ARCHITECTURE

GROUP 5 - BITCRUSHED BOB

Maryam Mathews
Joseph Hinde
Jacob Mace
Will Aston
Zathia Jacquesson-Ahmad
Bulganchimeg Munkhjargal
Evan Weston

<u>Structural and Behavioural Diagrams</u>

<u>Languages and Tools Used:</u>

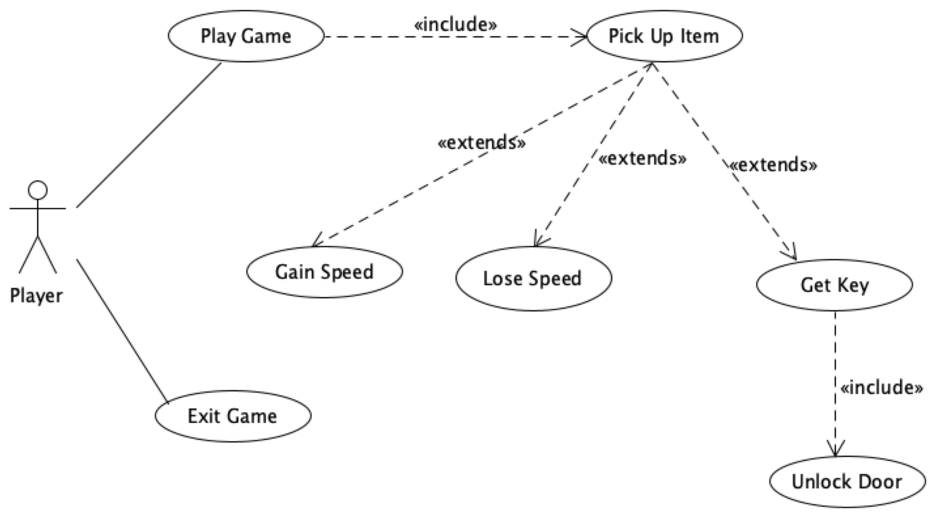
The architectural representations of the system were modelled using the Unified Modeling Language (UML). All diagrams were produced using UMLet, chosen for it's compatibility with standard UML notation.

- Structural aspects: Captured through a Class Diagram developed from a specification perspective, focusing on the system's interfaces and classes. This perspective was chosen as it shows how components of the system are structured and will interact, without the addition of implementation-level detail.

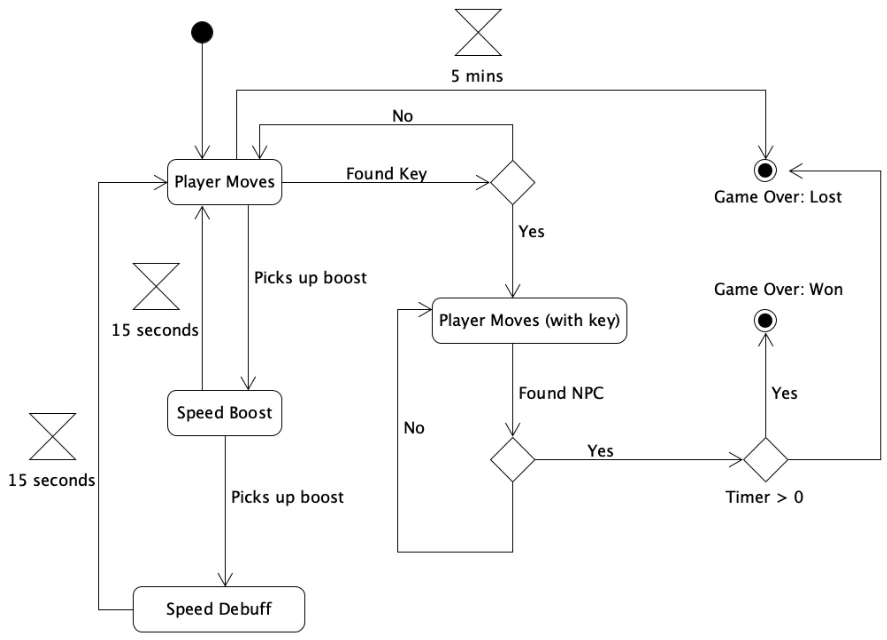
- Behavioural aspects: Illustrated using Use Case and Activity Diagrams, with a user perspective and workflow perspective respectively. These diagrams demonstrate how a user will interact with the system, and the internal flow of control within that scenario.

<u>Behavioural Diagrams:</u>

Use-Case Diagram:

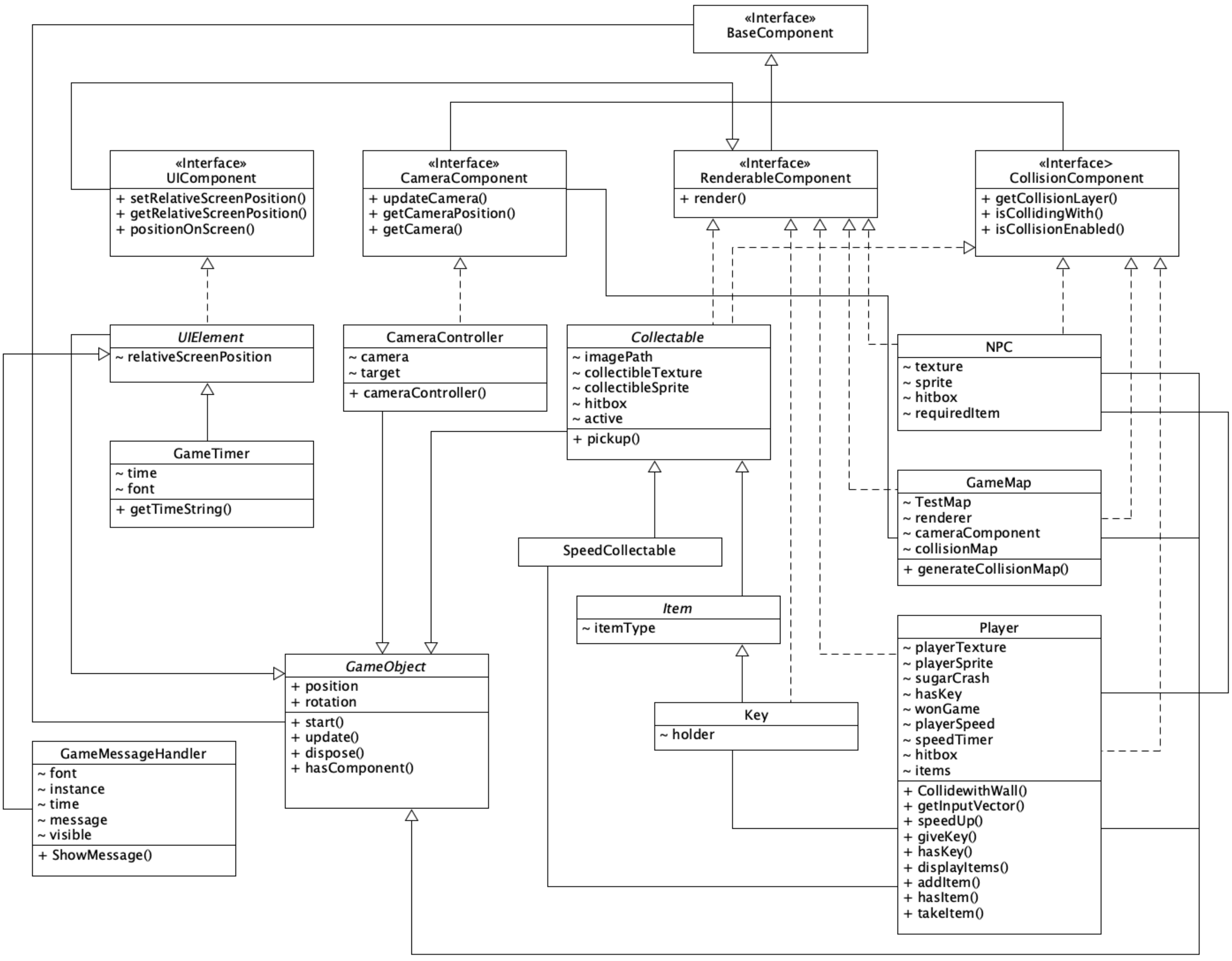


Activity Diagram:



Structural Diagrams:

Class Diagram:



Justification and Evolution of Architecture

Justification:

The project required a flexible and modular game design to handle multiple entity types with varying behaviours, including unusual features such as the speed boost mechanic, where using boosts too quickly results in temporary slowness. Traditional inheritance-based architecture would have created rigid structures, making it difficult to extend or modify functionality.

Therefore, the Entity-Component System (ECS) architecture was chosen, because it systematically addresses these requirements:

- Components as Modular Data Containers: Behaviours such as rendering and collision are encapsulated in components, enabling reuse and easy addition of new features.
- Entities as Identifiers: Each entity (player, key, NPC, etc.) serves as a unique identifier, allowing flexible composition without rigid class hierarchies.
- Maintainability: Isolated components reduce the risk of cascading bugs and makes debugging or updating individual components easier.
- Scalability: New components or mechanics can be added without affecting existing entities, allowing the game to expand easily beyond the first iteration.
- Alignment with Agile Development: Components can be developed and integrated incrementally, supporting parallel work and iterative development.

Overall, the ECS architecture provides a flexible, maintainable, and scalable framework - making it the most suitable choice to support this project.

Design Evolution:

The design process for the game followed an iterative and evolutionary approach, starting from a simple object-oriented structure and gradually moving toward a flexible, component based model as the project complexity increased.

Initial Design Phase

- At the beginning of development, the design was considerably more straightforward, with elements being hardcoded in the base controlling class first with the intention to refactor in the future.
- The main class had a reference to each individual game object and all game objects (eg. Player, Map, Key) extended a common GameObject class

Component-Based Composition Phase:

- The initial design worked for initial prototypes, but quickly led to difficulties extending functionality as more object types were introduced. So, to address this we shifted towards component-based composition.
- Shared behaviours, such as rendering, collision, and UI handling were extracted into interfaces. Entities then implemented only the interfaces relevant to their role, allowing greater flexibility and reuse.
- The main class was updated to have an array of game objects rather than individual references to further aid future expansion.

On the team's website (`https://escape-from-uni.github.io`) there are interim architectural diagrams, as well as CRC cards which provide evidence of the design process followed.

Relation to Requirements

From the earliest stages of development, the design of the game was shaped by key requirements identified during the elicitation process. The need for a short, single-player, event-driven game set in a university environment established the functional boundaries, while the non-functional requirements directly informed the system's architecture and it's evolution.

Extensibility and Maintainability:
Related Requirements: NFR_MAINTAINABILITY, NFR_SCALABILITY

- As the project was intended as an initial iteration of the game, and was to be handed over to another team, the system had to be easy to understand and extend.
- This directly led to the decision to use a component-based architecture as a modular design ensures that future developers can add new components (e.g. more events or level mechanics) without rewriting existing code.

Event Management and Gameplay Interaction:
Related Requirements: FR_EVENT_INTERACTION, FR_VISIBLE_HINDERANCES, FR_VISIBLE_BENEFITS, FR_HIDEN_EVENTS, NFR_SCALABILITY

- The requirement for multiple event-types necessitated an event-driven design early in development. The architecture incorporated an event system capable of handling generic game events, such as collisions and item pickups which allows further events to be introduced later without changing the event infrastructure

User Interface and Accessibility:
Related Requirements: FR_USER_INTERFACE, NFR_ACCESSIBILITY, NFR_USABILITY

- The inclusion of the UIComponent interface and a dedicated UI system addressed the requirement for a clear, consistent, and accessible interface
- By isolating UI logic from gameplay components, the team ensured that menus, timers, score counters, and tutorials could be developed and iterated independently, supporting both ease of use and accessibility goals

Performance, Reliability, and Portability:
Related Requirements: NFR_PERFORMANCE, NFR_RELIABILITY, NFR_PORTABILITY

- The ECS structure inherently supports efficient data processing, as systems iterate only over relevant component sets
- This streamlined update process helps meet the requirement for maintaining stable performance on standard desktop systems and ensures the game would remain stable and crash-resistant through isolated error handling in each system.
- Lightweight rendering also ensured portability across desktop operating systems

Game Flow and Functional Features:

Related Requirements: FR_MAP_STRUCTURE, FR_PLAYER_MOVEMENT, FR_TIMER_LIMIT, FR_EXIT_CONDITION, FR_SCORING_SYSTEM, FR_TUTORIAL_MENU, FR_MAIN_MENU

- Core systems such as Player, GameMap, and GameTimer were designed as entities composed of multiple components, supporting smooth player control, clear win/lose conditions, and reliable timing
- The modular component approach also allowed for features like pause/resume, scoring, and tutorial integration to be implemented cleanly

Theme, Content, and Cohesion:

Related Requirements: NFR_THEME_CONSISTENCY, NFR_CONTENT_CONSTRAINTS

- The use of isolated systems supports consistent theme and art direction, as each subsystem could be updated independently while maintaining stylistic coherence through internal review upon each addition