team can work in parallel to tackle different aspects of the system at the same time. This is because the separated nature of the MVC design pattern means that developing on one of the three parts of the pattern is unlikely to interfere with the other parts. For example, one team member can design the menu user interfaces whilst another is working on collision logic. This can greatly speed up the development cycle, as less conflicts will have to be resolved whilst parallel development of the architecture occurs. This is essential for this task given the short timeframe we have. The decoupling of each section of the architecture reduces dependencies and prevents code from becoming tangled. This means that the architecture has a low technical debt, making future refactoring much simpler. Furthermore, MVC supports multiple, modular views, meaning that adding new screens to the architecture is quick. This allowed us to fulfill FR_MENU, FR_SHOWTUTORIAL and FR_FINISHTUTORIAL.

We considered a monolithic architecture, however quickly rejected the idea as it would lead to tight coupling between elements of the architecture, meaning that our project could quickly become difficult to manage, refactor and up-scale.

Game entities and objects have been separated into separate packages in the architecture. This signifies and enforces the separation of the dynamic objects (entities) and static objects (objects). This means that interactions between dynamic and static objects can have rules. For example, an entity can interact with other entities as well as objects, however objects cannot interact with entities.

We decided to design all of the entities and objects as extensions of an abstract class. We designed one within the "entities" package and one within the “objects” package. This means that it is simple to set up new entities and objects, as essential attributes and methods are inherited from the parent class. The architecture is therefore scalable, with new models (entities and objects) being quick and easy to add.

We designed our collision detection system within the “core” package so that it can be abstracted from the implementation of the entities and objects that require collision logic. This means that each new entity and object can use the existing collision logic for their detections. The abstraction also means that if bugs were to occur due to the collision system, or the collision system needed refactoring, then the system would only need to be changed once to take effect on each collidable entity and object. This follows our architecture’s modular design.

## Architecture Design Process

### Initial Design - A.1

We decided to represent our architecture using class diagrams that adhere to the UML standard. This meant that each component of the diagram had a single clear definition. Initially, we designed the project with little knowledge of the LibGDX game library. So, we created a higher level class diagram to lay out the foundations of how objects relate to others and how they will interact with each other in the game. This consisted of objects such as Player, Character, Game and Maze. We linked each class to a requirement using their unique IDs so that we could track which ones have been fulfilled by the system. Using this system of design, we could focus on creating an initial architecture that allows us to fulfil each requirement once implemented. The initial design we came up with is shown in appendix A.1.

### First Iteration - A.2

Now that each requirement had been fulfilled, we iterated on this initial design to optimise the implementation process. We decided that the methods and attributes in a class should be given access restrictions to maximise encapsulation of important attributes such as textures, with getters and setters implemented where required. We decided to make use of the layering architecture pattern, as shown in the next architecture design iteration in appendix A.2. This meant that abstract versions of methods and attributes could be shared to multiple subclasses by a parent class, increasing the modularity of the architecture.

### Second Iteration - A.3

We then decided that we had a good enough architecture to begin implementing the game in LibGDX. This is because we had a solid base to begin the implementation of the system, but the architecture was still of a high enough level to allow flexibility around the limitations and requirements of the LibGDX library which we were not yet aware of. This meant we could begin a cycle of iteration, first attempting to implement the architecture that we had designed, learning the functionality of the LibGDX library along the way, and then updating the

architecture based upon what we learned. We found out that we do not need a Maze class for the map, as LibGDX has a TiledMap class that allows you to load maps created in tiled straight from the tmx file and then render them using methods within the class. This meant that we could remove the Maze class and replace the maze attribute with the built in tiled map loader and renderer. We also learnt that each part of the game should be separated into different screen classes, with a Main class holding the logic to decide which screen to display. Each entity is now defined as its own attribute in GameScreen so each one can be assigned its own type. We found out that LibGDX stores all code inside of packages. To begin the process of transferring over to using packages, we first created an architecture based fully within the “core” package, as displayed in appendix A.3. The “Game” and “Screen” classes methods and attributes are not displayed as they are LibGDX classes.

**Third Iteration - A.4**

With this re-evaluation of the architecture, we now had a window displaying a tiled map with menus. The implementation team then notified the architecture team that it was becoming difficult to navigate the codebase with many different classes being introduced. So, we proceeded to modify the architecture to separate related classes into their own packages, as can be seen in appendix A.4. This would help the implementation team create a more navigable codebase and promote encapsulation and modularisation.

**Fourth Iteration - A.5**

The implementation team then reported that the entity system was flawed. According to their research, it was difficult to implement collisions with the current system. This was due to the hitbox implementation not taking into account the hitboxes of the walls in the Tiled map. This meant that the entities' hitboxes had nothing to interact with. We decided that the best way to fix this was to introduce a new class to check for collisions, and to also store the objects placed in the tiled map in a variable. We decided to store this in the core package so that the collision checking can be imported wherever it is needed throughout the program with the required amount of encapsulation. We also decided to separate the entity architecture into two parts, an “entities” package and an “objects” package. This is because the logic required for each object was much less than the logic required for each entity, so separating them and simplifying the Object classes would make the codebase more easy to manage for the implementation team. This also optimised the architecture as each object holds less properties than each entity, meaning that the memory gains stack up as more objects are added to the game. This architecture helps our game satisfy NFR_LOAD and NFR_FRAMERATE. With the new collision system, we also moved the input handling into the GameScreen class to check for collisions when the user is controlling the character. This meant that the objects from the Tiled map could be kept private to the GameScreen class. We also added a Logic method to the GameScreen class to handle game logic related to collisions.

**Fifth Iteration - Final Architecture**

Finally, we altered the architecture to handle collisions between objects and entities. The implementation team reported that it was difficult to create the logic for the Exam classes being completed, as there was no way to locate what Exam the player was interacting with. We updated the design of the CollisionChecker class to contain a new method that checks for collisions between the player and objects in the “objects” layer of the Tiled map. The method returns the name of the object so that the Object class instance with a matching name can be marked as completed. We also moved the logic for adjusting the score, timer and reset into the logic method of the GameScreen class. The implementation team also requested for the architecture to define the respawn functionality, meaning that we had to add spawn point attributes to the design of the Player class. Also, displaying classes already implemented by LibGDX within the architecture diagrams was causing confusion within the implementation team, so we removed these from the class diagrams.