Architecture

Cohort 4 Group 6

Javengers

Braithwaite, Max
Faruque, Amber
Fu, Zhuoran
Kocaman, Melike
McDermott, John
Rissen, James
Scott, Charlotte

Architecture:

Abstract Architectural tools & notation:

For the architecture design, we used UML class and sequence diagrams, created with PlantUML, to create a layout for the architecture of the system as well as the entity and component interactions. These UML diagrams will follow a standard UML 2.5 Specification. All code was also completed using a libGDX framework and all project progress and code has been uploaded via our Group Github. All diagrams were created using google docs, Table Chart Maker for tables/CRC cards and PlantUML for sequence and class diagrams.
All interim architectural diagrams have been uploaded to our team website: ENG1 Project - Cohort 4 Team 6

Architectural style/decisions:

For the abstract architectural style of this system, we decided it was best to implement a layered architecture with an OOP(Object-Orientated-Programming) structure to fulfil the language requirement of Java as well as offering easy code structuring and planning early on. For the concrete architecture, we decided to use libGDX and various different parts of its framework to make it easier to structure the code(e.g. The libGDX Screen framework). Although due to this, we had to alter some of the normal style for the layered architecture, which is explained below (justification of abstract architecture).

Example of Architectural style (not all classes are included):

The domain layer manages the game state and rules, the presentation layer handles the user experience/ui and the infrastructure layer provides seamless communication and method calls between classes and layers.

PRESENTATION LAYER:
Components: GameScreen, MainMenuScreen, Lighting, HUD, etc.
Responsibilities: Render game world/menus, handle screen transitions and display UI/HUD
Dependencies: Bi-directional for infrastructure and can call domain layer
DOMAIN LAYER:
Components: Player, Goose,GameRules,Event tracking
Responsibilities: Enforce game events, manage entity behaviour and handle win/lose conditions
Dependencies: Pure game logic, little to no libGDX (infrastructure) dependencies
INFRASTRUCTURE:
Components: Main class, Asset loading, Input mapping
Responsibilities: Bridge between libGDX framework and game, managing resources
Dependencies: Can (may not need to) be used by both layers

Component evolution analysis

Following this, we attached a table showing the initial planned components, what they evolved into, and our rationale behind each of these changes.

#	Planned component	Implementation	Rationale for change
1	State Manager	LibGDX Screen interface for Screen-Based management*	Leveraging the frameworks built-in lifecycle management reduces custom code and maintains state separation (Simple program in need of less customization).
2	Resource/Asset Manager	Direct Asset Loading Assets loaded in constructors/main class	Simplified architecture for the small asset scale and the frameworks "garbage collection"/disposing is sufficient for program requirements
3	Input Handler	Distributed Input Processing Input handled per-screen	Reduces coupling* between unrelated input (in given contexts).
4	Core Game Loop	Main Class Extends "Game", handles window lifecycle*	Aligns with previously mentioned screen lifecycle of framework and avoids unnecessary code.
5	Game Screen	Implemented as planned	Core component stayed the same. Gameplay features changed during development.
6	Menu Screens	Improved on original UI*	Improving user experience and polishing original planned simple UI.
7	Map Manager	Implemented as planned*	No deep planning was given for this feature and was decided as options were discovered.

Behavioural Architecture

Our gameplay loop follows a structured event sequence where players progress through key locations while managing time pressure and dynamic positive, negative and neutral events (as shown below). The sequence diagrams shown on the Web site illustrate the core gameplay loop, all optional events, and win/lose conditions.

Play event sequence(Sequence diagrams):

This follows a framework handled lifecycle (screen transitions), where the users input with the presentation layer (menu screen) will trigger a state/screen transition which is managed via the infrastructure layer in main, and will then initialize the domain layer for the gamescreen entities/logic.

Movement sequence(Sequence diagrams):

Movement allows the infrastructure to capture user input, domain entities (player) to process the game logic and the presentation layer (game screen) handles output via visual feedback/movement of the sprite.

Player event sequence(Sequence diagrams):
This provides a clear example of how our architecture handles multiple interactions between layers whilst also maintaining a clear separation of concerns. The user input collected via the domain layer, triggers a screen/state change for the presentation layer (moving to the Ron Cooke hub) and the infrastructure layer providing communication and method calling across the entire chain of interactions for this sequence.

Iteration:

ClassV1 -> ClassV2 Changes and rationale(Class diagrams):
Initially, as shown in ClassV1, we had planned a traditional manager-based/parent architecture with most features having centralized controllers such as “StateManager”, “InputHandler” and “EventManager” inside of our infrastructure layer. Once we began further implementation, we discovered that the LibGDX framework we decided on naturally led towards a more distributed approach rather than centralized. One example of this, as mentioned below, is the “Screen system” which replaced our original “State Management”, allowing for our state lifecycle to be more easily managed and our input handling was naturally changed into a more distributed system along with this for each individual screen. This is shown in classV2. Therefore keeping all of the input code context relevant to the current state of the game and the screen the user is on.

LibGDX screen interface - Trade off - This provides less granular control than a custom state machine but due to the lack of complex state nesting, unpredictable transitions or undo operations/state history there would be no need for such granular control, therefore making the trade off a better fit for this project.

Distributed input handling - The input processing is distributed across different screen classes rather than the original idea to have a centralized handler (InputHandler) as shown below. This provides each screen with the specific input requirements, reduces coupling (mentioned in a later section) and allows for simpler debugging and performance improvements due to localised input issues and reducing input checks for inactive screens.

These changes moved us from centralized managers (infrastructure) to allow for a clearer and more easily visible layer separation. This is shown with our presentation of screens, domains containing entity logic and how they handle their own responsibilities. This aligns better with the layered architecture and the LibGDX framework/conventions.

ClassV2 -> ClassV3 Changes and rationale(Class diagrams):

The architectural stability between classV2 and classV3 validates the maturity and scalability of our architecture and is shown in the lack of structural changes required to integrate new features into existing layers. The established separation of layers

between presentation, domain and infrastructure provided a consistent and solid foundation to accommodate all new features added during the iteration.

We had the addition of AudioManager to the infrastructure layer which proved the scalability of our architecture and that new cross cutting concerns can be integrated to this layer without disrupting existing layers or code.

We also had the addition of new domain classes such as BuildingManager and Collectable, as well as screen classes (e.g. RonCookeScreen, LangwithScreen and InstructionsScreen). These confirm the effectiveness and scalability of our architecture and screen-based presentation layer while avoiding redesign or altering the architecture.

CRC Iteration(CRC Cards):

A complete requirements traceability matrix showing how each requirement maps to specific CRC cards is available on our team website: Mapping

As shown below, most of the iteration from our original CRC cards to our final ones suggested changes of:

- Distributed screen input handling
- Direct method calls for simpler communication of events
- Screen system handling state management

Planned CRC	**Final CRC/Method**	**Responsibility/method shift**
State manager	GameScreen + All Screens	State management is distributed for each context (event/screen)
InputHandler	Player + Screen Classes	Input processing is localised to users/screens
EventManager	Direct method calling	Simpler communication, no need for manager class
AssetManager	Constructor loading	Simplifies resource managing/loading
HUD	GameScreen.renderUI()	Integrated to gameplay screen

Project Modularity:

Architecture characteristics:

- Maintainability: Good - Layers allow for easy modification
- Performance: Good - FrameBuffer optimization for lighting system

- Testability: Moderate - Presentation layer requires LibGDX framework, other layers are testable

Cohesion:
- BuildingManager handles only transition logic - Medium cohesion
- Every screen class contains rendering logic for its state - High cohesion

Coupling:
- Afferent Coupling - GameScreen has high incoming connections
- Efferent Coupling - Main class has high outgoing dependencies (Expected)
- Connascence - Player/Goose share static connascence due to SpriteAnimation

Abstract architecture justification:

As mentioned above, OOP with a layered architecture was chosen as it works with the required language as well as allowing for easier structuring of code and abstraction for code modularity. Due to Java's class inheritance type structure of OOP, we used this to plan the original planned classes, subclasses, methods,etc. These changed throughout development and iteration but allowed for a much cleaner and more simple planning system without bloating or confusing our code structure.
We also implemented a layered structure due to the small team scale, in which layered architecture allows for easy parallel work such as splitting the presentation and domain layers into different tasks. This also aligned well with allowing for more comfortable allocation of tasks to team members who prefer front end in presentation, or logic in domain, etc. Layered architecture also provides a clear separation between framework code and game logic, as well as making it easier to learn libGDX since the layers split responsibilities. Layered architecture also provides a clear separation of concerns for a 2D game, as well as being easily maintainable for a student team project.
Overall, we used layered architecture as an overarching layout for how our system would be developed, while using OOP to structure inside of our individual layers to reduce clutter and keep code structured.

Concrete architecture justification:
As mentioned above, we decided to use LibGDX as our framework for concrete architecture due to its simplicity, deployability and pre-built frameworks. LibGDX's screen framework naturally suggests a layered system and allows for easier loading, managing and deposing of screens. This comes hand in hand with our layered architecture and allowed us to remove the need of a screen manager which we had originally planned. This allows us to take advantage of the LibGDX Screen interface which fits directly into the infrastructure layer of our planned architecture. It also provides a built-in screen lifecycle and also functions as a memory management system since it will automatically dispose and isolate unused or separate game states.
Another abstract architecture decision would be the bi-directional dependency between our presentation and infrastructure layers. Traditionally layered architecture would not use this approach, but since we are implementing libGDX's framework-controlled lifecycle, we adapted the layered architecture to this approach,

allowing for the bi-directional dependency. This is because the infrastructure layer (main class) owns the screen initialization and rendering/deposing loops, therefore calling the presentation layer. This is the rationale behind our change and the change still maintains separation of concerns as a typical layered architecture would but also respects the framework implemented for the game.