Pattern Use Case Mapping

Below is a **Pattern–Use Case Mapping Matrix**, followed by brief explanations and justifications for each mapping. Every use case has at least one primary pattern, with additional patterns as secondary options.

| **Use Case** | **Primary Pattern** | **Secondary Patterns** |
| --- | --- | --- |
| **1. Play Game** | Factory Method | Strategy, Observer |
| **2. View Leaderboard** | Strategy | Template Method |
| **3. Create Profile** | Builder | — |
| **4. Buy Avatar** | Command | Observer |

## **Use Case 1: Play Game**

#### **Factory Method**

## **Why it’s appropriate:**

## Each game type (Hangman, Car Race, Balloon, etc.) has unique setup, question pools, and win/loss conditions that must be instantiated correctly.

## Embedding all creation logic in one place would lead to tangled, hard-to-maintain code as more games are added.

## Factory Method cleanly delegates creation to subclasses, keeping the controller simple and focused.

## **Benefits:**

## New games can be added by simply creating a new Factory subclass—existing code remains untouched.

## Encapsulation of creation logic improves readability and enforces single responsibility on each factory.

## Adheres to the Open/Closed Principle, making the system easy to extend without modification of core modules.

#### 

#### **Strategy**

## **Why it’s appropriate:**

## Difficulty levels (“easy”, “medium”, “hard”) require different algorithms for move generation, scoring, or timing.

## Hard-coding these variations would scatter conditionals throughout the game engine.

## Strategy allows selecting the appropriate algorithm at runtime based on user choice.

## **Benefits:**

## Each difficulty algorithm lives in its own class, making it straightforward to test and maintain.

## Adding or tweaking a difficulty only requires implementing or updating one Strategy class.

## Core game logic remains decoupled from difficulty details, reducing risk of unintended side effects.

#### **Observer**

## **Why it’s appropriate:**

## Gameplay generates events (score changes, health updates, level completion) that multiple components must react to.

## Tight coupling between the game engine and UI or analytics modules would make these reactions hard to evolve.

## Observer pattern publishes events to any number of listeners without the game needing to know their identities.

## **Benefits:**

## UI components, live leaderboards, and analytics services can subscribe or unsubscribe independently.

## Promotes a clean, event-driven architecture that scales as features grow.

## 

### **Use Case 2: View Leaderboard**

#### **Strategy**

## **Why it’s appropriate:**

## Leaderboards can be “global,” “weekly,” or “friends-only,” each with a distinct ranking algorithm.

## Embedding all logic in a single service method would require complex conditionals and reduce clarity.

## Strategy cleanly separates each ranking algorithm into its own class, chosen at runtime.

## **Benefits:**

## Ranking modes are independently testable and maintainable.

## Adding a new mode is as simple as implementing a new Strategy class.

## The leaderboard service remains open for extension but closed for modification.

#### **Template Method**

## **Why it’s appropriate:**

## The overall leaderboard workflow (fetch → filter → sort → format) is common to all modes.

## Individual steps (e.g., filtering by date range, formatting for mobile) may need customization per mode.

## Template Method defines the invariant sequence in a base class while letting subclasses override specific steps.

## **Benefits:**

## Shared workflow logic is centralized, reducing code duplication.

## Subclasses can override only the steps they need to change, preserving the overall structure.

## Ensures consistency across modes while supporting targeted variations.

## 

### **Use Case 3: Create Profile**

#### **Builder**

## **Why it’s appropriate:**

## Profile objects have many fields—some mandatory (username, password), others optional (avatar, preferences).

## Constructors with long parameter lists are error-prone and hard to read.

## Builder provides a clear, fluent API for step-by-step object construction.

## **Benefits:**

## Centralizes validation of required fields in the build() method.

## New optional attributes can be added via new builder methods without breaking existing calls.

## Improves code readability by naming each parameter explicitly in the chain.

## 

### **Use Case 4: Buy Avatar**

#### **Command**

## **Why it’s appropriate:**

## The purchase flow (balance check → deduct currency → assign avatar → log) is a cohesive unit of work.

## Encapsulating it in a command object decouples the controller from business logic.

## Commands can be queued, retried, or undone, offering flexibility in execution.

## 

## **Benefits:**

## Keeps controllers thin by delegating complex workflows to command objects.

## Supports transactional execution and built-in retry or undo semantics.

## Simplifies logging and metrics collection by wrapping execution in a single interface.

#### **Observer**

## **Why it’s appropriate:**

## Once a purchase completes, multiple subsystems (wallet display, notification, achievements) must react.

## Tight coupling between purchase logic and each subscriber would hinder maintainability.

## Observer lets the purchase service publish a PurchaseCompleted event to any number of listeners.

## **Benefits:**

## Subscribers handle their own concerns independently, promoting single responsibility.

## Decouples side effects from core business logic, making the system more modular.

## 