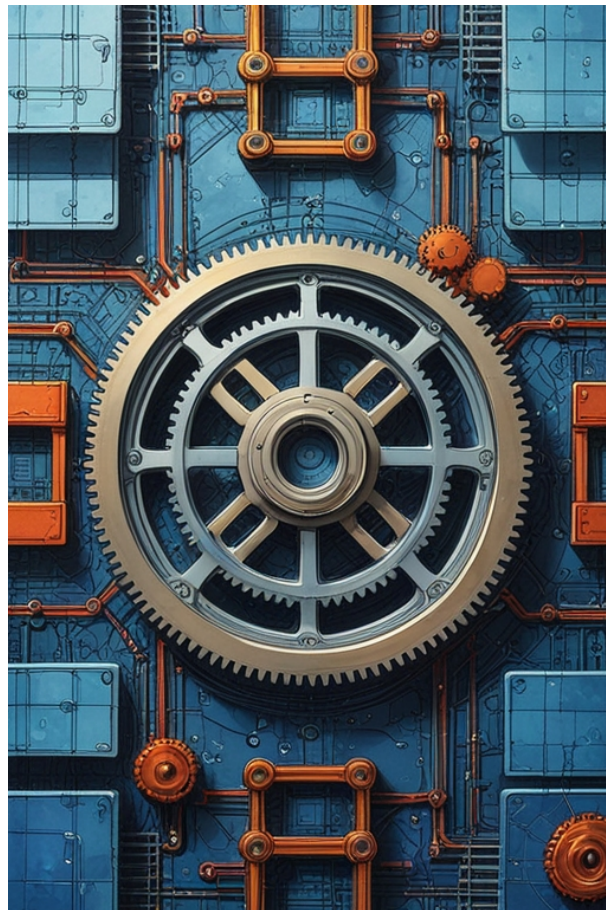

Game Design Patterns






























Patterns for the clear development practices to improve your code quality and maintainability. 🧠

All patterns organized by pattern types and sorted alphabetic to help you find the right pattern for your project. 🔍



March 19, 2025

Contents

1	 Patterns	2
1.1	Creational Patterns	2
1.1.1	 →  Abstract Factory	2
1.1.2	 Builder	2
1.1.3	 Factory Method	3
1.1.4	 Object Pool	3
1.1.5	 Prototype	3
1.1.6	 Singleton	4
1.2	Behavioral Patterns	4
1.2.1	 Chain of Responsibility	4
1.2.2	 Circuit Breaker	4
1.2.3	 Command	4
1.2.4	 Dirty Flag	5
1.2.5	 Event Queue	5
1.2.6	 Interpreter	5
1.2.7	 Iterator	5
1.2.8	 Mediator	6
1.2.9	 Memento	6
1.2.10	 Observer	6
1.2.11	 State	7
1.2.12	 Strategy	7
1.2.13	 Template Method	7
1.2.14	 Visitor	7
1.3	Structural Patterns	8
1.3.1	 Adapter	8
1.3.2	 Bridge	8
1.3.3	 Composite	8
1.3.4	 Decorator	8
1.3.5	 Facade	9
1.3.6	 Flyweight	9
1.3.7	 Proxy	9

1 Patterns

Patterns are divided into three categories: Creational, Behavioral, and Structural.











- Creational Patterns: These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
- Behavioral Patterns: These patterns deal with object collaboration and how objects interact with each other.
- Structural Patterns: These patterns deal with object composition and how objects can be composed to form larger structures.

1.1 Creational Patterns

1.1.1 → Abstract Factory

Create families of related objects.

Usage:





1. I want to create different products (e.g. furniture and electronics).  
2. I have different factories for different products.   and  
3. Each factory makes products according to a certain type (e.g. furniture, electronics).  
4. I have a variety of products according to the need.  ,  ✓

1.1.2 Builder




Create complex objects step by step.

[HamburgerBuilder]  → Add  bottom bun →  Add patty →  Add cheese →  Add lettuce → Add  Top with bun → Build to get hamburger 

Usage:

1. I want to build something more complex (e.g. a car). 
2. I have a builder who will help me. 
3. I choose step by step what features the car will have (e.g. color, wheels, engine). 
4. At the end I get the finished car.  ✓

Usage 2: with Director

1. I want to build a specific type of car (e.g. sports car). 
2. I have a director who knows the exact recipe. 
3. The director tells the builder what to do in the right order. 

4. The director can use the same builder to create different car types. 🚗🚚

The director is optional and can be used to simplify the building process.

1.1.3 🏭 Factory Method

Centralize object creation.

Usage:

1. I want to create an object (e.g. a car). 🚗
2. I have a factory that knows what object to make. 🏭
3. The factory creates the object according to the specifications (e.g. electric car). ⚙️
4. The factory gives me the finished object. 🚗✅

1.1.4 🗃️ Object Pool

Recycles objects instead of creating and destroying them frequently (very useful for performance in games).

Usage:

1. I have objects that are expensive to create (e.g. bullets). 💣
2. I create a pool of objects that I can reuse. 🗃️
3. Instead of creating new objects, I take them from the pool. 📦↑
4. When I'm done, I return the object to the pool. 📦↓

1.1.5 🔄 Prototype


Copy existing objects instead of creating them from scratch.

Usage:




1. I have an original object (e.g. a car). 🚗
2. I want to create a new object that will be identical to the original. 🔄
3. Instead of creating a new object from scratch, I copy the original object. 📄🚗
4. I get a new object with the same properties. 🚗✅

1.1.6 Singleton

Ensures a class has only one instance and provides a global point of access to it.

Only one  object is created and shared across the game.

Usage:




1. I have one object. 
2. This object is shared cross the game. 
3. No matter how many times I use it, it will always be the same object. 

1.2 Behavioral Patterns

1.2.1 Chain of Responsibility

Allows you to pass requests between objects until an object resolves the request.






Usage:

1. I have a request I want to solve. 
2. I pass the request to other objects until someone solves it. 
3. Each object tries to resolve the request before passing it on. 

1.2.2 Circuit Breaker

Protects the system from cascading failures and enables graceful degradation.



Usage:



1. I have a system that can fail (e.g. network calls). 
2. I want to prevent cascading failures. 
3. Circuit breaker monitors failures and temporarily isolates problematic parts. 
4. The system can automatically recover. 
5. I prevent system overload. 

1.2.3 Command

Encapsulate requests as objects that can be passed, logged, or cancelled.

Usage:






1. I want to perform an action (e.g. turn on the TV). 
2. I create a command that represents this action. 

3. The command is executed when I click on it (e.g. a button on the remote). 
4. The action is executed according to the command. 

1.2.4 ⚡ Dirty Flag

It marks objects as “dirty” and updates them only when necessary (UI and computation optimization).





Usage:

1. I have objects that need to be updated. 
2. I mark the objects as “dirty” when they need to be updated. 
3. I update the objects only when they are marked as “dirty”. 
4. I save resources by updating only when necessary. 
5. The objects are updated only when needed. 

1.2.5 🧑🧑🧑 Event Queue

Allows you to manage events to be processed in a specific order (e.g. animations, AI decision making).





Usage:

1. I have multiple events that need to be processed in a specific order. 
2. I add events to the queue. 
3. The events are processed one by one in the order they were added.  

1.2.6 🗒️ Interpreter

Takes text commands and turns them into actions by following a set of rules.

Usage:

1. I have a simple command language (e.g. “MOVE UP 10”). 
2. The interpreter knows how to understand these commands. 
3. It breaks down the command and executes the corresponding action. 
4. The command gets processed according to defined rules. 

1.2.7 🔁 Iterator

Allows you to browse through the objects of a collection without needing to know its internal structure.

Usage:

1. I have a collection of objects (e.g. a list of books 📖).
2. I don't want the client to see the details of its implementation 🔒.
3. I will create an Iterator that allows easy traversal of the collection 🔍.
4. The client can iterate without worrying about how the data is organized ✅.

1.2.8 🤝 Mediator

Controls communication between objects so they don't have to talk to each other directly.

Usage:

1. I have multiple objects that communicate with each other. 💬
2. Instead of communicating directly, they use a mediator to coordinate. 🧑🏫💬
3. The mediator makes sure everything is done correctly. ✅

1.2.9 💾 Memento

Saves the state of the object so that it can be rolled back.

Usage:

1. I have an object that I want to save (e.g. a game state of the object). 🎮
2. I create a memento that saves the state of the object. 💾
3. I can restore the object to its previous state using the memento. 🔄
4. I can save and restore the object at any time. ✅

1.2.10 👁️ Observer

Allows objects to track and react to changes in other objects.

Usage:

1. I want to know when something important happens (e.g. when my package is delivered). 📦
2. I subscribe to notifications about this event. 🔔
3. When the event happens, the system notifies all subscribers. 📱
4. I receive the notification and can react to it. ✉️

1.2.11 🏃 State

Allows the object to change its behavior when the state changes.

Usage:

1. I have an object that has different states. 🔄
2. When the state changes (e.g. the car is on or off), its behavior changes. 🚗
3. The behavior of the object adapts to the current state. ⚡

1.2.12 🧑 Strategy

It allows you to change the algorithm (strategy) of an object at runtime without changing its code.

Usage:

1. 🗂️ I have different ways to do something (e.g. calculating travel costs).
2. 🧠 I can choose the strategy that suits me best.
3. 🔄 I can change my strategy depending on the circumstances.
4. 🏁 The result will adapt to the chosen strategy.

1.2.13 📝 Template Method

It defines the skeleton of the algorithm, but lets subclasses implement specific steps.

Use case:

1. I have a general procedure to follow (e.g. a cake recipe). 🍰
2. The template contains fixed steps. 📝
3. I only change the details (e.g. ingredients or baking temperature). 🔍🍰
4. At the end I get the finished result according to the template. 🎉🍰

1.2.14 🧑 Visitor

Allows you to add new operations to objects without changing their classes.

Usage:

1. I have objects that need new operations (e.g. documents). 📄
2. I create a visitor that knows how to handle different objects. 🧑
3. The visitor adds new operations without changing the objects. ⚙️
4. Each object accepts the visitor and lets it perform operations. 🤝
5. I can add new operations by creating new visitors. ✅

1.3 Structural Patterns

1.3.1 🧩 Adapter

Used for incompatible interfaces to work together by providing a wrapper that converts one interface into another.

Usage:

1. I have a device that doesn't work with my current system. 🖱️❌
2. I use an adapter so that both devices can work together. 🖱️➡️💻
3. Now I can use the devices without any problems. 📱✅

1.3.2 🌉 Bridge

It separates abstraction from implementation so that they can be changed independently.

Usage:

1. I have an object that can work on different platforms. 🎮
2. I separate the object from the platform. 🖱️
3. I can change the platform without changing the object. 🔄
4. The object works on different platforms. 🎮✅

1.3.3 🧱 Composite

It allows you to work with individual objects as well as entire groups.

Usage:

1. I have simple objects that I want to group into complex structures. 🧩
2. These objects can have subobjects that can also have their subobjects. 📁
3. I can work with groups as well as individual objects. ✅

1.3.4 🌲 Decorator

Dynamically adds new object functionality.

Usage:

1. I have an object that I want to enrich with new properties. 🌲
2. I use a decorator to add new features to the object. ☀️
3. The object now has new features, but still remains the same base. ✅

1.3.5 🏛️ Facade

Provides a simple interface for a complex system.

Usage:

1. I have a complex system (e.g. home theater). 🎧🎵
2. I want to control this system simply (e.g. with a button on the remote control). 🕸️
3. the facade will hide the complexities and offer me a simple approach. 🙌
4. For me, everything works without the need to know the details. 🎬🎉

1.3.6 🖋️ Flyweight

Shares common data between many object instances (saves memory).

Usage:

1. I have many objects that are the same (e.g. letters). 📄
2. I want to share them to save memory. 💾
3. I share objects instead of creating them for each letter separately. 🖋️

1.3.7 🛡️ Proxy

Provides a shortcut for another object to be controlled or protected.

Usage:

1. I have an object that I want to control or protect. 🔒
2. I create a proxy that will help me with this. 🔗
3. The proxy can control access to the object or add additional functionality. 🔒🔗
4. I can use the proxy instead of the object. 🛡️✅