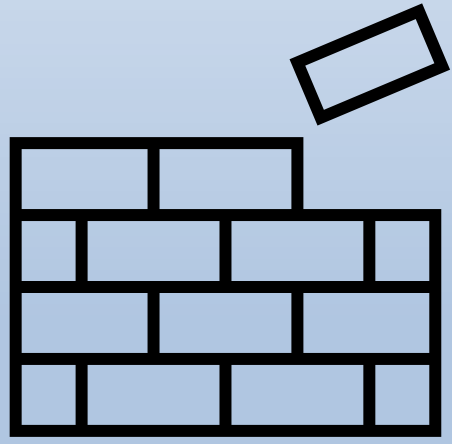
Patrones de diseño

Patrones Creacionales



Fernando Díaz-Alonso Dorado

Patrones creacionales

¿Qué son?

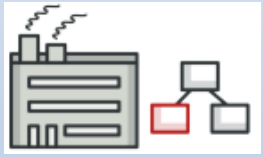
Los patrones creacionales son patrones que nos proporcionan varios mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.

Patrones creacionales

Principales patrones creacionales

Algunos patrones creacionales son:

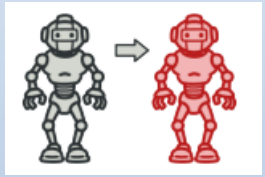
- Factory Method



- Abstract Factory

- Builder

- Prototype



- Singleton

Factory Method

Método fábrica, constructor virtual.

Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Aplicabilidad:

- Se aplica cuando no se conoce de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tú código.
- Cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos.
- Cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.

Factory Method

Método fábrica, constructor virtual II

Pros y contras:

- ✓ Evitas un acoplamiento fuerte entre el creador y los productos concretos.
- ✓ *Principio de responsabilidad única*. Puedes mover el código de creación de producto a un lugar del programa, haciendo que el código sea más fácil de mantener.
- ✓ *Principio de abierto/cerrado.* Puedes incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.
- ✗ Puede ser que el código se complique, ya que debes incorporar una multitud de nuevas subclases para implementar el patrón. La situación ideal sería introducir el patrón en una jerarquía existente de clases creadoras.

Abstract Factory

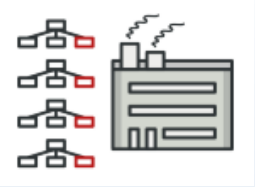
Fábrica abstracta

Es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

Aplicabilidad:

- Se aplica cuando tu código deba funcionar con varias familias de productos relacionados, pero no desees que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad.
- Cuando tengas una clase con un grupo de *Factory Method* que nublen su responsabilidad principal.

Abstract Factory



Fábrica abstracta II

Pros y contras:

- ✓ Puedes tener la certeza de que los productos que obtienes de una fábrica son compatibles entre sí.
- ✓ Evitas un acoplamiento fuerte entre productos concretos y el código cliente.
- ✓ *Principio de responsabilidad única*. Puedes mover el código de creación de productos a un solo lugar, haciendo que el código sea más fácil de mantener.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente.
- ✗ Puede ser que el código se complique más de lo que debería, ya que se introducen muchas nuevas interfaces y clases junto al patrón.

Builder

Constructor

Es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

Aplicabilidad:

- Se aplica para evitar un “constructor telescópico”.
- Cuando quieras que el código sea capaz de crear distintas representaciones de ciertos productos (por ejemplo, casas de piedra y madera).
- Para construir árboles con el patrón *Composite* y otros objetos complejos.

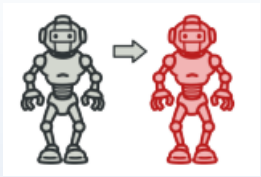
Builder

Constructor II

Pros y contras:

- ✓ Puedes construir objetos paso a paso, aplazar pasos de la construcción o ejecutar pasos de forma recursiva.
- ✓ Puedes reutilizar el mismo código de construcción al construir varias representaciones de productos.
- ✓ *Principio de responsabilidad única*. Puedes aislar un código de construcción complejo de la lógica de negocio del producto.
- ✗ La complejidad general del código aumenta, ya que el patrón exige la creación de varias clases nuevas.

Prototype



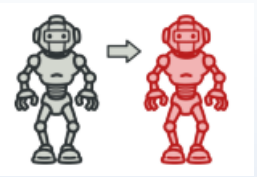
Prototipo, Clon, Clone

Es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

Aplicabilidad:

- Cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.
- Cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.

Prototype



Prototipo, Clon, Clone II

Pros y contras:

- ✓ Puedes clonar objetos sin acoplarlos a sus clases concretas.
- ✓ Puedes evitar un código de inicialización repetido clonando prototipos prefabricados.
- ✓ Puedes crear objetos complejos con más facilidad.
- ✓ Obtienes una alternativa a la herencia al tratar con preajustes de configuración para objetos complejos.
- ✗ Clonar objetos complejos con referencias circulares puede resultar complicado.

Singleton

Instancia Única

Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Aplicabilidad:

- Cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.
- Cuando necesites un control más estricto de las variables globales.

Singleton

Instancia Única II

Pros:

- ✓ Puedes tener la certeza de que una clase tiene una única instancia.
- ✓ Obtienes un punto de acceso global a dicha instancia.
- ✓ El objeto Singleton solo se inicializa cuando se requiere por primera vez.

Singleton

Instancia Única III

Contras:

- Vulnera el *Principio de responsabilidad única*. El patrón resuelve dos problemas al mismo tiempo.
- El patrón Singleton puede enmascarar un mal diseño, por ejemplo, cuando los componentes del programa saben demasiado los unos sobre los otros.
- El patrón requiere de un tratamiento especial en un entorno con múltiples hilos de ejecución, para que varios hilos no creen un objeto Singleton varias veces.

Singleton

Instancia Única IV

Contras:

- Puede resultar complicado realizar la prueba unitaria del código cliente del Singleton porque muchos frameworks de prueba dependen de la herencia a la hora de crear objetos simulados (mock objects).Debido a que la clase Singleton es privada y en la mayoría de los lenguajes resulta imposible sobrescribir métodos estáticos, tendrás que pensar en una manera original de simular el Singleton. O, simplemente, no escribas las pruebas. O no utilices el patrón Singleton

Principios del diseño de software

Bibliografía

- Patrones de Diseño – Alexander Shvets