SEN3006 Project – Game with LibGDX

Group 1
Yavuzhan Özbek -2201927
Ozan Halis Demiralp -2203046

Main software problem of our project is building a 2D action game which has replayability value with its wave-based nature. It might look easy from outside; just some assets of a knight and bunch of goblins walking around on an open field, fighting each other. But game is a continuous real time software with many different mechanics and parts that must work smoothly and together. And it's not an easy thing to achieve.

One of the requirements is making different types of enemies spawn each wave. We also need NPC offering upgrades. Also code needs to be open to addition of new weapons, items and upgrades, since new ideas pop up as we keep developing. Another requirement is code must be readable so we can both understand what's going on.

In short, we are not making pixels fight each other. We are trying to engineer a complex real-time system that stays understandable and therefore extensible.

Key Challenges

1. Object Creation Volatility: Enemy, item and upgrades will change a lot of times during development. We cannot just instantiate classes with new keyword around.
2. Run-time Coupling: Knight, enemies, NPC and HUD of the game; all of these elements must react to each other's inputs and actions immediatly. Direct method calls would break the last principle of SOLID by creating a web of unwanted dependencies.
3. State Explosion: Game runs in various states like intro, combat, upgrade and game-over. We have a monitoring component that organizes all these.

Why a structured design is necessary?

We can answer this question with some examples:

Let's say we want to add a "Goblin Boss". Adding it in an amateur way means copy pasting hundreds of lines of Goblin code. It's terrible. We are implementing factory method in order to prevent this.

HUD of our game must poll both player' and enemies' hp each frame over and over again. Observer design helps with this.

Debugging who called *player.takeDamage()* is painful. Observer gives a single dispatch point for this.

As we add new functionalities and extend our code it becomes untestable. SOLID and patterns help this through isolating responsibilities for each class.

So basically structured approach helps with extensilibility, reusability, maintainability, relability and performance.

1. Factory Pattern
Where: GameEntityFactory.createGoblin(…), createPlayer(…), createNPC(…), etc.
Problem solved: Centralizes all object-construction logic (textures, animations, initial state) in one place.
Benefits
Single Responsibility: The rest of the code just asks the factory for "a Goblin" or "a Player," without knowing the main point of atlas slicing, initial HP or map references.
Easy Extension: To add a new enemy type, you implement one new factory method and no callers need to change.
Real-World Example: Car manufacturers build vehicles on demand, dealers never assemble parts by hand.

2. Observer Pattern

Where: EventBus.post(…), implements GameEventListener on WaveManager, NPC, CoreGame UI.
Problem solved: Loose-coupling between event producers (enemy death, wave countdown, upgrade selection) and consumers (UI overlays, camera shake, NPC dialogue).

Benefits

Low Coupling: Publishers don't need to know who's listening; listeners can be added or removed at runtime.
Broadcasting: A single event (for example WAVE_COUNTDOWN) can drive multiple reactions like HUD update, NPC pointer hiding without explicit wiring
Real-World Example: Newswire services deliver press releases to thousands of subscribers (news outlets, mobile apps) in parallel.

3. State Pattern

Where: Each enemy (Goblin, DynamiteGoblin, BarrelBomber) keeps an internal enum (MOVE, ATTACK, DIE or PREPARE, EXPLODE) and switches behavior based on it.
Problem solved: Clean separation of per-state logic (movement vs. attacking vs. death animation) without writing dozens of if…else trees.

Benefits

Clarity: Each switch(state) branch handles just one behavior.
Extendability: New states (for example Stunned or Fleeing) can be slotted in without touching unrelated code.
Real-World Example: A vending machine transitions cleanly through states
Idle -> AcceptingCoins ->Dispensing ->OutOfService.

4. Strategy Pattern

Where: Pathfinding is delegated to GridPathfinder (A* implementation) but each enemy "steers" by calling findPath() or "flee" by a simple vector inversion.
Problem solved: Encapsulates the pathfinding algorithm separately from enemy AI logic.

Benefits

Algorithm Swap-out: You could inject a different Pathfinder (for example flow fields or navmesh) without changing the Goblin code.
Testability: You can unit-test GridPathfinder in isolation.
Real-World Example: GPS apps let users choose "fastest route" "avoid tolls" or "shortest distance" each a different routing strategy.

5. Command Pattern

Where: Player input handling in the CoreGame loop and in Player.update() (dash, attack, movement) is vented through a consistent interface.
Problem solved: Decouples input polling from actual game-state mutations.

Benefits

Rebindable Controls: You could map "F" for dash or "Space" for attack at runtime.
Undo/Replay: Lays the groundwork for recording or undoing player actions if needed.
Real-World Example: A universal remote issues the same "volume up" command whether you press a physical button, speak a voice prompt, or tap a mobile app.

6. Singleton / Central Configuration

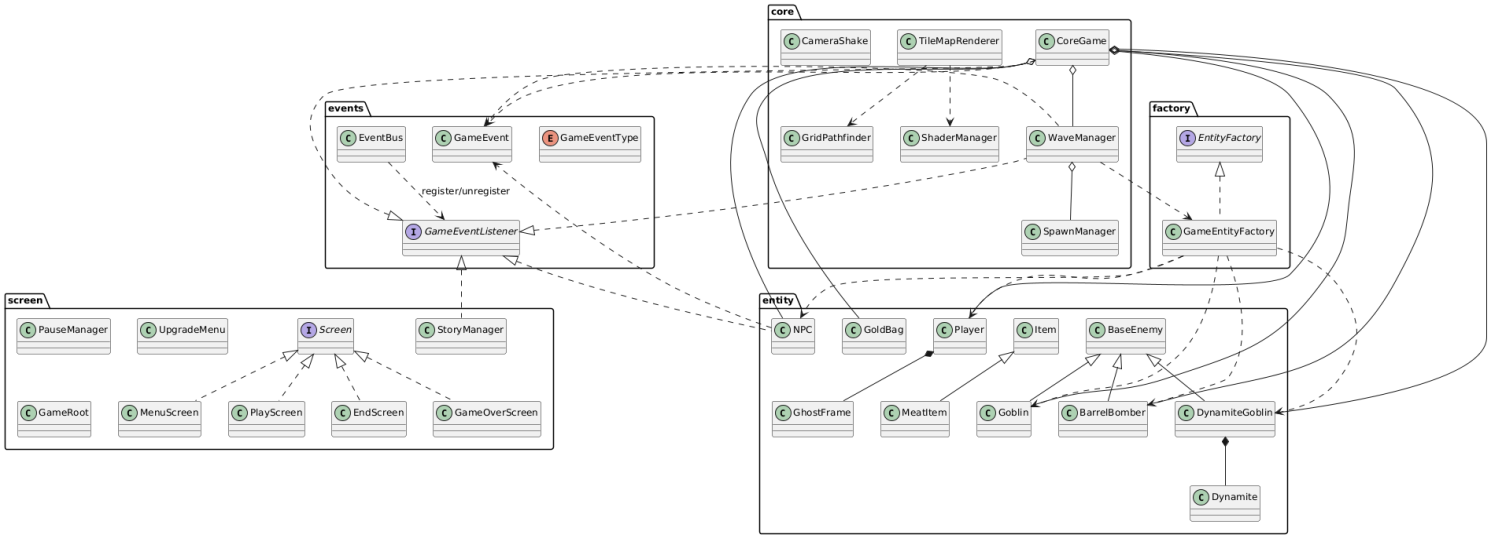
Where: Resource managers like ShaderManager, global settings in the Gradle properties (Java 17 compatibility) and the single CoreGame instance.
Problem solved: Ensures only one copy of expensive resources (shaders, fonts, map renderer) exists.

Benefits

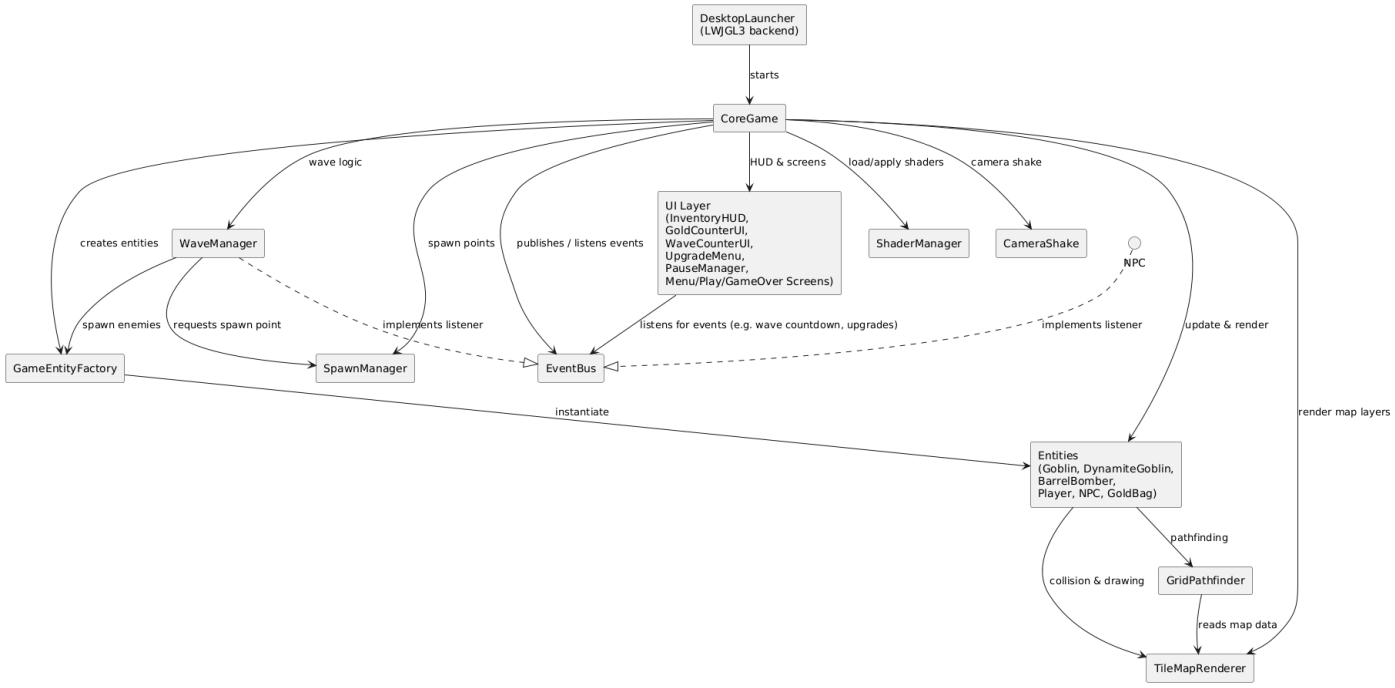
Performance: Avoids duplicated GPU uploads or texture loads.
Consistency: Uniform configuration (for example same "circle fade" shader) everywhere.
Real-World Example: The Singleton pattern is like a government mint the sole authorized source for producing all of a nation's currency.

UML Diagrams

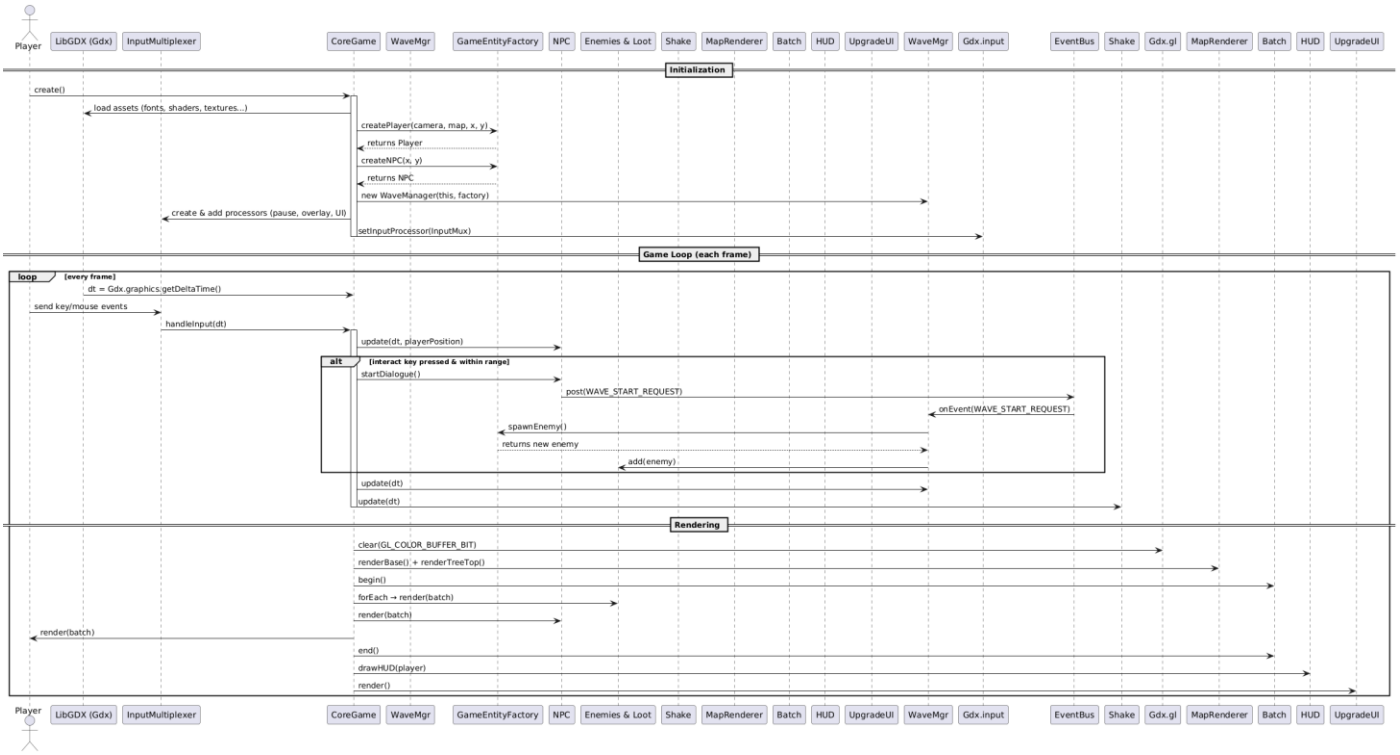
Class Diagram



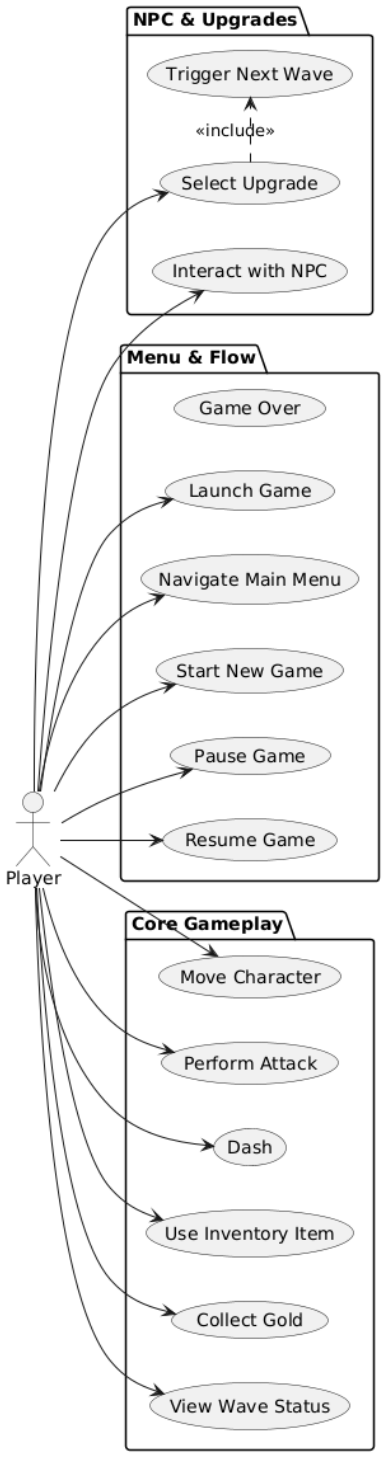
Component Diagram



Sequence Diagram



Use Case Diagram



How the UML Diagrams relate to Our Code?

The UML diagrams we produced correspond directly to the packages, classes and interactions in our Java code.

Component Diagram - Package Structure

DesktopLauncher (LWJGL3 backend) starts the application and instantiates CoreGame, matching the "starts" arrow in the component view.
CoreGame acts as the central manager: it holds references to subsystems (for example WaveManager, SpawnManager, TileMapRenderer, ShaderManager, CameraShake) and UI layers (HUD, menus, screens).
The EventBus component is shown as a globally-available event hub, matching the static EventBus.post(...) and EventBus.register(...) calls scattered throughout entities, UI components and WaveManager.

Class Diagram - Core Types and Relationships

BaseEnemy is the abstract superclass for all NPCs; it provides baseUpdate(dt), takeDamage(), collision detection (canMoveTo) and A* pathfinding (findPath).
Goblin, DynamiteGoblin and BarrelBomber extend BaseEnemy their inheritance and composition (for example DynamiteGoblin has a List<Dynamite>) map directly to the UML class boxes and arrows. Each subclass overrides update(dt), implements its own movement, attack logic and render(batch).
Player encapsulates input handling, movement, dashing, attacking (including ghost-trail generation via GhostFrame), health and inventory. Its state machine for attacks and heals corresponds to the fields in the class diagram.
NPC implements GameEventListener and manages dialogue state. The "implements listener" dashed arrow in the class diagram shows how it subscribes to wave events and triggers upgrades or wave begins.

Use Case Diagram - User Interactions

The actor "Player" in the use case view mirrors our InputMultiplexer wiring in CoreGame.create() key presses for moving, attacking, interacting (E key), pausing and opening menus trigger methods on Player, NPC, UpgradeMenu and PauseManager.
"Select Upgrade" and "Trigger Next Wave" map to UpgradeMenu.applyUpgrade(...) posting UPGRADE_SELECTED, which WaveManager.onEvent(...) listens for to call startNextWave().

Sequence Diagram - Runtime Flow

Initialization: CoreGame.create() → factory calls (createPlayer(), createNPC()), listener registrations and setting the InputMultiplexer.

<u>Game Loop</u>

Input: LibGDX polling → CoreGame.handleInput(dt) → Player.update(dt) or menu logic.
Update: Entities' update(dt) methods cascade WaveManager.update(), enemy update(), NPC.update(). They use EventBus.post() for wave requests or countdown events.
Render: CoreGame.render() clears the buffer, calls TileMapRenderer.renderBase() then entity.render(batch) for each entity, followed by HUD/menu rendering.

<u>Key Methods</u>

CoreGame.create() / render(): bootstrap and drive the main loop.
WaveManager.startNextWave(): enqueues spawn tasks and begins countdown, firing WAVE_COUNTDOWN events.
BaseEnemy.findPath(...) and GridPathfinder.findPath(...): implement reusable A* logic for all moving enemies.
Player.update(float): handles input, dash cooldown, attack execution (iterating over enemy lists and calling takeDamage(...)) and triggers UI/hud updates.
NPC.update(...): checks player proximity, types dialogue lines with a timed effect and posts WAVE_START_REQUEST or UPGRADE_MENU_REQUEST.

UI Components (UpgradeMenu, PauseManager, WaveCounterUI) subscribe to events via EventBus.register(this) and update their onscreen widgets accordingly.

Why These Patterns Matter

Modularity

Each subsystem-entity creation, event dispatching, AI pathfinding, UI screens lives in its own module, with minimal public surface area. You can upgrade or rewrite one without breaking the rest.

Testability

You can unit-test the GridPathfinder or GameEntityFactory in isolation and simulate EventBus messages to verify UI-layer reactions.

Readability & Maintenance

Other developers can quickly locate "where do we spawn Goblins?" (the factory), "how do enemies talk to the UI?" (the event bus) or "where's the A* code?" (the pathfinder).

-By combining these patterns, libGDXgame achieves a strong and extensible architecture that will stand up to new features (more enemy types, levels, UI screens) with minimal compatibility issues.

Advantages of the Chosen Patterns

Factory (GameEntityFactory)

Pros: Centralizes object creation, hides complex initialization (texture loading, animation slicing) and makes it trivial to swap or extend enemy types without touching gameplay logic.
Cons: Can become unwieldy if too many creation methods accumulate and testing factories in isolation sometimes requires extra effort.

Observer / EventBus

Pros: Decouples event producers (enemies, wave manager) from consumers (UI overlays, NPC dialogue, wave countdown). Adding new possible listeners (for example analytics, sound effects) is as simple as registering for events.
Cons: Because EventBus is global and static, it can be hard to see which listener reacts to each event, often forcing you to search through the code. Over-publishing can lead to performance or ordering issues.

State

Pros: Encapsulates behavior transitions a clear per-frame update() structure. It's easy to add new states such as "STUNNED" without rewriting existing logic.
Cons: State code can grow with many conditional branches; extracting each state into its own class (the full State pattern) would improve separation but at the cost of more classes.

Strategy

Pros: By injecting a map-based GridPathfinder strategy into BaseEnemy, different pathfinding algorithms could be swapped in (for example Dijkstra or flow fields) without touching enemy logic. Similarly, TileMapRenderer can choose at runtime whether to apply a fade shader.
Cons: The current usage is minimal; it could be overkill if only one strategy is ever used.

Disadvantages of the Chosen Patterns

Inheritance-Heavy Enemy Hierarchy

Having Goblin, DynamiteGoblin and BarrelBomber all inherit from BaseEnemy means shared code lives in the superclass, but subtle differences sometimes require overriding multiple methods (for example takeDamage(), render()). A component-based ECS (Entity-Component System) might reduce code duplication by composing reusable "movement," "attack," and "health" components.

Static EventBus

While simple, the static bus makes unit testing more difficult (listeners persist across tests unless explicitly unregistered) and can introduce hidden dependencies.

Alternative Pattern Suggestion

For a larger-scale or more extensible engine, adopting an Entity-Component-System (ECS) architecture could improve flexibility:

Entities become mere IDs.

Components (Position, Velocity, Health, AIState, Renderable) hold data only.

Systems (MovementSystem, CombatSystem, RenderingSystem) iterate over component sets to apply behavior.

Conclusion and Evaluation

The finished KnightsCrusade game demonstrates a fully playable prototype with wave-based enemy spawning, AI pathfinding, player movement and combat, UI overlays (health bars, gold counter, upgrade menu) and environmental effects (camera shake, shader-based tree fade). Performance remains smooth at 60 FPS and the codebase compiles cleanly under Java 17 with Gradle.

References

- Tiny Swords Asset Pack
  Pixel Frog. *Tiny Swords*. Itch.io.
  https://pixelfrog-assets.itch.io/tiny-swords
- Tiled Map Editor
  Thorbjørn Lindeijer et al. *Tiled: A Flexible 2D Level Editor*.
  https://mapeditor.org/
- libGDX
  libGDX Team. *libGDX Game Development Framework* (v1.13.1).
  https://libgdx.com/
- libGDX Texture Packer
  libGDX Wiki. *Texture Packer (Sprite Atlas Generator)*.
  https://github.com/libgdx/libgdx/wiki/Texture-packer
- libGDX FreeType Extension
  libGDX Wiki. *Gdx-freetype: Bitmap Font Generator*.
  https://github.com/libgdx/libgdx/wiki/Gdx-freetype
- Gradle
  Gradle Inc. *Gradle Build Tool* (v8.12).
  https://gradle.org/