Object-Oriented Design Patterns in a Mini 2D Java Game Prototype

Manikandan Gunaseelan, Owen Flack
University of Colorado Boulder

November 8, 2025

Abstract

The following is our research project on applying Object-Oriented (OO) design patterns within a simple 2-D java game. Modern game engines rely heavily on maintainable and reusable code, involving the use of design patterns such as Factory, Observer, and State. Our lightweight prototype aims to investigate how these patterns help to create a flexible entity system in our game's modular code by observing the ease of adding a new feature. We also did an extensive literature review where we observed the use of design patterns in game engines like Unity and Godot, and how the maintainability of the code-base benefited, and the associated cost of complexity with introducing these patterns.

Introduction

Object-Oriented (OO) design patterns provide proven templates for structuring complex systems into manageable, reusable components (Bezditnyi and Chebanyuk, 2024). In game development, where multiple entities interact in real time, the use of patterns enhances reusability and the speed of development (Yunato et al. 2023). Popular engines such as Unity, Unreal, and Godot rely on combinations of patterns to separate game logic, rendering, and physics (GDQuest, 2023).

Our project implements and analyzes a miniature 2D Java-based engine employing four major OO design patterns: Factory, Observer, State, and Component. The purpose is to study how these patterns improve maintainability and extensibility in small-scale game engines, and how professional practices can inform lightweight prototypes. We do this by evaluating a few metrics like code reuse, ease of adding new features, etc. The resulting system is intended as both an instructional tool and a demonstration of design quality metrics such as cohesion, flexibility, and code reuse.

Background and Related Work

Early work on pattern-based software architecture by Gamma et al. (1995) established the foundations for modular abstractions in object-oriented design, emphasizing how reusable patterns support clearer system organization and long-term maintainability. This theoretical grounding continues to influence modern game-engine research, where complexity and real-time constraints make modularity particularly important. Contemporary studies reflect an evolution from classical OO abstractions toward architectures capable of handling dynamic behavior, larger codebases, and rapidly iterating development cycles.

Building on these principles, Bezditnyi and Chebanyuk (2024) outline strategies for designing component-based architectures in modern game engines, highlighting the importance of scalability, well-defined module boundaries, and separation of responsibilities. Their work shows how traditional software-engineering fundamentals, such as interface clarity and loose coupling, translate into concrete design decisions in contemporary engines. Similarly, Yunanto et al. (2023) systematically evaluated Unity frameworks built on the Builder, Decorator, and Observer patterns, demonstrating measurable benefits in code reuse, development speed, and ease of extension when patterns are applied thoughtfully. Their findings reinforce that patterns do not simply impose structure but actively improve developer workflow and reduce technical debt over time.

From a practitioner's standpoint, modern developers often take a pragmatic perspective, adapting patterns to the realities of production workflows rather than strictly adhering to theoretical formulations. French (2024) illustrates how widely used Unity patterns, including Singleton, Observer, and State, can be integrated into everyday development to streamline communication between scripts, coordinate game-object lifecycles, and minimize dependencies across systems. His practical analyses highlight cases where patterns provide clear architectural benefits as well as situations where over-applying them may introduce unnecessary rigidity. This aligns with the broader theme in the literature: pattern conformance should support clarity, extensibility, and maintainability, not become an obstacle to rapid prototyping or creative iteration.

While Unity and other ECS-style engines offer one perspective on modularity, Godot's Node-based model, described by Linietsky (2021), represents a fundamentally different architectural philosophy grounded in hierarchical composition rather than strict component separation. This approach naturally encourages encapsulation and visual organization, making it appealing for teams that prioritize readability and scene-driven design. GDQuest (2023) expands on this perspective by offering practical techniques for applying design patterns—such as Singleton and Observer—in Godot while cautioning against misuse that could increase coupling or reduce flexibility. Their guidance demonstrates how Godot's unique abstractions require careful adaptation of classical patterns rather than direct transplantation.

More specialized work continues to push pattern-driven design into new areas of game-mechanics modeling. Mizutani and Kon's Rulebook architecture (2023) introduces a pattern for encapsulating rule-changing mechanics, such as temporary invincibility, power-ups, or environmental modifiers, into modular rule objects that can be dynamically attached, removed, or combined. This approach parallels the State pattern in its emphasis on behavioral encapsulation, but extends it to support systems in which the game's governing rules themselves may mutate at runtime. Their contribution highlights how classical design ideas can evolve to address the unique dynamism of gameplay logic, directly informing the type of modular behavior systems used in this work

Additional engine-specific research further demonstrates how classical software-engineering abstractions are adapted to the constraints of real-time interactive systems. Bucher (2017) identifies Unity-specific patterns such as GameSceneController, Concrete Factory, and Interaction patterns—each offering structured ways to manage object creation, organize complex scene transitions, or standardize player-interaction logic. He argues that these patterns help reduce code complexity, accelerate iteration speed, and improve project-wide consistency—key considerations for C-based Unity workflows where rapid prototyping often intersects with production-level constraints.

Beyond individual engines, broader architectural studies examine how subsystems interact at scale. Ullmann et al. (2024) propose the Subsystem-Dependency Recovery Approach (SyDRA), a method for analyzing and visualizing subsystem relationships across large-scale engines such as Unreal, Godot, and other open-source platforms. Their work shows how identifying dependency structures, coupling patterns, and architectural bottlenecks can directly impact maintainability, performance, and extensibility. This large-scale perspective illustrates that modularity is not merely a local design concern but a system-wide property that shapes engine evolution, developer onboarding, and long-term sustainability. Insights of this kind inform decisions even in smaller academic projects, where understanding architectural coupling early can prevent structural problems later.

Finally, the foundational contributions of Chidamber and Kemerer (1994) offer a quantitative basis for evaluating object-oriented design. Their OO metrics—focusing on coupling, cohesion, inheritance complexity, and reuse potential—provide widely adopted analytical tools for assessing maintainability and extensibility in software systems. These metrics will be applied in evaluating our own engine, allowing us to ground qualitative reasoning about modularity and pattern usage in measurable software-engineering indicators. Together, this literature forms the basis for our system's design philosophy.

System Design

Our prototype engine follows a modular architecture comprising an *Entity Manager*, *Renderer*, and *Game Loop*. Each subsystem interacts through defined interfaces, applying specific patterns to manage complexity.

Factory Pattern

The Factory Pattern dynamically instantiates game entities such as the player, enemies, or projectiles without exposing creation logic. This decouples object creation from usage:

```
Entity player = EntityFactory.create("Player");
Entity enemy  = EntityFactory.create("Enemy");
```

Observer Pattern

The Observer Pattern supports event-driven communication between modules like input, physics, and rendering. When a key event occurs, the input system notifies subscribed observers:

```
inputSubject.attach(player);
inputSubject.notify(Event.KEY_PRESSED);
```

Component Pattern

Entities in the engine are composed of interchangeable components—e.g., physics, rendering, and input—allowing flexible combinations and reuse.

```
player.addComponent(new PhysicsComponent());
player.addComponent(new RenderComponent(sprite));
```

This structure mirrors Unity's component-based design and adheres to composition-over-inheritance principles.

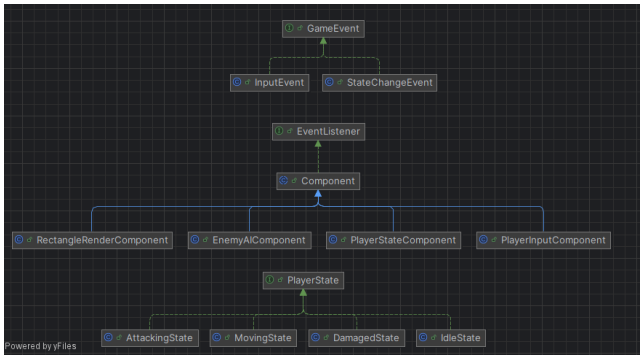

Figure 1: System Architecture

State Pattern

Entity behavior transitions (e.g., idle, move, attack, dead) are implemented via the State Pattern, improving readability and avoiding conditional sprawl:

```
player.changeState(new AttackState());
```

Architecture Overview

Figure 1 shows the relationships between the different classes in our game so far. This modular architecture enforces loose coupling and allows runtime behavior modification without significant refactoring.

Evaluation Plan

To evaluate our prototype, we define the following criteria derived from OO quality metrics:

- **Maintainability:** measured by number of existing classes modified per feature addition.
- **Flexibility:** ability to introduce new entity types or behaviors with minimal change.
- **Code Reuse:** proportion of shared components reused across entities.

Performance benchmarks (FPS stability, CPU utilization) will also be observed to ensure pattern abstraction does not impose prohibitive overhead.

Results

As we expected, the modularity of the project was greatly improved across certain aspects of the engine, specifically when it came time to add events for more complex interactions. This is due to the ease of being able to add event types quite easily without affecting any part of the program not directly affected, in this case the classes affected were the playerStateProvider class and World class which are responsible for keeping track of the player's state and taking the player input and any global logic like the position of entities respectively. This also speaks to our point on flexibility, with minimal change being required here, as to add the walls we only needed to create a new event class and modify the world to handle it, every other entity was isolated from this effect and remained unchanged. Using our evaluation criteria we can see:

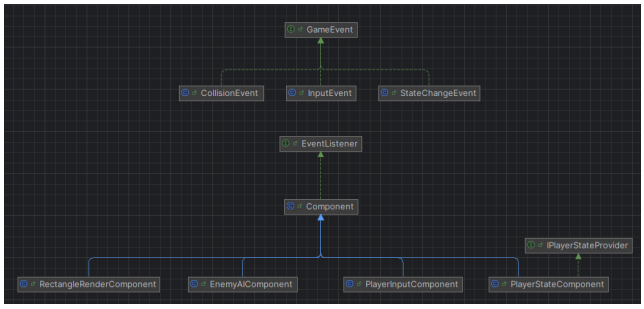

Figure 2: Modified Architecture after adding collision with walls

- **Maintainability:** We only had to modify two classes, that being the world class for the physics and the state provider class to provide the status to the player.
- **Flexibility:** To create the new collsion logic, we didn't need to do much, we extended the behavior of the StateEvent class as shown in Figure 2 and update the logic within the game engine.
- **Code Reuse:** The World class was able to reuse the same code it already used in the damage calculation for checking for overlapping entities, the logic simply had to be extended to distinguish between what type of collision it was and how to resolve it, this was primarily done by the newly created Collsion Event class.

Modifying Game Logic

Figure 2s shows the section of the architecture that had to be modified in order to create the logic for players and enemies interacting with walls.

The advantages of building our game with Object Oriented design principals are not exclusive to functional modularity, but also to the readability of the code. Creating these many small state objects for the player and for the events allows for a simple skimming of the implementation file to understand how a particular behavior is expected to affect the player or how the game responds to an event happening. For example, taking a look at the player damaged state

below shows how little code one has to read before being able to understand generally how the player is affected by damage:

```
public class DamagedState implements PlayerState
    @Override
    public String name() {
        return "Damaged";
    }

    @Override
    public void enter(
        PlayerStateComponent context,
        Entity entity) {
        entity.takeDamage(1.0);
    }

    @Override
    public void handleInput(
        PlayerStateComponent context,
        Entity entity,
        InputEvent event) {

    }

    @Override
    public void update(
        PlayerStateComponent context,
        Entity entity,
        double deltaSeconds) {
        if (context.hasMovementInput()) {
            context.setState(
                context.movingState());
        } else {
            context.setState(
                context.idleState());
        }
    }
}
```

Here we can instantly see the name of the state being Damaged and player input is unchanged when in this damaged state. Further we can see that when this state is updated by the engine, it simply checks if the player is pressing a movement input or not, which would then change the state to movement if there was a key pressed or idle if there was not. Finally there's the simple logic to take damage.

References

Bezditnyi, V., & Chebanyuk, O. (2024). *Software Engineering Fundamentals to Design Application for Modern Game Engines.* In *Proceedings of the 14th International Scientific and Practical Conference on Programming UkrPROG'2024*, Kyiv, Ukraine. CEUR Workshop Proceedings, Vol. 3806. Available at https://ceur-ws.org/Vol-3806/S_46_Bezditnyi_Chebanyuk.pdf.

Yunanto, A. A., Jamal, S., Sa'adah, U., Aziz, A. S., Permatasari, D. I., Nailussa'ada, N., & Hardiansyah, F. F. (2023). *Implementation of Design Patterns on Unity Components to Increase Reusability and Game Speed Development.* In *Proceedings of the 2023 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pp. 366–373. IEEE.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional. ISBN 0-201-63361-2.

Chidamber, S. R., & Kemerer, C. F. (1994). *A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering,* 20(6), 476–493. IEEE.

Mizutani, W. K., & Kon, F. (2023). *Rulebook: An Architectural Pattern for Self-Amending Mechanics in Digital Games.* In *Proceedings of the 2023 Brazilian Symposium on Games and Digital Entertainment (SBGames)*, pp. 1–10. SBC OpenLib.

GDQuest. (2023). *Intro to Design Patterns in Godot.* Retrieved from https://www.gdquest.com/tutorial/godot/design-patterns/intro-to-design-patterns/.

Linietsky, J. (2021). *Why Isn't Godot an ECS-Based Game Engine?* Godot Engine Official Blog. Retrieved from https://godotengine.org/article/why-isnt-godot-ecs-based-game-engine/.

Bucher, N. (2017). *Introducing Design Patterns and Best Practices in Unity.* In *Proceedings of the 2017 ACM Southeast Conference (ACM SE '17)*, Kennesaw, GA, USA, pp. 1–8. ACM. https://doi.org/10.1145/3077286.3077322.

Ullmann, G. C., Guéhéneuc, Y.-G., Petrillo, F., Anquetil, N., & Politowski, C. (2024). *SyDRA: An Approach to Understand Game Engine Architecture.* In *Proceedings of the 20th International Conference on Mining Software Repositories (MSR)*, pp. 1–15. arXiv preprint https://arxiv.org/abs/2406.05487.

French, J. (2024). *Design Patterns in Unity (and When to Use Them).* Game Dev Beginner. Retrieved from https://gamedevbeginner.com/design-patterns-in-unity-and-when-to-use-them/.