TDA367/DIT213

Software Design Document

(SDD)

Inheritance of Violence

A Roguelike Survival Game

Group 12

December 2025

Table of Contents

1 Introduction

1.1 Purpose

This Software Design Document (SDD) provides a comprehensive architectural overview of *Inheritance of Violence*, a roguelike survival game developed using Java and JavaFX. The document is intended for developers who need to understand, maintain, or extend the software.

1.2 Scope

Inheritance of Violence is a fast-paced roguelike survival game where players fight endless waves of enemies in a top-down arena. The core gameplay loop consists of:

- **Combat**: Players use weapons to defeat various enemy types
- **Progression**: Defeating enemies grants experience points (XP) leading to level-ups
- **Upgrades**: Each level-up offers a choice of three random upgrades to enhance the player's build
- **Survival**: Players aim to survive as long as possible while developing unique strategies

2 System Architecture

2.1 Architectural Overview

The system follows the **Model-View-Controller (MVC)** architectural pattern with a model-heavy design. The architecture ensures the Model is completely independent of the View and Controller, allowing the game logic to be tested and potentially reused with different frontends.

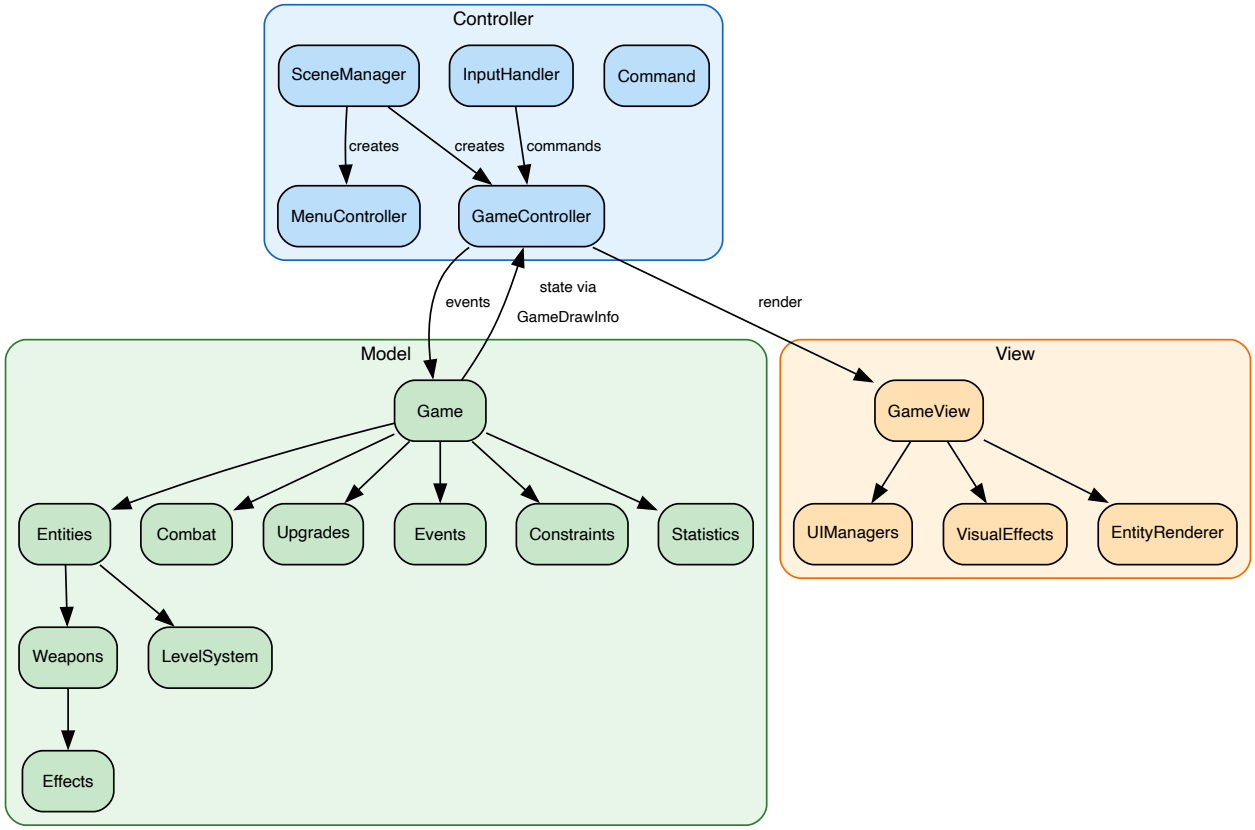

Figure 1: High-level MVC architecture diagram showing package dependencies.

Layer	Responsibility
Controller	Handles user input, manages game loop and scene transitions, coordinates Model and View
Model	Contains all game logic, state, and business rules. Completely independent of JavaFX
View	Renders game state visually using JavaFX. Subscribes to model events for reactive updates

2.2 Package Structure

```
com.grouptwelve.roguelikegame/
├── App.java                        # Application entry point
├── controller/
│   ├── Command.java                # Input command enum
│   ├── GameController.java         # Main game coordinator
│   ├── InputHandler.java           # Keyboard input translation
│   ├── InputEventListener.java     # Input callback interface
│   ├── MenuController.java         # Main menu handling
│   ├── MenuNavigator.java          # Menu keyboard navigation
│   └── SceneManager.java           # Scene transitions & DI
├── model/
│   ├── Game.java                   # Game facade
│   ├── GameWorld.java              # World dimensions
│   ├── GameDrawInfo.java           # Read-only view interface
│   ├── Velocity.java               # Direction management
│   ├── combat/                     # Combat system
│   ├── constraints/                # World constraints
│   ├── effects/                    # On-hit and duration effects
│   ├── entities/                   # Player and enemies
│   ├── events/                     # Event system
│   ├── level/                      # XP and leveling
│   ├── statistics/                 # Score tracking
│   ├── upgrades/                   # Upgrade system
│   └── weapons/                    # Weapon classes
└── view/
    ├── GameView.java               # Main view controller
    ├── ButtonListener.java         # UI button callback interface
    ├── effects/                    # Visual effects
    ├── rendering/                  # Entity rendering
    ├── state/                      # View state cache
    └── ui/                         # UI managers
```

2.3 Design Patterns

The following design patterns are employed throughout the system:

Pattern	Location	Purpose
Observer	Events package	Decouples event producers from consumers via publisher/subscriber interfaces
Singleton	EnemyPool	Ensures single instance for global access to enemy recycling
Factory	EnemyFactory	Creates entities without exposing instantiation logic
Object Pool	EnemyPool	Reuses enemy instances for memory efficiency
Strategy	UpgradeInterface, EffectInterface	Allows interchangeable upgrade and effect behaviors
Command	Command enum	Encapsulates input actions as objects
Facade	Game class	Provides simplified interface to model subsystems

2.4 Communication Patterns

The system uses two complementary communication patterns:

2.4.1 Event-Based Communication (Caused by a class in model)

State changes in the model (hits, deaths, level-ups) are broadcast through specific publisher interfaces. All subscribers reacts to relevant events. The subscribers use the events for handling different parts of the application, like visuals and game flow.

```
Model State Change → EventPublisher → Listener Interfaces → GameView/GameController/Model
```

2.4.2 Polling-Based Communication (View reads Model)

Position data is polled each frame through the read-only `GameDrawInfo` interface. The interface exposes only getter methods, preventing the view from accidentally modifying model state while polling frequently changing data like entity positions. By returning `Obstacle` interface types instead of concrete classes, the view depends only on minimal information (position, size, type), maintaining strict decoupling and complementing the event system. Polling handles high-frequency updates (movement), this avoids overhead of many events per second.

```
GameController.render() → GameView.render(GameDrawInfo) → EntityRenderer
```

3 Interface Design

3.1 External Interfaces

3.1.1 User Input Interface

The input system maps hardware inputs to abstract in-game actions, separating physical key bindings from gameplay logic. The `Command` enum defines actions like `MOVE_UP` or `ATTACK`, while `InputHandler` maintains a map translating physical keys to these commands. Multiple keys can be bound to the same command - for example, both `W` and `↑` trigger `MOVE_UP`.

This separation reduces coupling between input handling and game logic, allowing controls to be changed or extended with minimal code changes. It also enables potential support for alternative input devices (gamepads, remapped controls) without modifying the core gameplay code.

The `InputHandler` uses the **Observer** pattern to notify listeners of command events. When a key is pressed, it translates the `KeyCode` to a `Command` and notifies the registered `InputEventListener`. The listener receives both press and release events, enabling proper handling of held keys for continuous movement.

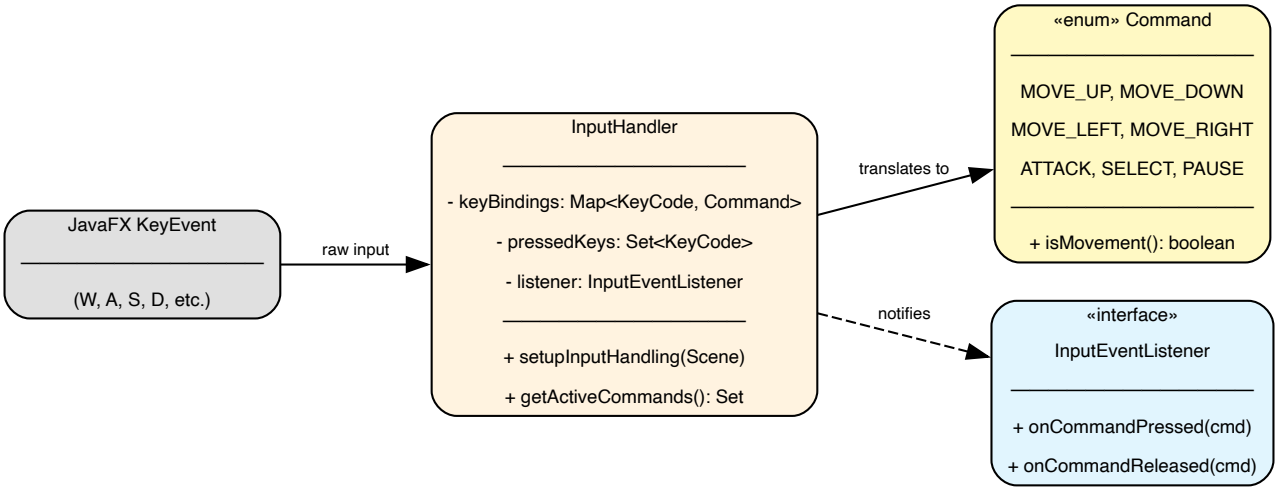

Figure 2: User input interface design.

Key Bindings:

Key	Command
W, Arrow Up	MOVE_UP
S, Arrow Down	MOVE_DOWN
A, Arrow Left	MOVE_LEFT
D, Arrow Right	MOVE_RIGHT
Space	ATTACK
Enter	SELECT
Escape	PAUSE

3.2 Internal Interfaces

3.2.1 Model-View Interface (GameDrawInfo)

The View accesses the Model through the read-only `GameDrawInfo` interface, following the **Interface Segregation Principle**:

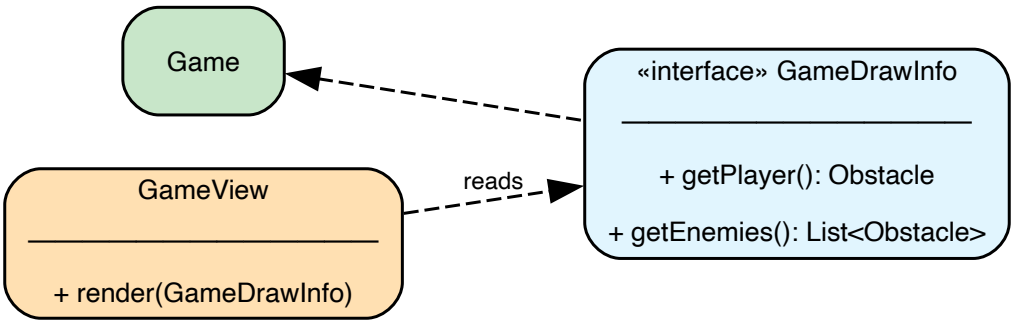

Figure 3: Read-only interface for View to access Model state.

This interface prevents the View from accidentally modifying game state while still providing access to necessary render data.

3.2.2 Output Event System

The output event system uses the **Observer** pattern to broadcast model state changes to interested listeners. Following the **Interface Segregation Principle**, the system is split into focused publisher interfaces, each handling a specific category of events.

3.2.2.1 Central EventPublisher

The `EventPublisher` class implements all publisher interfaces and maintains separate listener lists for each event type. Model classes receive only the specific publisher interface they need via dependency injection.

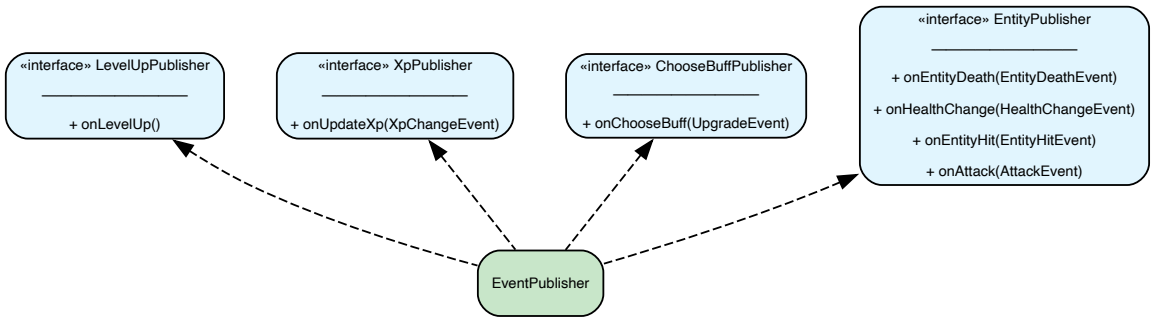

Figure 4: EventPublisher implements all publisher interfaces. Each interface also contains subscribe/unsubscribe methods.

3.2.2.2 Publisher to Listener Relationships

Each publisher interface has a corresponding listener interface. Publishers maintain a list of subscribers and notify all of them when an event occurs.

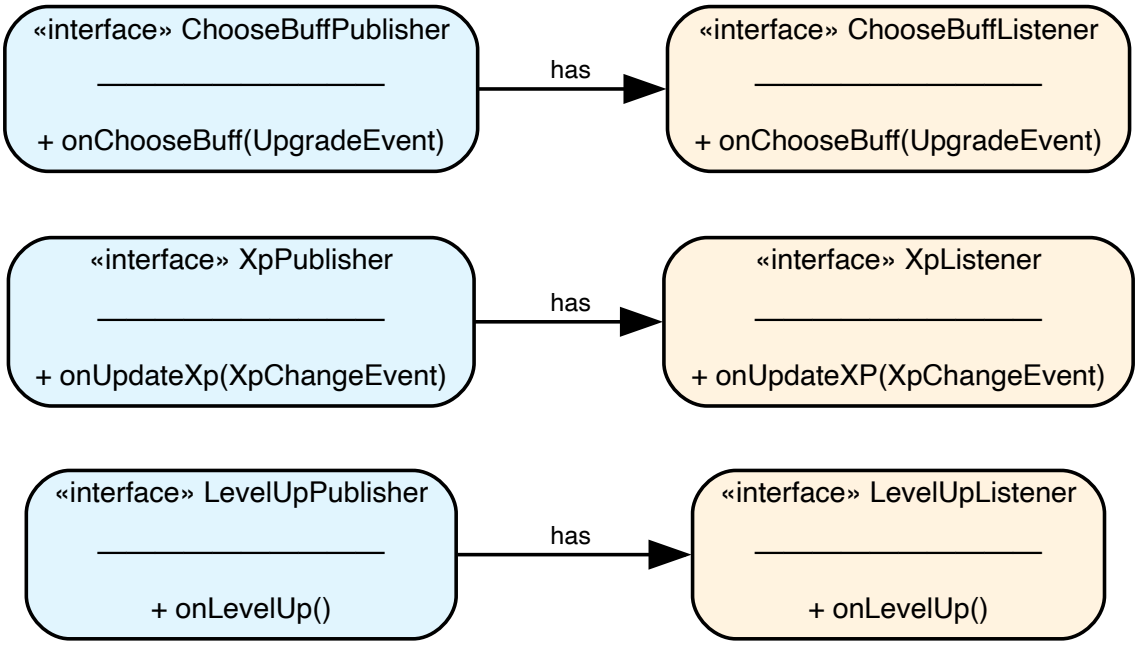

Figure 5: Publisher interfaces and their corresponding listener interfaces.

The `EntityPublisher` handles multiple related event types, each with its own listener:

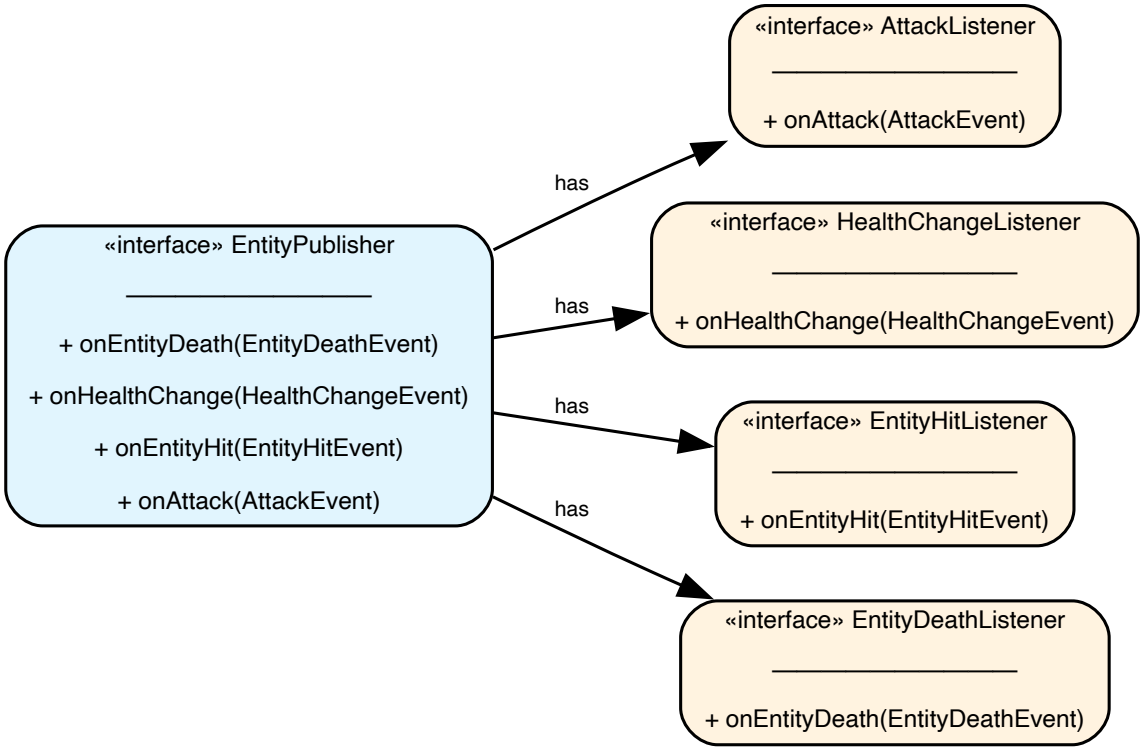

Figure 6: EntityPublisher manages four distinct listener types for entity-related events.

3.2.2.3 Event Emitters

Model classes emit events through their injected publisher interface. The `Entity` class (and its subclasses like `Player`) uses `EntityPublisher` for combat-related events. The `Player` additionally uses `LevelUpPublisher` for progression events. The `Game` class uses `XpPublisher` and `ChooseBuffPublisher` for game-flow events.

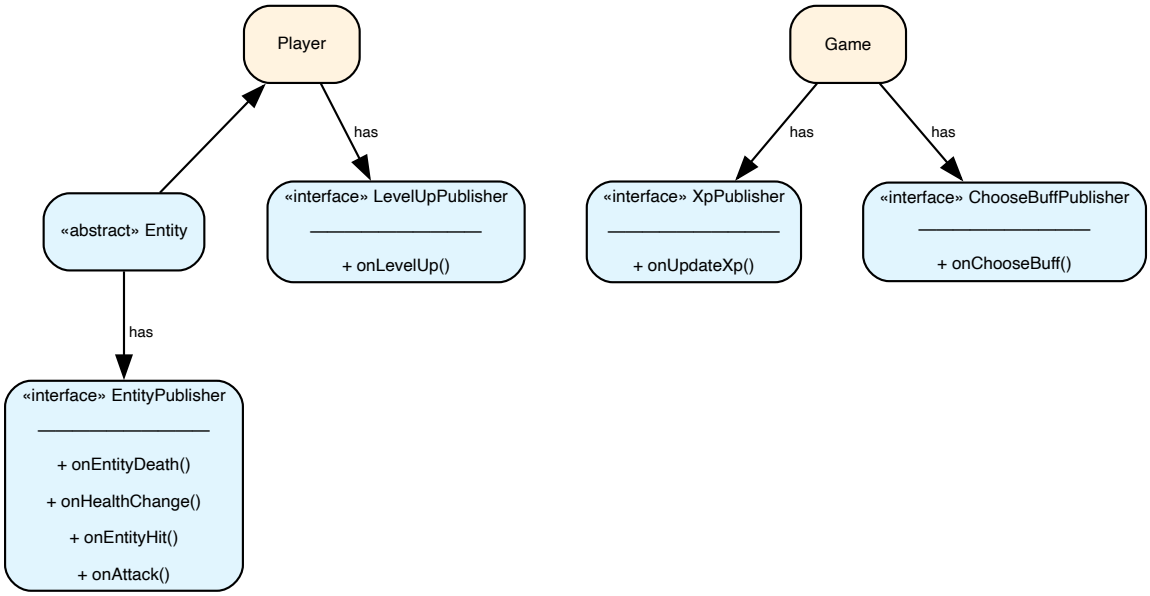

Figure 7: Model classes and the publisher interfaces they use to emit events.

3.2.2.4 Event Subscribers

Classes subscribe to listener interfaces to react to events. Subscriptions are wired up in `SceneManager.startGame()` when the game initializes.

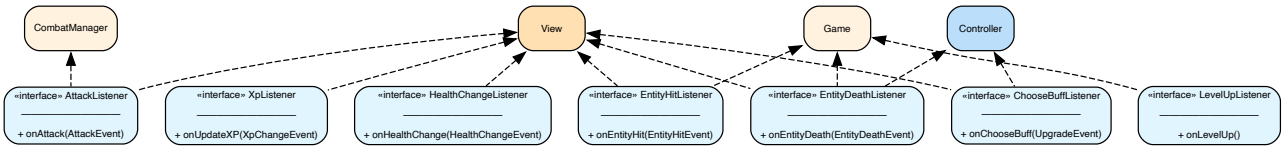

Figure 8: Classes and the listener interfaces they implement to receive events.

3.2.3 Input Events (Controller to Model)

Input events flow in the opposite direction, carrying user actions from the controller into the model. The `GameEventListener` interface defines the contract for actions the model can receive:

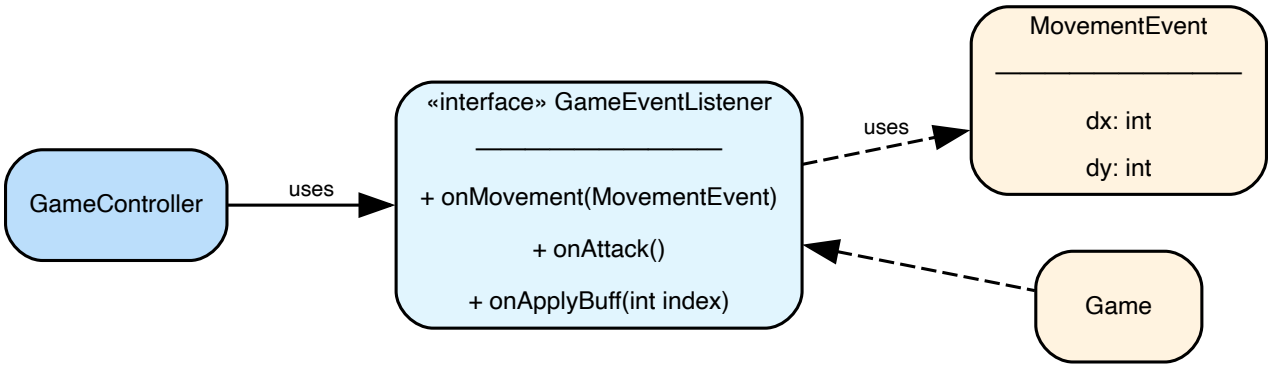

Figure 9: Input event interface: GameController calls GameEventListener methods implemented by Game.

3.2.4 Weapon Interface

Weapons implement a common interface for polymorphic combat:

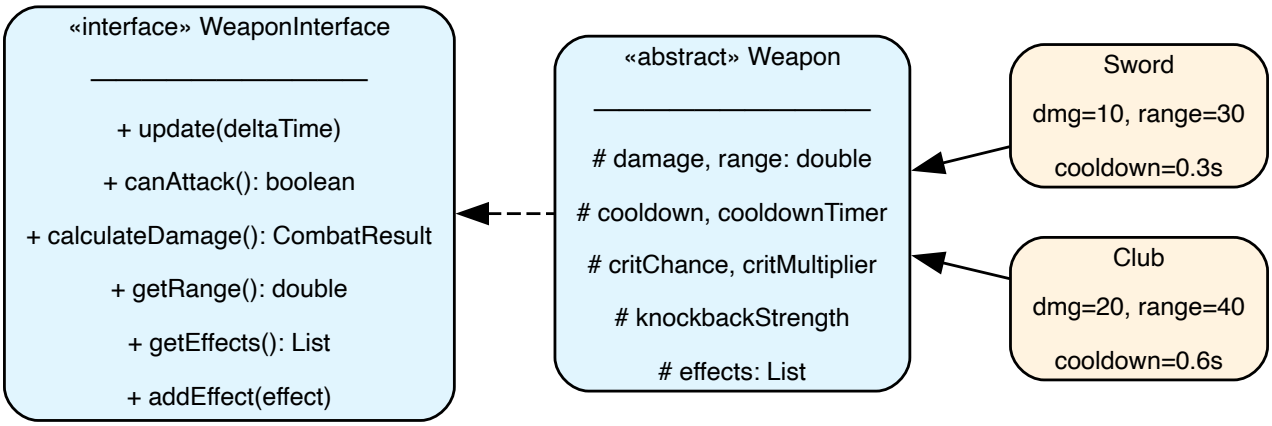

Figure 10: Weapon interface hierarchy.

3.2.5 Effect Interface

Effects use the Strategy pattern for extensible on-hit behaviors:

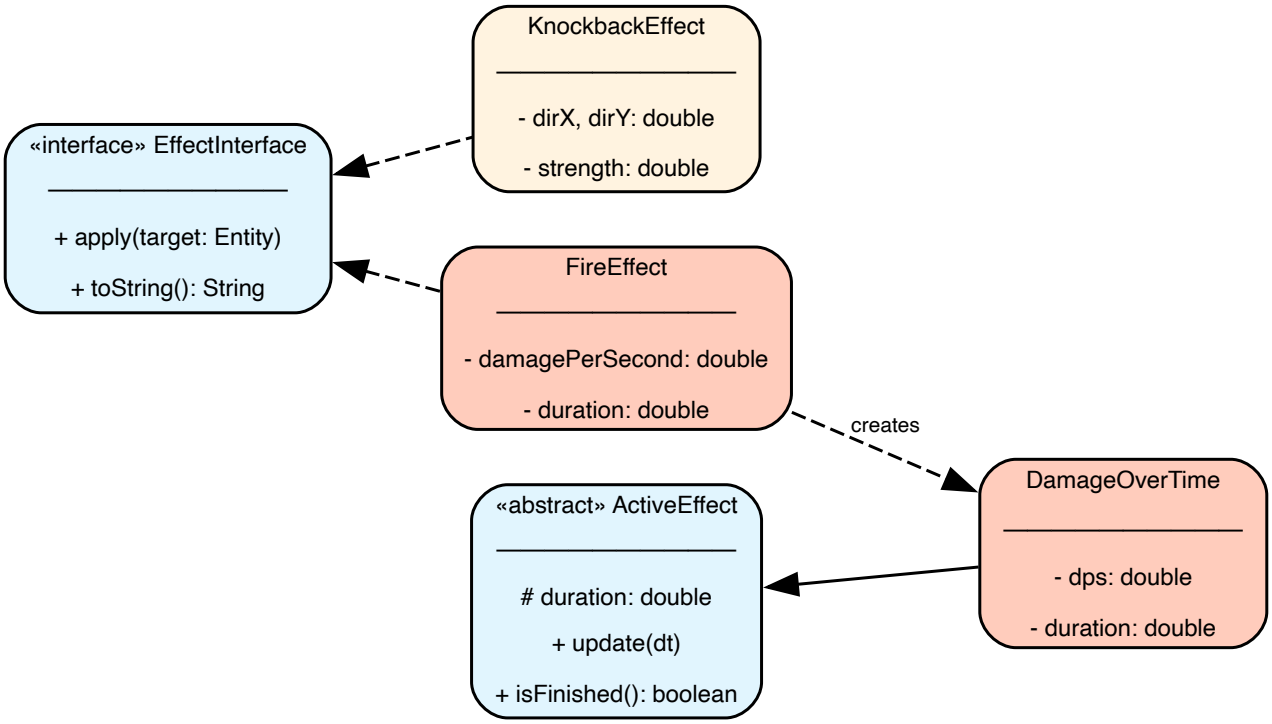

Figure 11: Effect interface hierarchy showing instant and duration-based effects.

4 Component Design

4.1 Controller Components

4.1.1 SceneManager

The `SceneManager` acts as the **Composition Root**, responsible for creating and wiring all dependencies when transitioning between scenes.

Responsibilities:

- Initialize JavaFX Stage and Scene
- Create MenuController for main menu

- Create and wire GameController, Game, GameView, and InputHandler for gameplay
- Handle scene transitions (menu to game, game to menu)

Key Method: `startGame()` - Creates all game components and connects them:

```
public void startGame() {
    Game game = new Game(worldWidth, worldHeight);
    GameView view = new GameView();
    InputHandler inputHandler = new InputHandler();
    GameController controller = new GameController(game, view, inputHandler);
    // Wire up event subscriptions...
}
```

4.1.2 GameController

The central coordinator implementing the game loop using JavaFX's `AnimationTimer`.

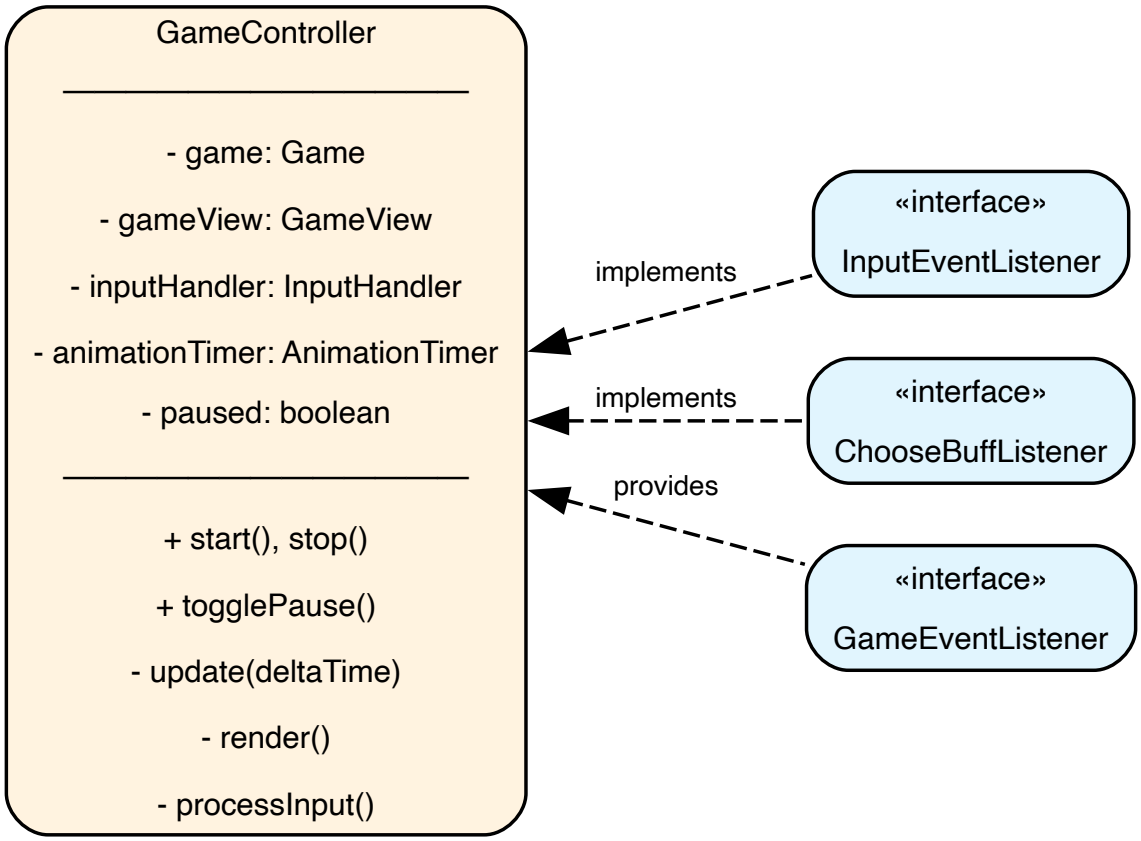

Figure 12: GameController component and its interfaces.

Game Loop Flow:

1. `AnimationTimer` triggers each frame
2. Calculate delta time since last frame
3. Process pending input commands
4. Update game state via `Game.update(deltaTime)`
5. Render current state via `GameView.render(gameDrawInfo)`

4.2 Model Components

4.2.1 Game

The `Game` class is the primary entry point for all model operations, following the **Facade** pattern.

Responsibilities:
- Aggregate all game subsystems
- Handle game update loop
- Spawn and manage enemies
- Implement event listeners for internal coordination

Implements:
- `GameDrawInfo` - Read-only view interface
- `GameEventListener` - Receives player actions
- `LevelUpListener` - Triggers upgrade selection
- `EntityDeathListener` - Handles enemy death/XP gain
- `EntityHitListener` - Coordinates hit reactions

4.2.2 Entities

Entities are the core components representing all game actors (player, enemies). Enemy types are differentiated using an `Enemies` enum rather than subclasses, allowing the `EnemyFactory` to create varied enemies with different stats from a single `Enemy` class.

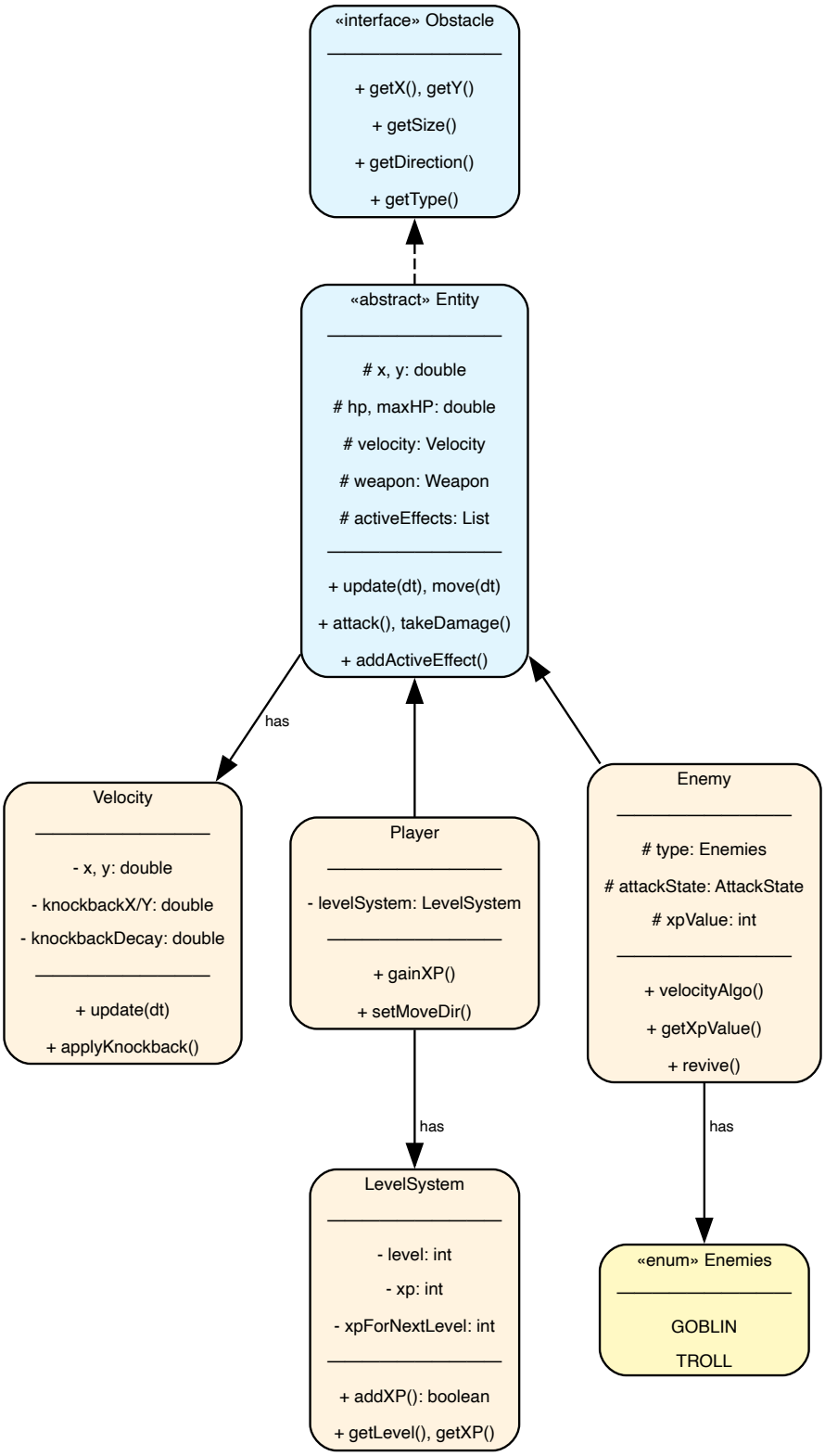

Figure 13: Entity class diagram.

4.2.3 Enemy Factory and Pool

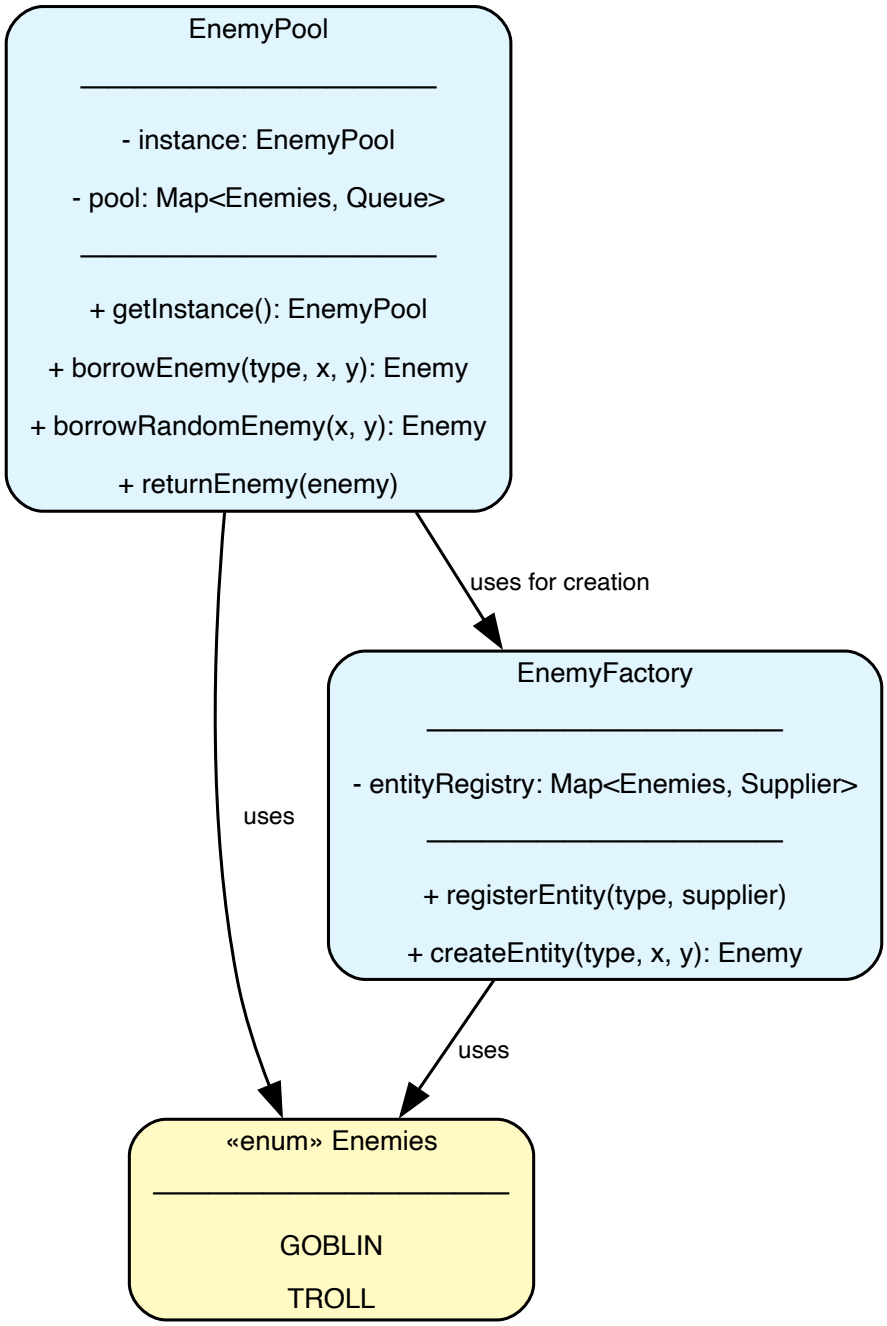

Figure 14: Enemy creation and pooling system.

Object Pool Benefits:

- Reduces garbage collection pressure during gameplay
- Enemies are "revived" with `revive(x, y)` rather than recreated
- Consistent memory footprint during long sessions

4.2.4 CombatManager

Orchestrates all combat interactions.

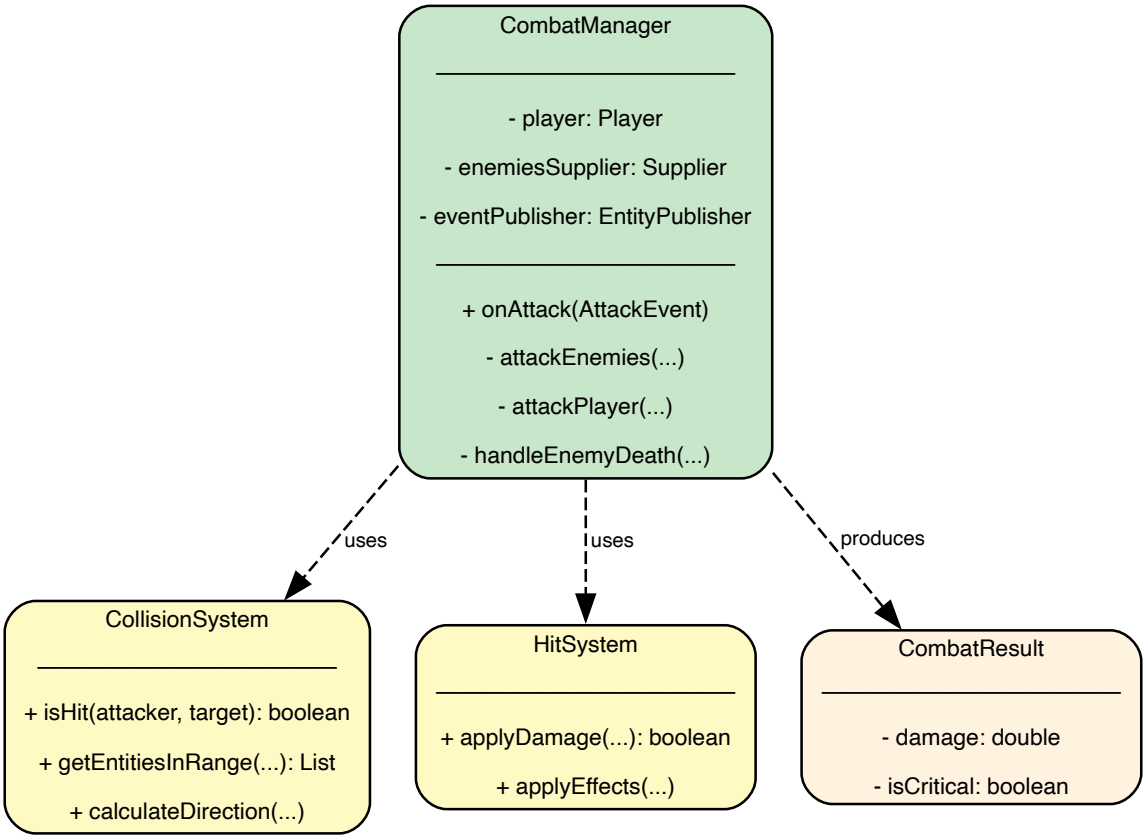

Figure 15: Combat system components.

Combat Flow:

1. Entity calls `attack()` - publishes `AttackEvent`
2. `CombatManager.onAttack()` receives event
3. `CollisionSystem.getEntitiesInRange()` finds targets
4. For each target: `HitSystem.applyDamage()` and `applyEffects()`
5. Events published: `EntityHitEvent`, potentially `EntityDeathEvent`

4.2.5 ConstraintSystem

Externalizes world rules from entity movement logic.

Current Constraints:

- `BoundsConstraint` - Keeps entities within world boundaries

Extensibility: New constraints can be added (zone effects etc.) without modifying entity code.

4.2.6 Upgrade System

The upgrade system is implemented using the **Strategy** pattern, where each upgrade represents a distinct strategy for modifying an entity's state or behavior. All upgrades implement a shared `UpgradeInterface`, allowing them to be applied through a common mechanism regardless of their internal logic.

Upgrades are categorized by what they modify:

- **Attribute Upgrades**: Modify core entity properties such as health or movement speed
- **Weapon Upgrades**: Adjust weapon parameters like damage, range, or critical hit chance
- **Effect Upgrades**: Alter behavior by adding or enhancing weapon effects, such as fire damage over time

The `UpgradeRegistry` stores upgrades as `Supplier<UpgradeInterface>` for lazy instantiation, enabling randomized values to be generated fresh each time an upgrade is offered to the player.

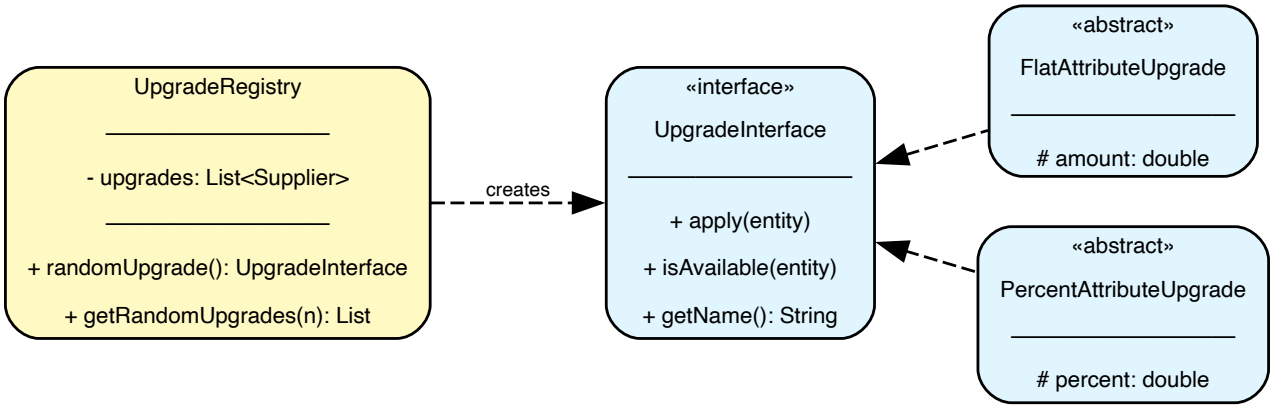

Figure 16: Upgrade system class diagram.

4.3 View Components

4.3.1 GameView

The main view controller, loaded from FXML, coordinating all visual subsystems.

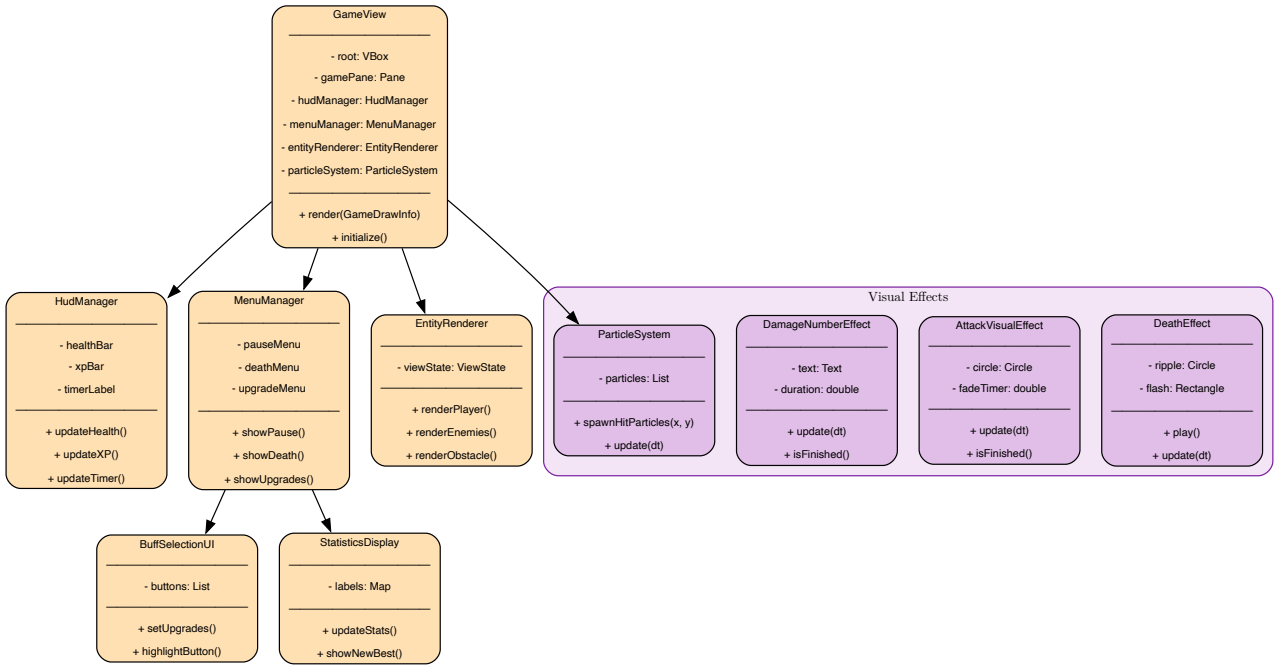

Figure 17: View component hierarchy including visual effects.

4.3.2 Visual Effects

The visual effects system provides feedback for game events:

ParticleSystem: Manages particle effects for hit feedback. When an entity is hit, small circles spawn at the impact location and radiate outward with random velocities, fading over time.

DamageNumberEffect: Displays floating damage numbers above hit entities. Numbers rise upward and fade out. Critical hits are displayed with distinct styling (larger size, different color).

AttackVisualEffect: Shows a temporary circle at the attacker's position indicating the attack range. The circle fades out quickly to provide visual feedback without cluttering the screen.

DeathEffect: Plays when the player dies. Consists of an expanding shockwave ripple emanating from the death location combined with a red screen flash overlay.

4.4 Component Interaction Flow

4.4.1 Attack Event Flow

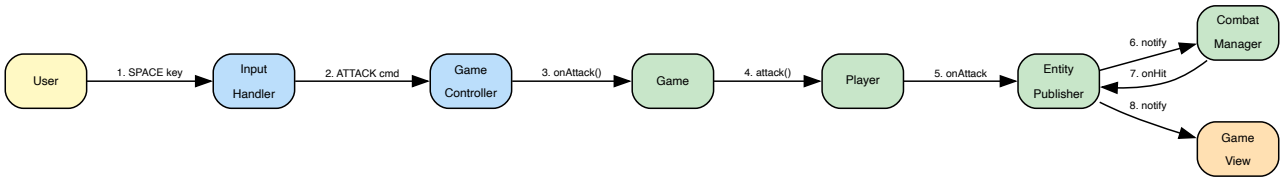

Figure 18: Attack event flow through the system.

4.4.2 Level-Up Flow

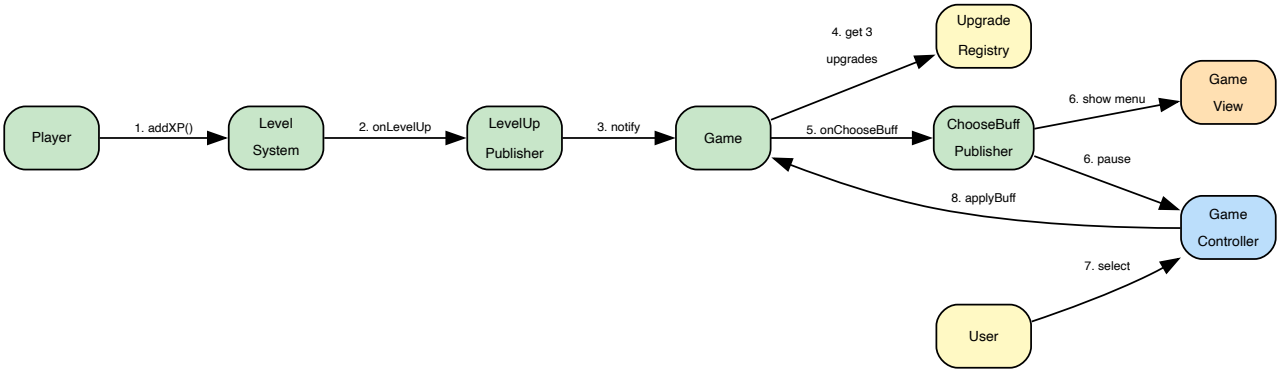

Figure 19: Level-up and upgrade selection flow.

5 Data Design

5.1 Statistics and Persistence

The game tracks player performance during each run and persists the best score between sessions.

5.1.1 Run Statistics

`GameStatistics` tracks metrics during gameplay:

- **Time Survived**: Total seconds the player stayed alive
- **Level Reached**: Highest level attained through XP gains
- **Enemies Killed**: Total enemy defeat count
- **Damage Dealt/Taken**: Cumulative combat damage values

The final score is calculated using a weighted formula:

$$\text{Score} = (\text{Time} \times 2) + (\text{Level} \times 100) + (\text{Kills} \times 10) + (\text{Damage} \times 0.1)$$

5.1.2 High Score Persistence

The `HighScore` record is an immutable snapshot created from `GameStatistics` at the end of a run. `HighScoreManager` handles persistence using Java's `Properties` API:

- **Storage Location**: `data/highscore.properties` in the project directory
- **Save Trigger**: Automatically saves when a new best score is achieved
- **Load Trigger**: Loads existing high score on game startup

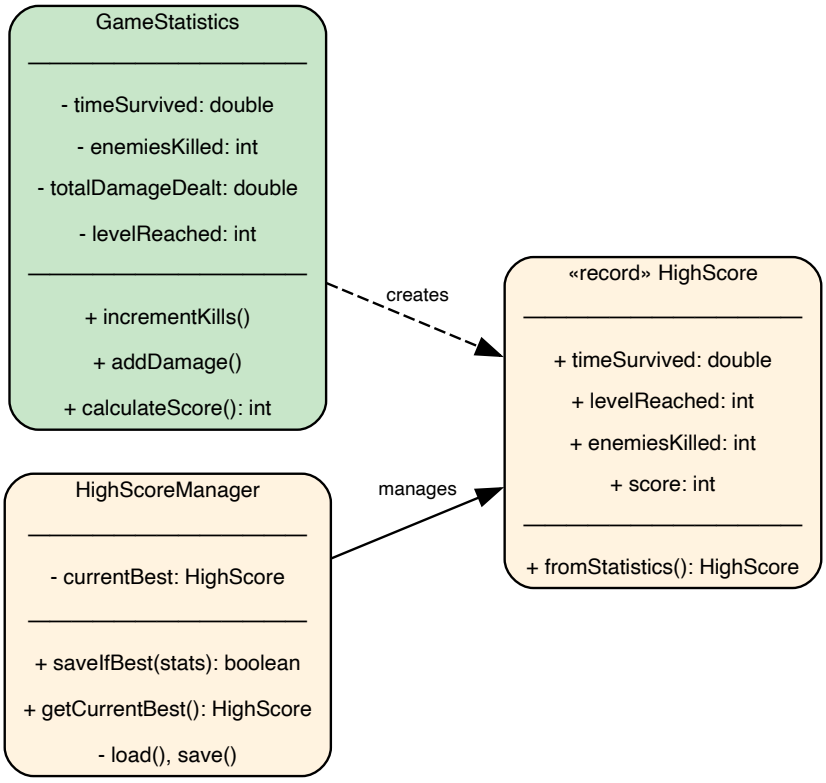

Figure 20: Statistics and high score data model.

6 Assumptions and Dependencies

6.1 Technical Assumptions

Category	Assumption
Runtime	Java 25 or later is installed on the target system
Platform	JavaFX runtime is available (bundled or system-installed)
Display	Minimum of an HD screen (720p)
Input	Keyboard input is available (no mouse required)
Storage	User has write access to the project folder for high score persistence
Performance	System can maintain 60 FPS game loop (low resource requirement)

6.2 Development Environment

- **Build System**: Maven 3.6+
- **Java Version**: Java 25+

6.3 External Dependencies

Dependency	Version	Purpose
JavaFX	25.0.1	GUI framework for rendering and input handling

6.4 Constraints

- **Input**: Keyboard-only interaction (no mouse support required)
- **Persistence**: Local file storage only (no network features)
- **Graphics**: 2D rendering using JavaFX shapes (no GPU acceleration)
- **Sound**: No audio (visual-only feedback)