Patrones de Diseño Creacionales

Serie de Diseño de Software

Me llamo Daniel

Vivo en Yucatán, México
y escribo código. 💻

¿Alguien dijo diseño de software?

Te escuchamos y tenemos grandes planes para ti:

Patrones creacionales

Abstraen el proceso de creación/instanciación de objetos:

1. Singleton
2. Factory
3. Abstract factory
4. Builder
5. Prototype

Singleton

¿De qué se trata Singleton?👀

Patrón de diseño que nos permite asegurarnos **que no se pueda crear más de una instancia de un objeto**.

Con esto aseguramos un único punto global de acceso a la instancia.

¿Qué problema hay? 😱

1. Queremos asegurar el acceso a un recurso compartido en diferentes partes de la app.
2. Asegurar que la modificación al recurso compartido se lleve a cabo en un solo punto de acceso.

Solución 😎

1. El patrón sugiere hacer privado el constructor de la clase para evitar que sea usado al utilizar el operador *new()*
2. Crear un método estático que actúe como “constructor” y que tras bambalinas llame al constructor privado para crear un objeto que estará guardado en una variable estática que funcionará como caché.

Implementemos Singleton en JS

Pros y cons

Cosas cool 👍

- **Certeza** de que solo existirá una sola instancia de una clase.
- **Un solo punto de acceso global** a dicha instancia.
- La instancia es **inicializada solo cuando se requiere** por primera vez.

Cosas no tan cool 👎

- ✖ **Vulnera el Principio de Responsabilidad Única.** El patrón resuelve dos problemas al mismo tiempo.
- ✖ **Complejidad incrementada en ambientes de múltiples hilos de ejecución.** ¿Cómo hacer que muchos hilos no creen un objeto Singleton múltiples veces?
- ✖ **Complejidad a la hora de crear pruebas unitarias** debido al uso de elementos estáticos.

¿Cuándo usarlo? ☝️

1. Úsalo cuando requieras que exista un **único punto de acceso** a los recursos compartidos y que pueda ser usado por diversos clientes.
2. ¿Para conexiones a bases de datos?

Ejemplos: bibliotecas de manejo del estado.

Factory Method

¿De qué va Factory Method? 👀

Patrón de diseño que nos provee de una **interfaz para crear objetos basados en una superclase**, posibilitando que las subclases creadoras alteren el tipo de objetos a retornar en su proceso de fabricación.

¡Fabriquemos coches! 🚙

1. Creamos una aplicación que maneja la producción de coches. La primera versión solo considera el modelo **Mastodon**, todo el código está acoplado a él.

2. La app es un éxito, nos piden agregar un nuevo modelo, el **Rhino**.

¡Fabriquemos coches! 🚙

3. Para añadirlo hay que modificar el código base, es decir, **muchas, muchas, muchas** líneas de código.

4. Condicionales por todos lados...

Solución 😎

1. El patrón sugiere que en lugar de usar el operador *new()* se invoque a un método **fábrica** que se encargue de la creación de los objetos. Estos objetos son llamados **productos**.

2. Internamente, este método seguirá usando el operador *new()*.

Solución 😎

3. Las superclases **fábrica**, estarán basadas **en una clase/interfaz común**, esto nos permite intercambiarlas según sea requerido.

4. Los **productos** retornados por las **fábricas** deben estar basados **en una clase base o una interfaz.**

Código

Pros y cons

Cosas cool 👍

- ✔ **Evitamos** un acoplamiento alto entre los elementos creadores y los productos.
- ✔ La creación de productos sucede en **un único punto.**
- ✔ Agregar nuevos productos no requiere modificar el código existente, **lo extiende**.

Cosas no tan cool 👎

❌ Demasiado **código genérico**, nuevo producto, nueva fábrica.

❌ ¡**Muchas** abstracciones!

¿Cuándo usarlo? ☝️

1. Úsalo cuando no sabes cuántos **productos diferentes** habrá.
2. Cuando necesites **desacoplar** el uso de los productos con su creación.
3. **Extender** el funcionamiento de bibliotecas o frameworks.

Ejemplos: Bibliotecas de UI, Frameworks para Web

Abstract Factory

¿De qué va Abstract Factory? 👀

Patrón de diseño que nos provee de una estrategia para **encapsular múltiples fábricas** de diferentes productos bajo **una sola familia** sin especificar clases concretas.

¡Nuevas versiones de coches! 

1. Creamos una aplicación que maneja la producción de coches. Hasta ahora solo producimos dos modelos, **cada uno en su fábrica** y sin importar la **versión**.

2. El negocio es un exito, por lo que nos piden **agregar familias de versiones** por cada coche: la sedán y la hatchback.

¡Nuevas versiones de coches! 

3. Los directivos también decidieron que la producción de cada versión se llevará a cabo en una **fábrica particular.** Es decir, tendremos la fábrica de coches sedán y la fábrica de coches hatchback.

4. El patrón factory no parece funcionar ya que crearemos **más de un producto por fabrica!**

Solución 😎

1. El patrón sugiere que declaremos clases base/interfaces **por cada uno de los productos en el catálogo** (Mastodon, Rhino).

2. Implementar clases concretas de los productos **por cada una de las familias de versiones** (RhinoSedan, MastodonSedan).

Solución 😎

3. Declarar la clase base/interfaz Abstract Factory, que declare métodos de creación **por cada uno de los productos en el catálogo**.

4. Crear clases fábrica concretas **por cada una de las familias de versiones** (SedanFactory, HatchbackFactory) que implementen los métodos de creación.

Código

Pros y cons

Cosas cool 👍

- ✔ **Evitamos** un acoplamiento alto entre los elementos creadores y los productos.
- ✔ Los productos retornados por las fábricas son **intercambiables**.
- ✔ La creación de los productos sucede en un **único** punto.
- ✔ Agregar nuevos productos no requiere modificar el código existente, **lo extiende**.

Cosas no tan cool 👎

- ✖ Demasiado **código genérico**, nueva versión, nuevos productos, nuevo método de creación.

- ✖ Si los productos base agregan algún elemento, **todos los productos** concretos deben de implementar el cambio.

- ✖ **Depende** de que existan familias de productos.

¿Cuándo usarlo? ☝️

1. Úsalo cuando tengamos **variantes de los productos base** que compartan similitudes y puedan ser agrupados en familias.
2. Cuando queramos que los productos nuevos se **integren** con los ya existentes.

Ejemplos: Implementación de un desing system.

Builder

¿De qué va Builder? 👀

Patrón de diseño que nos permite **dividir la creación de un objeto en pasos.** Utilizando el mismo proceso de construcción podremos crear **diferentes representaciones** del mismo objeto.

¡Nuevas versiones de coches! 

1. Continuando con el éxito en la producción de coches, aparece un nuevo requerimiento: los coches sedán tendrán **diferentes versiones en cada familia: *CVT* y *Signature***.

2. En cada versión se pueden modificar **2 diferentes elementos:** el **color** y el número de **bolsas de aire.**

¡Nuevas versiones de coches! 🚙

3. A algunos miembros del equipo se les ocurre usar **Factory** y declarar una clase que contemple **la familia y la versión** (familia + coche + versión).

4. A otro se les ocurre usar **Abstract Factory** y hacer que la fábrica de cada familia regrese cada una de las **versiones por cada coche en el catálogo** (coche + versión).

¡Nuevas versiones de coches! 

5. Al último grupo se les ocurre hacer que los métodos de creación de cada coche reciban **todos los parámetros de personalización** posibles. Generando el problema del **“constructor microscópico”**.

Solución 😎

1. El patrón sugiere que declaremos clase base/interfaz que definirá los **pasos generales de creación del producto** (línea de producción de cada familia).

2. Implementar clases concretas de builders que ofrezcan **diferentes versiones** de los pasos de creación.

Solución 😎

3. Implementar productos concretos que puedan ser **retornados por los builders**. No tienen que seguir una clase base/interfaz.

4. Implementar una clase directora que conocerá **el orden** en el que se llamarán a los **pasos de construcción** de cada configuración (los pasos necesarios para la versión Signature y la CVT).

Código

Pros y cons

Cosas cool 👍

- ✔ Puedes construir objetos **paso a paso**, aplazar algunos de estos o utilizar recursividad.
- ✔ Poder reutilizar el mismo proceso de construcción para construir **diferentes representaciones** de productos.
- ✔ Aislar las configuraciones de construcción en **un solo lugar**.
- ✔ Nuevas configuraciones **no requieren modificar** las existentes.

Cosas no tan cool 👎

❌ Demasiado **código genérico**, nuevo producto, nuevo builder.

❌ **Mutación** del objeto producto.

¿Cuándo usarlo? ☝️

1. Úsalo cuando quieras evitar tener un **constructor telescópico**.
2. Cuando requieras que existan **diferentes representaciones** de algunos productos.
3. Cuando quieras tener **control** sobre el **proceso de creación** de un objeto.

Ejemplos: Construcción de *queries* para bases de datos

Prototype

¿De qué va Prototype? 👀

Patrón de diseño que nos permite **hacer clones de objetos existentes** sin que dependamos de clases concretas.

¡Pruebas de seguridad!

1. ¡Un nuevo requerimiento aparece! Tenemos que realizar **pruebas de impacto** sobre los coches.

2. Estas pruebas no pueden realizarse con los autos que están listos para ser vendidos, por lo que se decidió fabricar **prototipos de prueba**.

Solución 😎

1. El patrón sugiere que deleguemos el proceso de **generación de clones** a los propios objetos. A estos se les llamará **prototipos**.

2. Para esto debemos **declarar una clase base/interfaz** para todos los objetos que soportan el ser clonados.

Pros y cons

Cosas cool 👍

- ✔ Puedes clonar objetos **sin asociarlos** a sus clases concretas.
- ✔ Podemos **ahorrarnos** la creación de muchas sub clases.
- ✔ Podemos **evitar** código de instanciación de objetos repetido.

Cosas no tan cool 👎

❌ Implementar el método de clonación en **todas** las clases.

❌ **Referencias circulares** con objetos compuestos por otros.

¿Cuándo usarlo? ☝️

1. Úsalo cuando quieras **reducir** la cantidad de subclases.
2. Cuando quieras beneficiar la **reusabilidad.**

Ejemplos: Javascript