Metodología de Sistemas II

Clase 4

Patrones de Diseño: Creacionales

Jeremías Fassi

2025

Patrones de Diseño: Creacionales

Introducción a los Patrones Creacionales

Jeremías Fassi

2025

Introducción a los patrones creacionales

- ¿Qué son los patrones creacionales y por qué son importantes?
- Cumplen un rol importante en el ciclo de vida de los objetos: **controlan cómo se crean**.
- Beneficios:
  - Reducen acoplamiento en la creación de objetos.
  - Promueven la reutilización de código.
  - Facilitan cambios futuros en la lógica de construcción.

Introducción a los patrones creacionales

- ¿Qué son los patrones creacionales?

  Los patrones creacionales son un conjunto de técnicas que definen cómo se instancian los objetos en un sistema.
  No se limitan a la construcción básica con `new` (o su equivalente en cada lenguaje), sino que aportan formas estructuradas y flexibles de crear instancias, evitando dependencias rígidas en el código.

- En otras palabras:
  - No solo crean objetos, **crean las formas de crearlos**.

Introducción a los patrones creacionales

- Rol en el ciclo de vida de los objetos

  Se aplican en la fase de creación del ciclo de vida del objeto.

  Encapsulan la lógica de construcción, haciendo que el resto del sistema no dependa de cómo se crean los objetos.

  Permiten introducir cambios sin romper el código cliente.

Introducción a los patrones creacionales

Ejemplo comparativo:

- Sin patrón: el código cliente decide cómo se crea un objeto.
- Con patrón: el cliente pide un objeto, pero la lógica de construcción está abstraída (Factory, Builder, etc.).

Introducción a los patrones creacionales - Ejemplo simple

python

```python
class Database:
    def __init__(self, url):
        self.url = url


# El cliente decide cómo se crea el objeto
db = Database("postgres://localhost:5432")
```

Introducción a los patrones creacionales - Ejemplo simple

```ts
class Database {
  constructor(public url: string) {}
}

// El cliente decide cómo se crea el objeto
const db = new Database("postgres://localhost:5432");
```

Con un patrón creacional (ej. Singleton), esa lógica se abstrae, evitando que cada parte del sistema repita la creación.

Introducción a los patrones creacionales
Beneficios principales

- **Reducción del acoplamiento**:
  El código cliente no conoce los detalles internos de la creación.
- **Reutilización del código**:
  Se centraliza y estandariza la lógica de instanciación.
- **Flexibilidad y escalabilidad**:
  Facilitan cambios en el futuro (agregar nuevas clases, cambiar la forma de construir un objeto, etc.) sin modificar el código existente.
- **Claridad conceptual**:
  El nombre del patrón comunica la intención: Singleton (única instancia), Factory (fábrica de objetos), Builder (constructor paso a paso).

Patrón Singleton

Jeremías Fassi

2025

Concepto

El patrón **Singleton** garantiza que una clase tenga una única instancia en todo el sistema y provea un punto de acceso global a esa instancia.

Desactiva todos los demás métodos para crear objetos de una clase, excepto el método de creación especial. Este método crea un nuevo objeto o devuelve uno existente si ya se ha creado.

- Es útil cuando se necesita coordinación centralizada o un recurso compartido.
- Ejemplo clásico: conexiones a bases de datos, gestores de logs, configuración global.

Características principales

- **Una única instancia**: la clase controla que solo exista un objeto creado.
- **Acceso global**: el resto del sistema usa un método estático (o similar) para obtener la instancia.
- **Control centralizado**: se concentra en un punto la gestión de un recurso.

Singleton

- Ventajas
    - Evita la creación múltiple de objetos pesados (ej. DB connection).
    - Centraliza la configuración y el estado.
    - Fácil de implementar.
- Desventajas
    - Si se abusa, puede convertirse en anti-patrón (estado global oculto).
    - Dificulta el testeo unitario (acoplamiento).
    - Puede ser una “puerta trasera” para romper la inmutabilidad del sistema.

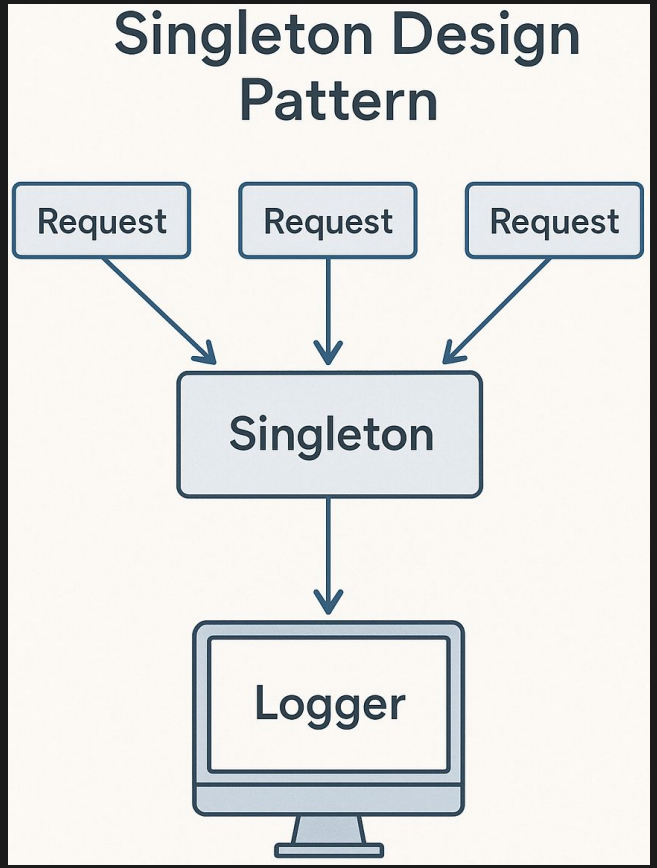
Singleton - Ejemplo

- Python
  - singleton.py
- TS
  - singleton.ts

Casos de uso típicos

- Configuración global: acceder a parámetros de forma unificada.
- Conexión a base de datos: evitar múltiples conexiones innecesarias.
- Gestor de logs: centralizar los mensajes en una sola instancia.

Singleton – Diagrama



Patrón Factory

Jeremías Fassi

2025

Concepto

El patrón **Factory Method** propone delegar la creación de objetos a un método especializado, en lugar de usar directamente el constructor (`new`).

- La idea es que el código cliente no sepa qué clase concreta está instanciando, sino que trabaje con interfaces o clases base.
- Su intención es ofrecer una interfaz para crear objetos en una *superclase*, pero permitiendo que las subclases modifiquen el tipo de objeto que se creará.

Características principales

- **Encapsula la creación de objetos**: el cliente no necesita usar `new`.
- **Favorece la abstracción**: el cliente trabaja con una interfaz común.
- **Facilita la extensión**: si aparecen nuevas clases concretas, basta con extender la Factory sin romper el código cliente.

Factory

- Ventajas
  - Aísla el código cliente de los detalles de instanciación.
  - Permite cambiar o agregar clases sin modificar el cliente.
  - Hace que el sistema sea más flexible y escalable.
- Desventajas
  - Introduce más clases/métodos → puede parecer “sobrecarga” en sistemas simples.
  - Si no se diseña bien, se puede terminar con una “God Factory” que decide demasiado.

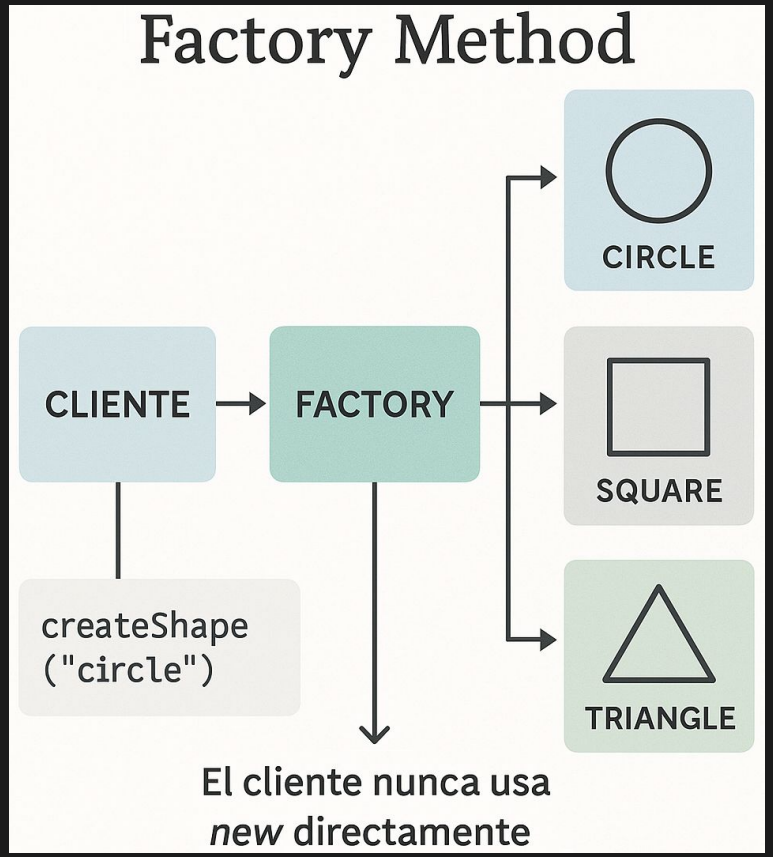
Factory - Ejemplos

- Python
  - factory.py
- TS
  - factory.ts

Casos de uso típicos

- Creación de distintos tipos de usuarios en un sistema (Admin, Guest, Member).
- Generación de objetos según parámetros de configuración o datos de entrada.
- Frameworks que deben instanciar clases sin conocerlas de antemano.

Factory - Diagrama



Patrón Builder

Jeremías Fassi

2025

Concepto

El patrón **Builder** se usa para construir objetos complejos paso a paso, separando:

- La lógica de construcción (qué partes se agregan y en qué orden).
- Del objeto final que se obtiene.

La clave es que el mismo proceso de construcción puede generar diferentes representaciones del objeto.

Características principales

- **Separación de responsabilidades**: la construcción se abstrae en un Builder, no en el cliente.
- **Flexibilidad**: permite armar objetos con muchas combinaciones de parámetros opcionales.
- **Reutilización**: se puede tener distintos “constructores” (builders) para distintos tipos de representación.

Builder

- Ventajas
    - Maneja bien objetos con muchos parámetros opcionales.
    - Hace que el código sea más legible y mantenible que un constructor gigante.
    - Permite crear diferentes configuraciones sin duplicar código.
- Desventajas
    - Introduce más clases (el Builder y a veces un Director).
    - Puede ser más complejo de lo necesario para objetos simples.

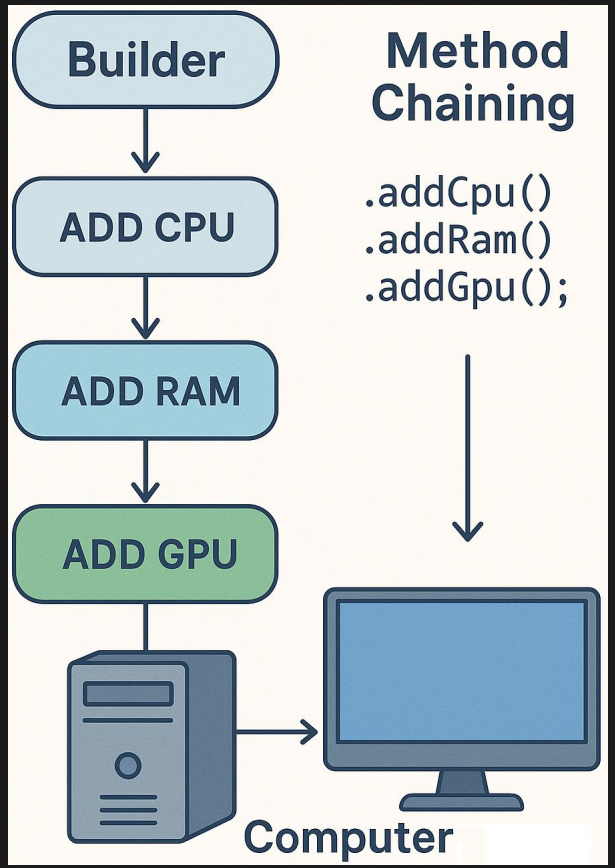
Builder - Ejemplos

- Python
  - builder.py
- TS
  - builder.ts

Casos de uso típicos

- Objetos con muchos parámetros opcionales:
  - Ej: Documentos con título, autor, contenido, formato, etc.
- Distintas representaciones del mismo objeto:
  - Ej: construir un PDF, un HTML y un Markdown a partir de los mismos datos.
- Frameworks de UI o juegos: objetos de configuración (ventanas, escenas, personajes).

Builder – Diagrama



Comparación de Patrones Creacionales

Patrón	Concepto	Cuándo	Ventajas	Desventajas
Singleton	Garantiza que exista una única instancia de una clase y acceso global	Cuando se necesita un recurso compartido: configuración, conexión DB, logs	✔ Control centralizado ✔ Evita múltiples instancias ✔ Fácil de implementar	❌ Puede convertirse en estado global oculto ❌ Dificulta el testeo ❌ Puede romper inmutabilidad
Factory	Define una interfaz de creación y delega la decisión de qué clase instanciar	Cuando el cliente debe trabajar con interfaces y no con clases concretas	✔ Desacopla el código cliente ✔ Fácil de extender (nuevos tipos) ✔ Centraliza la creación	❌ Más clases y métodos ❌ Una "God Factory" puede volverse un antipatrón
Builder	Construye objetos complejos paso a paso separando construcción y representación	Cuando hay objetos con muchos parámetros opcionales o distintas representaciones	✔ Maneja bien objetos complejos ✔ Código legible (method chaining) ✔ Flexibilidad	❌ Sobrecarga innecesaria en objetos simples ❌ Aumenta número de clases

Comparación de Patrones Creacionales

Resumen intuitivo

- Singleton → “Siempre el mismo objeto”.
- Factory → “Decide qué objeto crear según lo pedido”.
- Builder → “Arma el objeto de a pasos, combinando piezas”.

Otros patrones creacionales

- **Abstract Factory**
  - Idea principal: provee una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.
  - Ejemplo típico: librería de interfaces gráficas → una fábrica puede crear botones, ventanas y menús para Windows, y otra para MacOS, pero el cliente trabaja solo con la interfaz.
  - Cuándo usarlo: cuando querés asegurarte de que un conjunto de objetos coincidan entre sí en estilo o comportamiento.

→ Permite cambiar “de un golpe” toda una familia de productos.

Otros patrones creacionales

- **Prototype**
  - Idea principal: en lugar de crear un objeto desde cero, se clona un objeto existente (prototipo).
  - Ejemplo típico: un editor gráfico → duplicar figuras con todos sus atributos (color, posición, estilo) sin tener que reconfigurar todo.
  - Cuándo usarlo: cuando la creación de un objeto es costosa o compleja, y resulta más eficiente copiar un objeto ya existente.

→ Muy útil para evitar constructores pesados.

Recursos:

- Patrones Creacionales

- DigitalOcean Community – GoF Patterns Explained

Muchas Gracias

Jeremías Fassi

Javier Kinter