Tipos de Patrones de Diseño

Tomás Felipe Melli

July 15, 2025

Índice

1 Tipos de Patrones de Diseño

1.1 Creational Patterns

Los patrones creacionales se encargan de abstraer el proceso de creación de objetos, evitando que el sistema dependa directamente de cómo se instancian, componen o representan.

Hay dos variantes:

- Patrones de clase: usan herencia para variar qué clase se instancia.
- Patrones de objeto: delegan la creación a otro objeto.

Son fundamentales cuando el diseño evoluciona de herencia rígida hacia composición flexible. En lugar de codificar directamente comportamientos complejos, se parte de comportamientos básicos que pueden componerse.

Todos los patrones creacionales:

- Encapsulan el conocimiento sobre qué clases concretas se usan.
- Ocultan cómo se crean y ensamblan esas instancias.
- Brindan flexibilidad para decidir qué se crea, quién lo crea, cómo y cuándo, en forma estática o dinámica.

Se usa un juego con laberintos para ilustrar estos patrones. El laberinto se compone de habitaciones (Room), puertas (Door) y paredes (Wall), todas heredan de MapSite:

```
class MapSite {
public:
  virtual void Enter() = 0;
};

class Wall : public MapSite {
public:
  virtual void Enter(); // No pasa nada
};

class Door : public MapSite {
  Room* _room1;
  Room* _room2;
  bool _isOpen;
public:
  Door(Room* = 0, Room* = 0);
  virtual void Enter(); // Entra si est  abierta
  Room* OtherSideFrom(Room*);
};

class Room : public MapSite {
  MapSite* _sides[4];
  int _roomNumber;
public:
  Room(int roomNo);
  MapSite* GetSide(Direction) const;
  void SetSide(Direction, MapSite*);
  virtual void Enter();
};

enum Direction { North, South, East, West };

class Maze {
public:
  void AddRoom(Room*);
  Room* RoomNo(int) const;
};
```

La clase MazeGame crea el laberinto con código como este:

```
Maze* MazeGame::CreateMaze () {
  Maze* aMaze = new Maze;
  Room* r1 = new Room(1);
  Room* r2 = new Room(2);
  Door* theDoor = new Door(r1, r2);

  aMaze->AddRoom(r1);
  aMaze->AddRoom(r2);

```

```
  r1->SetSide(North, new Wall);
  r1->SetSide(East, theDoor);
  r1->SetSide(South, new Wall);
  r1->SetSide(West, new Wall);

  r2->SetSide(North, new Wall);
  r2->SetSide(East, new Wall);
  r2->SetSide(South, new Wall);
  r2->SetSide(West, theDoor);

  return aMaze;
}
```

Este enfoque es rígido: si queremos cambiar los componentes (por ejemplo EnchantedRoom o DoorNeedingSpell), hay que modificar el método completo. Esto impide reutilizar el código y dificulta el mantenimiento.
El principal obstáculo para extender o modificar la creación de objetos está en el uso directo de clases concretas mediante constructores (new). Los patrones creacionales abordan este problema con distintas estrategias para desacoplar la lógica de construcción del tipo concreto de los objetos:

Factory Method

La creación de objetos se delega a métodos virtuales. En lugar de usar constructores directos dentro del método CreateMaze, se definen métodos como makeRoom, makeWall, etc., que pueden ser redefinidos en subclases. Permite variar las clases creadas mediante herencia.

Abstract Factory

CreateMaze recibe como parámetro un objeto fábrica (factory object) que conoce cómo crear los distintos tipos de componentes (habitaciones, puertas, paredes). Permite cambiar el tipo de objetos creados sin modificar el método consumidor, simplemente usando otra fábrica.

Builder

Se pasa un objeto builder a CreateMaze que se encarga de construir el laberinto completo paso a paso. El cliente llama a operaciones como "agregar habitación" o "conectar puertas", sin saber qué clases concretas se usan. Se desacopla completamente la construcción del producto de su representación.

Prototype

CreateMaze trabaja con instancias prototipo de habitación, puerta y pared. En lugar de construir nuevos objetos desde cero, copia (clona) los prototipos. Permite modificar la estructura simplemente reemplazando los prototipos originales.

Singleton

Garantiza que haya una única instancia compartida del laberinto (u otro objeto global) accesible desde todo el sistema sin necesidad de variables globales. Facilita el acceso centralizado al objeto y asegura su unicidad, además de permitir reemplazo sin cambiar código cliente.
Cada uno de estos patrones ofrece una forma distinta de eliminar la dependencia directa de clases concretas y facilita la extensión, configuración y reutilización del código. La elección depende del grado de flexibilidad necesario y de cómo se desea organizar la responsabilidad de creación.

1.2 Structural Patterns

Los **patrones estructurales** se enfocan en cómo se componen clases y objetos para formar estructuras más grandes y flexibles.

Patrones estructurales de clase

Usan **herencia** para combinar interfaces o implementaciones.

- Por ejemplo, la **herencia múltiple** permite mezclar varias clases en una.
- El patrón **Adapter** (en su forma basada en clase) hereda de una clase existente (*adaptee*) y adapta su interfaz a otra requerida.

Estos patrones son útiles para integrar bibliotecas desarrolladas independientemente.

Patrones estructurales de objeto

Usan **composición de objetos** para extender comportamientos. A diferencia de la herencia, permiten **cambiar la estructura en tiempo de ejecución**.

Ejemplos principales:

- **Composite**: permite construir estructuras jerárquicas con objetos *primitivos* y *compuestos* tratados uniformemente.
- **Proxy**: actúa como intermediario de otro objeto. Puede:
  - representar un objeto remoto,
  - diferir su creación (*lazy loading*),
  - controlar el acceso (seguridad, logging, etc.).
- **Flyweight**: comparte objetos para reducir el uso de memoria. Los objetos compartidos no tienen *estado dependiente del contexto* y se les pasa la información en cada uso.
- **Facade**: representa todo un *subsistema* mediante una única interfaz simplificada. Oculta la complejidad interna del sistema para facilitar su uso.
- **Bridge**: separa una **abstracción** de su **implementación**, permitiendo modificarlas de forma independiente.
- **Decorator**: permite agregar responsabilidades a objetos de forma dinámica. Se encadenan decoradores recursivamente, todos cumpliendo la misma interfaz que el componente base. Ejemplo: en una interfaz gráfica, un componente puede ser decorado con bordes, sombras, scroll, etc.

Los patrones estructurales ayudan a construir sistemas más **flexibles**, **reutilizables** y **mantenibles**, al separar la *estructura* del *comportamiento*, y permitir variaciones en tiempo de ejecución. Muchos de estos patrones están relacionados entre sí y pueden combinarse según las necesidades del diseño.

1.3 Behavioral Patterns

Los **patrones de comportamiento** se enfocan en los *algoritmos* y en la *asignación de responsabilidades* entre objetos. Describen no solo patrones de clases u objetos, sino también las formas en que estos **se comunican entre sí**. Ayudan a gestionar flujos de control complejos y a enfocarse más en las interacciones entre objetos que en el control explícito del flujo.

Patrones de comportamiento de clase

Utilizan **herencia** para distribuir el comportamiento entre clases.

- **Template Method**: define la estructura general de un algoritmo en una clase base, dejando que las subclases implementen los pasos concretos. Es un patrón común para reutilizar lógica general y personalizar detalles.
- **Interpreter**: representa una gramática como una jerarquía de clases, donde cada clase representa una regla del lenguaje. Se implementa un *intérprete* como operación sobre instancias de esas clases.

Patrones de comportamiento de objeto

Usan **composición de objetos** para distribuir responsabilidades dinámicamente. Varios patrones abordan cómo colaborar entre objetos sin acoplamiento fuerte.

- **Mediator**: introduce un objeto mediador que centraliza la comunicación entre objetos pares (peers), evitando referencias directas entre ellos. Favorece el *acoplamiento débil*.
- **Chain of Responsibility**: permite enviar una solicitud a través de una *cadena de objetos* candidatos. Cada objeto decide en tiempo de ejecución si maneja la solicitud o la pasa al siguiente.
- **Observer**: define una dependencia entre objetos para que uno (el sujeto) notifique a otros (observadores) cuando cambia su estado. Ejemplo clásico: el patrón Model/View/Controller de Smalltalk.
- **Strategy**: encapsula un *algoritmo* dentro de un objeto, permitiendo elegir o cambiar dinámicamente el algoritmo que usa un cliente.
- **Command**: encapsula una *solicitud* como objeto. Esto permite pasarla como parámetro, almacenarla, deshacerla o reenviarla.

- **State**: encapsula los *estados internos* de un objeto. El objeto cambia su comportamiento según el estado activo actual.
- **Visitor**: encapsula operaciones sobre estructuras de objetos, evitando que esas operaciones queden distribuidas en múltiples clases. Permite definir nuevas operaciones sin modificar las clases existentes.
- **Iterator**: proporciona una interfaz uniforme para acceder secuencialmente a los elementos de un agregado (por ejemplo, una colección) sin exponer su estructura interna.

Los patrones de comportamiento ayudan a estructurar el flujo de control y la colaboración entre objetos. Permiten definir algoritmos reutilizables, separar decisiones de implementación, y lograr un diseño más modular, extensible y flexible en cuanto a cómo los objetos interactúan y responden a eventos en tiempo de ejecución.