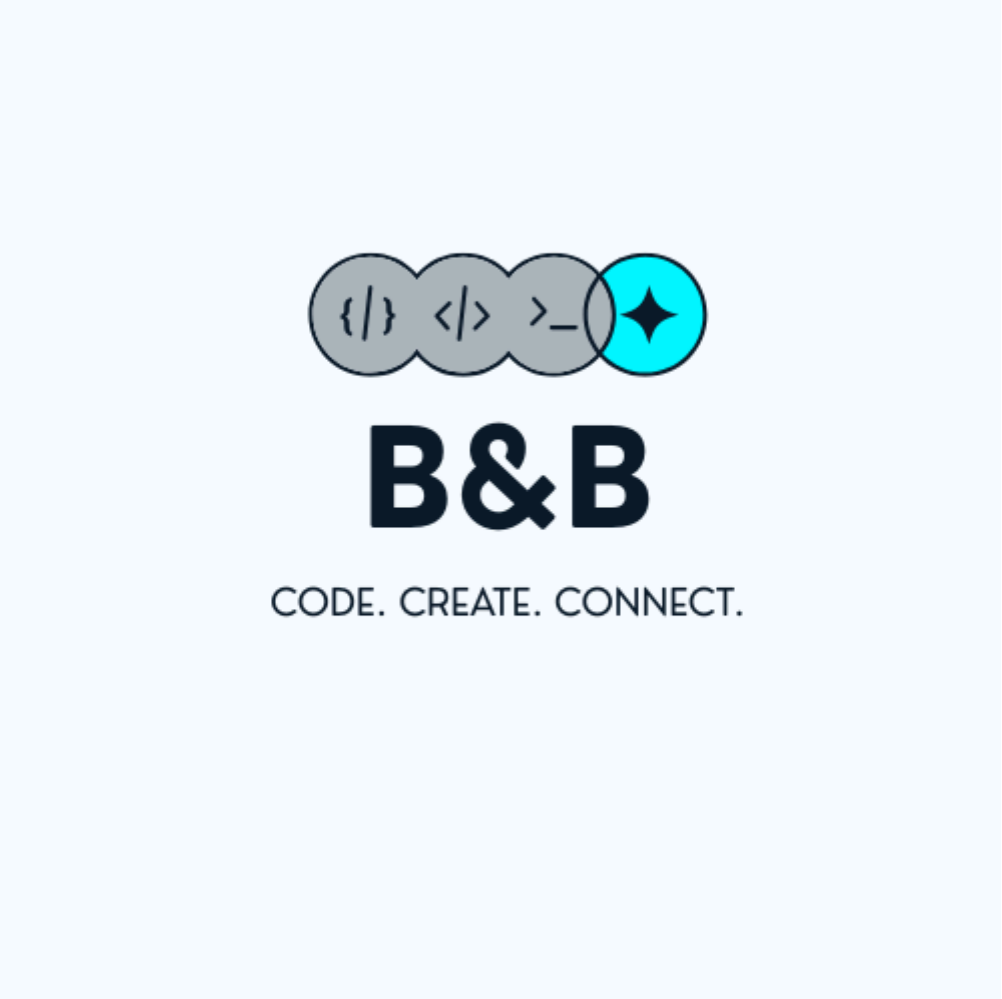
Team 8: Ben and Bensons



Team Members: Alyssa Skipper, Ben Senior, Benson Chow, Chloe Ward, Florian Mengkris, Hannah Thomas, James Ingram, Olivia Spencer

Diagrammatic Representation

Structural diagrams:

- **Figure 1:** Class diagram
- **Figure 2:** Component diagram
- **Figure 3:** Deployment diagram
- **Figure 4:** Package diagram
- **Figure 5:** Sequence diagram
- **Figure 6:** State diagram

Interim Diagrams can be found here.

Each UML diagram provides a specific architectural viewpoint:

- The Class diagram illustrated structural relationships and inheritance hierarchies.
- The Component diagram abstracts key modules such as GameManager, Screens, and Logic packages.
- The State diagram models gameplay transitions and system states.
- The Sequence diagram shows runtime interactions between user input, the player entity and game flow.
- The Deployment diagram depicts runtime execution on the JVM and its interaction with the LibGDX [2] framework.
- The Package Diagram represents modular organisation within the codebase.

Architecture Development

We used a UML class diagram to describe the architecture visually, created using PlantUML [1], a text-based modelling language that promotes precision and reduces ambiguity. UML was chosen for its language-independence and tool-neutrality, making it well suited for representing LibGDX-based java systems.

The diagrams together communicate the system at multiple levels - structural, behavioural and deployment - ensuring comprehensive understanding of both the static and dynamic aspects of the architecture. Each diagram was iteratively refined as the project evolved. Early versions were generated after initial prototyping, while final versions reflected the implemented system.

Design Process

Our architectural design followed a Responsibility-Driven Design (RDD) approach. The process began by identifying responsibilities directly from our functional and user requirements before grouping them into logical classes and packages. Using collaborative tools such as PlantUML, we iteratively refined class responsibilities and relationships based on early prototypes and team feedback. Interim versions of the architecture diagrams are published to our team website to provide evidence of this iterative refinement process and transition from version 1 to the final implemented architecture.

Systematic Justification

The initial architecture for our game was developed using object-oriented programming principles, with an emphasis on inheritance, polymorphism and encapsulation to provide a modular and maintainable design.

Early development began with flow diagrams which helped categorise the different elements of the game into logical groups based on functionality and interaction. This conceptual modelling was then formalised using UML diagrams to visualise both the structural (class relationships) and behavioural (state and sequence) aspects of the system.

The design goal was to create a single-player top-down maze exploration game, themed around the University of York campus. The player must navigate from their starting point to Central Hall within a five-minute time limit, avoiding dead ends and negative events while optimising movements through positive and hidden events to escape before time expires.

The architecture had to satisfy core requirements such as:

- FR_MOVEMENT - the player must be able to navigate the maze smoothly.
- FR_COLLISIONS - movement must be restricted to walkable areas within map boundaries.
- FR_TIMER - the game must impose a time limit and end when it expires
- FR_MAIN_MENU / FR_PAUSE_MENU / FR_END_SCREEN / FR_INSTRUCTIONS_SCREEN - screens must manage transitions between gameplay states.
- UR_EASE / UR_GRAPHICS - the interface must be intuitive and visually consistent

These requirements directly influenced the use of screen-based architecture, where each game state is encapsulated as a separate class extending the shared Screen superclass. This decision aligns with LibGDX design conventions, facilitating clear separation of concerns, flexible transitions, and simplified lifecycle management.

Overall Architecture

At the highest level, the architecture follows a Model-View-Controller (MVC) style separation, achieved through composition and delegation rather than rigid framework. The GameManger class functions as the central controller, extending the LibGDX Application class and managing all screen transitions through setScreen(). The Main class simply launches the GameManager instance when the application starts. Each subclass within the screens package (TitleScreen, StatusScreen, PauseScreen, EndScreen, InstructionsScreen) encapsulates its own display logic, rendering and input handling, inheriting shared functionality from the ScreenBuilder abstract class, which centralises common camera setup, rendering utilities and input behaviours to ensure consistency and reduce supplication across all screens.

The UML diagrams (see Figure 1) illustrates the system's structural hierarchy:

- The Main class extends the LibGDX application adapter class, enabling GameManger to call setScreen() to switch between subclasses of Screen.
- Each of these subclasses overrides lifecycle methods (show(), render(), dispose()) to define its behaviour, utilising polymorphism for consistency.

The component relationships between the GameManager, Screens and Logic modules are summarised in Figure 2 (Component Diagram). The Package diagram supports this view, dividing the codebase into core, logic and screen packages - a structure that ensures scalability and facilitates testing and maintenance.

This architectural pattern offers the following advantages:

- Loose coupling: each screen can be modified independently.
- Extensibility: new screens or gameplay modes can be added with minimal impact on existing code
- Maintainability: logic and rendering are clearly separated by responsibility.

The core game logic is contained primarily within the StatusScreen class, which represents the active gameplay state. It composes several other classes:

- Player - handles movement and interactions
- Map - loads and renders the maze layout
- Camera - follows the player dynamically
- MovementHandler - processes user input and applies velocity
- CollisionHandler - restricts movement within the maze boundaries
- GameManager - coordinates higher-level game logic like state transitions.
- GameMusic - handling the game music.

Each of these classes demonstrates encapsulation by managing its own data and exposing only relevant methods to other objects. For instance, the MovementHandler interacts with the Player through well-defined accessors, ensuring that movement logic is contained and reusable.

Behavioural Structure

The UML State Diagram (Figure 6) models the high-level game flow:

- The player starts at the TitleScreen, with options to Play, View Instructions or Quit.
- Play transitions to the StatusScreen (active gameplay).
- Instructions lead to the InstructionsScreen.
- During gameplay, Pause transitions to the PauseScreen, which can resume the game or return to the main menu.
- Upon timeout or successful completion, the system transitions to the EndScreen, displaying win/lose status before allowing exit.

This state-driven design ensures that every interaction corresponds to a clear and predictable system state. The use of state-driven architecture aligns with LibGDX's lifecycle methods (show(), render(), dispose()), ensuring consistent transitions and resource management.

The UML Sequence Diagram (Figure 5) complements this by describing the dynamic behaviour of a typical gameplay session - from player input in the StatusScreen, through movement updates, to eventual transitions to the EndScreen once the win condition or timer is reached. This diagram clarifies how control flows between classes, especially the interaction between the Player, StatusScreen and Camera.

Technology Stack

The game is implemented in Java [3] using the LibGDX framework for rendering, input handling and lifecycle management, packaged as a desktop .exe application. Assets are loaded at runtime from the core resources folder, ensuring cross-platform compatibility and straightforward deployment.

The Deployment diagram (Figure 3) visualises this runtime configuration, depicting interactions between the Java Virtual Machine, LibGDX framework and local asset directories.

Justification of Design Decisions

1. Use of Object-oriented design and Polymorphism
   The hierarchy of screen subclasses (TitleScreen, PauseScreen, EndScreen, etc.) demonstrates polymorphism, allowing uniform lifecycle management, while enabling specialised behaviour. This simplified screen transitions, reduced code duplication and improved maintainability.
   For example:
   - All screens respond uniformly to show(), hide() and dispose() methods, simplifying lifecycle management.
   - The GameManager controller can switch screens without needing to know implementation details, satisfying UR_DOC and maintainability-related requirements.
   - The Screens only have access to the public methods in main, not exposing it to the core game play in GameManager.
2. Composition for Game Entities
   Rather than using deep inheritance for gameplay entities, composition was used to keep components modular.
   - The StatusScreen (gameplay state) composes Player, Map, Camera and handlers rather than inheriting from them.
   - This improves encapsulation, allows independent testing of each class and adheres to separation of concerns.
     It also directly supports FR_MOVEMENT and FR_COLLISIONS, as movement and physics logic can evolve independently from rendering.
3. Screen-base Architecture for State Management
   The use of discrete screen classes reflects a finite state machine approach (TitleScreen -> StatusScreen -> PauseScreen -> EndScreen).
   This clear mapping structure was chosen to fully meet FR_MAIN_MENU, FR_PAUSE_MENU, and FR_END_SCREEN, while improving traceability between requirements and implementation, alongside simplifying debugging.
4. Resource and Performance Management
   LibGDX's lifecycle and rendering pipeline were leveraged to ensure proper memory handling. Each screen's dispose() method deallocates textures and sounds once no longer needed, ensuring the game remains efficient. This satisfies non-functional requirements such as efficiency, performance and resource management, aligning with good software engineering practice for real-time applications.
5. Camera and Rendering Separation
   The introduction of a dedicated Camera class provides abstraction overview handling. Each screen disposes of textures, sounds and assets when no longer needed, satisfying non-functional requirements such as efficiency, performance and resource management, aligning with good software engineering practice for real-time applications.

Architecture Development

During implementation, several adjustments were made to refine the initial design and improve maintainability. The original plan was to use a single Main controller managing all logic, but this was refactored into a dedicated GameManager class to align with LibGDX's lifecycle and improve state transitions. Handler classes such as MovementHandler and CollisionHandler were introduced to separate movement logic and collision detection from rendering. The ScreenBuilder abstract superclass was added to centralise shared camera and rendering setup, reducing code duplication across screens. These changes were reflected in updated UML diagrams on the team website, maintaining full traceability between architecture and implementation.

Evolution of the Design

Initially, the team intended to implement the project using a flat OOP structure, where classes such as MainMenu, Game and EndScreen were manually instantiated and controlled. However, as development progressed, this proved unwieldy and difficult to manage due to the complexity of screen transitions and resource handling.

During early iterations, the design was refactored into a screen-based state management system, adopting LibGDX's setScreen() approach.

This evolution resulted in:

- Cleaner separation between input, logic and rendering.
- Improved memory and lifecycle management.
- Reduced the need for global variables and manual resource tracking.
- Improved clarity and testability, satisfying UR_DOC and UR_MODULARITY.

Throughout development, UML diagrams were iteratively updated to reflect the evolving architecture. Early diagrams included only the MainMenu and GameScreen, but later versions incorporated the PauseScreen and EndScreen once these were implemented. This process ensured the documentation remained accurate and traceable to functional requirements.

Requirement Traceability

The architecture was systematically designed to meet the specified functional and user requirements as shown below:

Requirement ID	Requirement Description	Architectural Element(s)
FR_MOVEMENT	The player can move through the maze.	Player, MovementHandler, StatusScreen
FR_COLLISIONS	Movement restricted to walkable map areas.	CollisionHandler, Map, Camera
FR_TIMER	5-minute Countdown to complete the maze.	StatusScreen, GameManager
FR_MAIN_MENU	Start, Instructions and quit options.	TitleScreen. State transitions
FR_PAUSE_MENU	Pause/resume functionality	PauseScreen, state transitions
FR_END_SCREEN	Display win/lose.	EndScreen

FR_INSTRUCTIONS_SCREEN	Display gameplay instructions.	InstructionsScreen
UR_GRAPHICS	Clear, consistent UI and map visuals.	Camera, rendering logic
UR_KEYBIND	Straightforward controls and navigation.	Screen flow design
UR_DOC	Clear documentation and modular design.	UML diagrams, consistent naming

This clear mapping provides full traceability from requirements to implementation, as recommended in software engineering methodology.

Architectural Trade-offs

A key design trade-off involved balancing modularity and simplicity. Using discrete Screen subclasses and handler components introduced some communication overhead but significantly improved maintainability, scalability and clarity. This modular approach enables future features additions - such as saving, accessibility options or new map layouts - without restructuring the core architecture.

Conclusion

The final architecture of our game successfully integrates object-oriented design principles, screen-based state management,and UML-driven documentation to produce a maintainable, extendable and requirement-compliant system.

By combining inheritance and polymorphism in screen handling with composition in gameplay entities, the project achieves both conceptual clarity and functional robustness. The systematic evolution - from an initial OOP prototype to a refined state-driven architecture - demonstrates sound software design reasoning, directly fulfilling the expectations of the module's architecture criterion.

References

[1] Plant UML, "Plant UML at a Glance", 2024. Available: https://plantuml.com/
[2] LibGDX, "LibGDX Framework Documentation", 2024. Available: https://libgdx.com/
[3] Java Platform SE 17, "Java OpenJDK", OpenJDK, 2024. Available: OpenJDK