Patrones Creacionales

MSc. César Augusto López Gallego.

Profesor Facultad Ingeniería en TIC – UPB.

Coordinador Área de Programación, Computación y Desarrollo de Software.

cesar.lopezg@upb.edu.co

El **patrón Factory Method** es un patrón creacional que proporciona una interfaz para crear objetos en una superclase, pero permite a las subclases alterar el tipo de objetos que se crearán.

Situación: Se tiene una app para la gestión logística que solo maneja transporte en camiones, todo el código de la aplicación está acoplado a estas clases.

¿Si nos piden incluir transporte en barcos?

Sería un cambio mayor para todo el código.

Y si después piden incluir otro medio de transporte? Por ejemplo, aéreo?

Otro cambio mayor…..

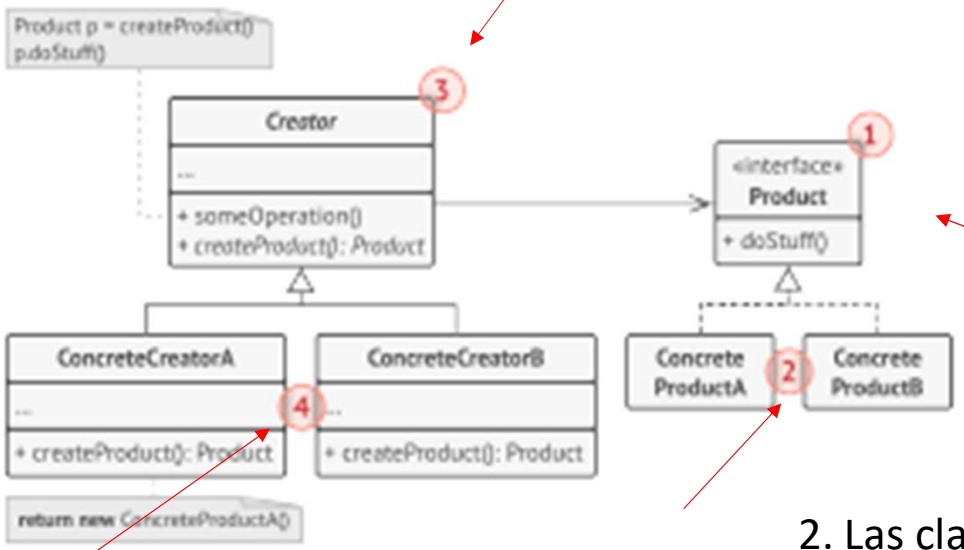
El patrón sugiere que remplace las llamadas directas de construcción de objetos (usando el operador new) con llamadas a un método de fábrica especial.

Los objetos se siguen creando a través del operador new, pero se le llama desde dentro del método de fábrica. Los objetos devueltos por un método de fábrica se suelen denominar “productos”.

Se podría Sobrescribir el método de fábrica en una subclase y cambiar la clase de productos que se crean mediante el método.

Structure

3. La clase Creator declara **el método de fábrica** **CreateProduct()** que devuelve nuevos objetos de producto. El tipo de retorno de este método debe coincidir con la interfaz del producto.



N1: Puede declarar el método de fábrica como abstracto para obligar a todas las subclases a implementar sus propias versiones del método

N2: La creación del producto no es la responsabilidad principal de Creator.

N3: El método de fábrica ayuda a desacoplar las clases de producto concretas

1. La interfaz declara un producto, que es común a todos los objetos que pueden ser producidos por el Creator y sus subclases.

2. Las clases concretas son diferentes implementaciones de la interfaz del producto.

4. Las implementaciones concretas anulan el método de fábrica base para que devuelva un tipo de producto diferente.

Implementemos

Ventajas de Factory Method

Desacoplamiento:
- Elimina la dependencia directa entre el código cliente y las clases concretas.
- El cliente usa interfaces o clases abstractas sin preocuparse por cómo se crean los objetos reales.

Facilita la Extensión:
- Agregar nuevos tipos de productos (como un nuevo tipo de automóvil) es fácil.
- Solo necesitas crear una nueva fábrica sin modificar el código existente, siguiendo el principio **Open/Closed** (abierto para extensión, cerrado para modificación).

Soporte para Productos Complejos:
- Si la creación de un objeto es compleja o tiene múltiples pasos, el Factory Method ayuda a encapsular esos detalles.

Reutilización de Código:
- Permite reutilizar la lógica de creación en varias partes de la aplicación.
- Cada subclase de fábrica encapsula cómo crear un tipo específico de objeto.

Centraliza la Creación de Objetos:
- La lógica de creación no está dispersa en el código.
- Esto facilita cambios futuros, como modificar el proceso de creación en un solo lugar.

Promueve la Cohesión:
- Cada clase tiene una única responsabilidad: las fábricas crean objetos, los objetos realizan sus propias funciones.
- Esto hace que el diseño sea más **limpio y mantenible**.

Mejora las Pruebas Unitarias:
- Al trabajar con interfaces, puedes **simular** objetos más fácilmente durante las pruebas.

Desventajas de Factory Method

- ✓ Puede añadir complejidad si no se necesita tanta flexibilidad. El código puede volverse más complicado ya que es necesario introducir muchas subclases nuevas para implementar el patrón.

- ✓ Involucra más clases y abstracciones, lo que puede ser excesivo en proyectos muy simples.

¿Cómo lo usa un arquitecto de software profesionalmente?

Un arquitecto de software, al diseñar la estructura de un sistema, define los **puntos de extensión** y las **interfaces** que otros desarrolladores implementarán. Usa el Factory Method para:

- **Establecer un punto central de creación de objetos**, evitando que múltiples partes del sistema creen instancias de clases directamente.

- **Asegurar flexibilidad** al permitir que el sistema evolucione sin modificar grandes cantidades de código.

- **Facilitar la prueba y el mantenimiento** al crear objetos a través de interfaces bien definidas.

¿Cuándo usarlo?

El método Factory separa el código de construcción del producto del código que realmente utiliza el producto. Por lo tanto, es más fácil extender el código de construcción del producto independientemente del resto del código.

Utilice el método Factory cuando no conozca de antemano los tipos y dependencias exactos de los objetos con los que debería trabajar su código.

Por ejemplo, para agregar un nuevo tipo de producto a la aplicación, solo necesitará crear una nueva subclase de creador y anular el método Factory en ella.

Utilice el método Factory cuando desee proporcionar a los usuarios de su biblioteca o marco una forma de extender sus componentes internos.

La solución es reducir el código que construye componentes en todo el marco a un único método de fábrica y permitir que cualquiera reemplace este método además de extender el componente en sí.

Utilice el método de fábrica cuando desee ahorrar recursos del sistema reutilizando objetos existentes en lugar de reconstruirlos cada vez .

Esta necesidad es frecuente cuando al trabajar con objetos grandes que consumen muchos recursos, como conexiones de bases de datos, sistemas de archivos y recursos de red.

¿Cómo funciona en un equipo de desarrollo?

En un equipo, el arquitecto puede decir:

"Todas las clases de vehículos deben crearse usando un método fábrica.

No creen objetos con new directamente.“

Los desarrolladores, al seguir esta guía, implementan clases concretas sin modificar el núcleo del sistema. Así, si mañana el cliente pide un nuevo tipo de automóvil (como un eléctrico), solo crean una nueva fábrica sin tocar el resto del código

Preguntas que un equipo resuelve con Factory Method:

"¿Qué pasa si necesitamos agregar un nuevo tipo de producto sin afectar el código existente?“

Respuesta: Usamos Factory Method para agregar nuevos productos sin modificar el núcleo del sistema.

"¿Cómo podemos asegurarnos de que todos los objetos se crean de manera consistente?“

Respuesta: Implementamos un método fábrica central para manejar la creación de objetos.“

¿Qué hacemos si diferentes partes del sistema necesitan crear objetos, pero con ligeras diferencias?“

Respuesta: Usamos subclases de fábrica para que cada una cree su versión específica del objeto.

"¿Cómo evitamos duplicar código de creación de objetos en diferentes módulos?“

Respuesta: Encapsulamos la lógica de creación en un Factory Method reutilizable.

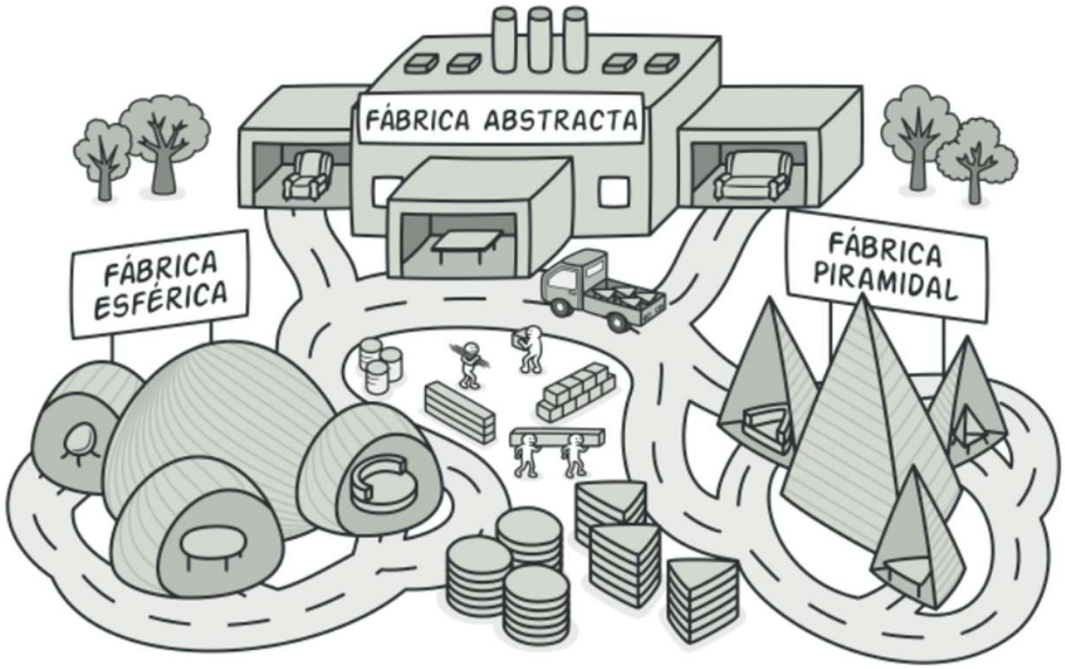
Ejemplos en la vida real del desarrollo de software:

Aplicaciones móviles: Cuando un equipo desarrolla una app multiplataforma (iOS y Android), un arquitecto puede definir un Factory Method para crear botones, menús o notificaciones según la plataforma.

Juegos: Crear diferentes tipos de enemigos, personajes o armas mediante un Factory Method que facilita agregar nuevos elementos al juego sin modificar el código base.

Sistemas financieros: Un arquitecto podría definir un Factory Method para crear objetos de transacciones que se adapten a diferentes bancos o monedas, permitiendo agregar nuevas entidades financieras sin cambiar el sistema central.

Beneficio clave para un equipo:
Un **arquitecto de software** sabe que los requerimientos siempre cambian. Con el **Factory Method**, el equipo está **preparado para el futuro** sin rehacer el sistema entero. Se logra un desarrollo más ágil, flexible y mantenible.



Patrón Abstract Factory

Abstract Factory es un patrón creacional que proporciona una interfaz para producir familias de objetos relacionados o sin especificar sus clases concretas.

Es ideal cuando un sistema debe ser independiente de cómo se crean, componen y representan sus productos.

Armar un carro = Ensamblar la Carrocería + Instalar el Motor

Carrocería Eléctrico

+

Motor Diesel

=

Los objetos no se relacionan

Carrocería Eléctrico

+

Motor Eléctrico

=

Los objetos se relacionan

- El patrón de fábrica abstracto es casi igual que el patrón de fábrica y se considera como otra capa de abstracción sobre el patrón de fábrica.

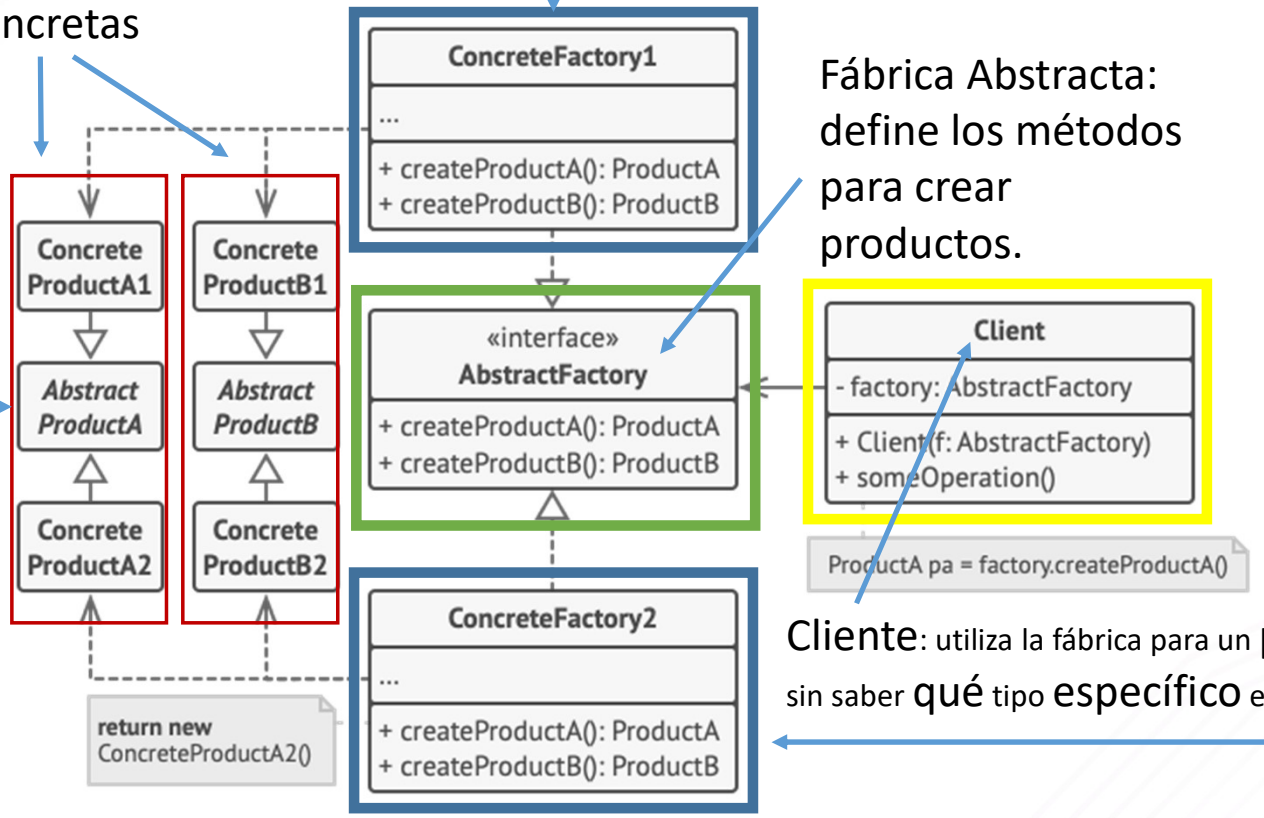
- Los patrones abstractos de fábrica funcionan alrededor de una superfábrica que crea otras fábricas.

- En tiempo de ejecución, la fábrica abstracta se acopla con cualquier fábrica concreta deseada que pueda crear objetos del tipo deseado.

Estructura Fábrica Abstracta

Productos Concretos: Implementaciones concretas

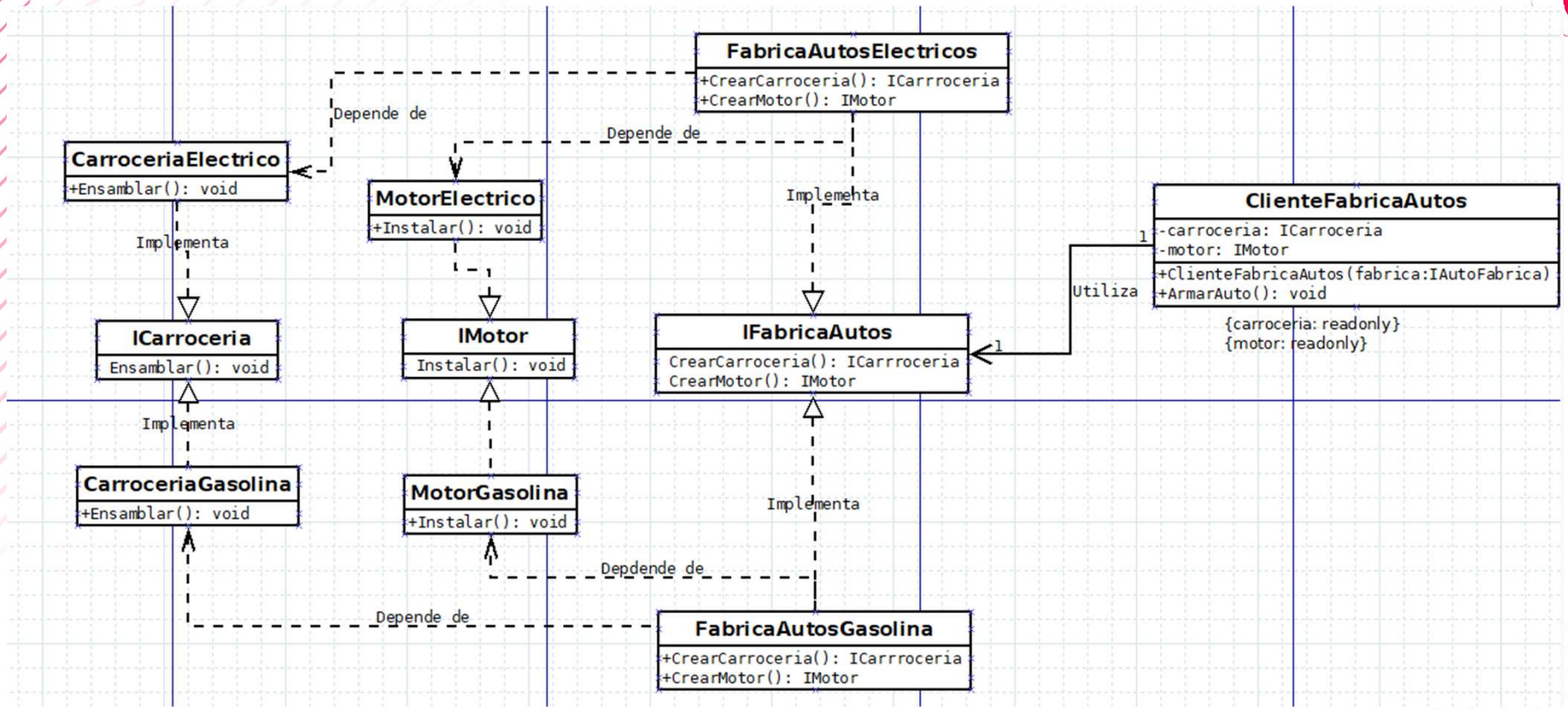
Productos Abstractos: definen las interfaces para los productos.



Fábrica Abstracta: define los métodos para crear productos.

Fábricas Concretas: crean productos de una misma familia.

Cliente: utiliza la fábrica para un producto sin saber qué tipo específico está creando.



Implementemos

Beneficios de utilizar el patrón Abstract Factory

- ✓ El patrón Abstract Factory separa la creación de objetos, por lo que los clientes no necesitan conocer clases específicas.
- ✓ Los clientes interactúan con los objetos a través de interfaces abstractas, manteniendo los nombres de clase ocultos del código del cliente.
- ✓ Cambiar la fábrica permite diferentes configuraciones de producto, ya que todos los productos relacionados cambian juntos.
- ✓ El patrón garantiza que una aplicación utilice objetos de una sola familia a la vez para una mejor compatibilidad.

Cuidados al utilizar el patrón Abstract Factory

- ✓ El patrón Abstract Factory puede agregar complejidad innecesaria a proyectos más simples con múltiples fábricas e interfaces.
- ✓ Agregar nuevos tipos de productos puede requerir cambios tanto en las fábricas concretas como en la interfaz de la fábrica abstracta, lo que afecta el código existente.
- ✓ La introducción de más fábricas y familias de productos puede aumentar rápidamente la cantidad de clases, lo que dificulta la gestión del código en proyectos más pequeños.
- ✓ Puede violar el principio de inversión de dependencia si el código del cliente depende directamente de fábricas concretas en lugar de interfaces abstractas.

Cuándo utilizar el patrón Abstract Factory

- ✓ Cuando el sistema requiere múltiples familias de productos relacionados y desea garantizar la compatibilidad entre ellos.
- ✓ Cuando necesita flexibilidad y extensibilidad, permitiendo agregar nuevas variantes de productos sin cambiar el código del cliente existente.
- ✓ Cuando desea encapsular la lógica de creación, facilitando la modificación o ampliación del proceso de creación de objetos sin afectar al cliente.
- ✓ Cuando se pretende mantener la coherencia entre diferentes familias de productos, se garantiza una interfaz uniforme para los productos.

Cuándo NO utilizar el patrón Abstract Factory

- ✓ Es poco probable que las familias de productos cambien, ya que esto puede agregar una complejidad innecesaria.
- ✓ Cuando su aplicación solo requiere objetos únicos e independientes y no se ocupa de familias de productos relacionados.
- ✓ Cuando los costos adicionales de mantener varias fábricas superan los beneficios, particularmente en aplicaciones más pequeñas.
- ✓ Cuando existen soluciones más simples, como el método Factory o el patrón Builder, satisfacen sus necesidades sin agregar la complejidad del patrón Abstract Factory.

Preguntas que Resuelve un Arquitecto con Abstract Factory

1.¿Cómo desacoplar la creación de objetos del cliente?

- Permite crear objetos sin que el cliente sepa qué implementaciones específicas se están utilizando.

2.¿Cómo asegurar consistencia en productos relacionados?

- Todas las implementaciones de una misma familia son creadas por una sola fábrica.

3.¿Cómo facilitar el mantenimiento y la expansión?

- Agregar una nueva familia de productos (por ejemplo, automóviles híbridos) solo requiere crear nuevas clases y una nueva fábrica, sin modificar el código existente.

4.¿Cómo apoyar el desarrollo ágil y modular?

- Facilita el trabajo en equipos al permitir que diferentes equipos trabajen en diferentes fábricas o productos sin interferir entre sí.

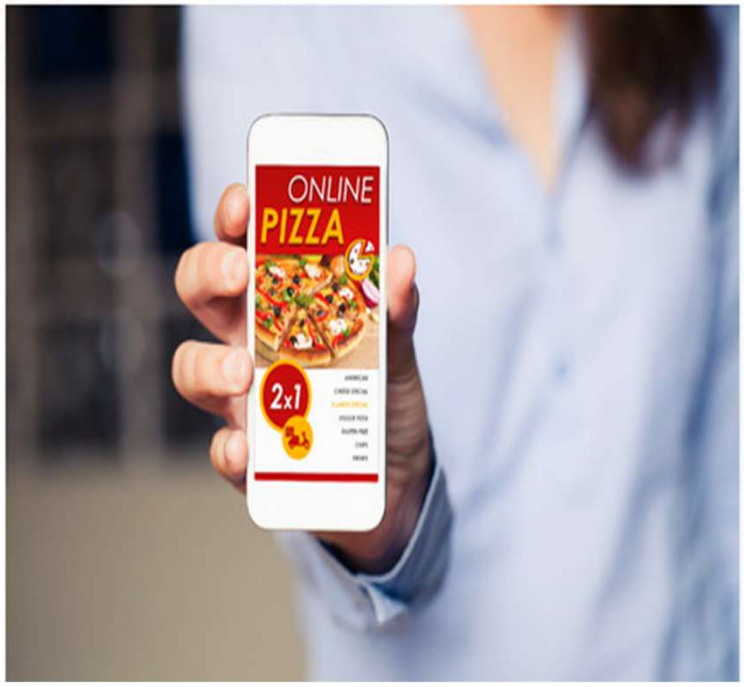
Un arquitecto de software en el plano profesional utiliza el patrón Abstract Factory para:

- **Asegurar la consistencia entre productos relacionados:** Por ejemplo, garantizar que un sistema de automóviles eléctricos no utilice un motor de combustión interna.

- **Facilitar el mantenimiento y escalabilidad:** Permite agregar nuevas variantes de productos (nuevos tipos de automóviles) sin modificar el código existente.

- **Resolver problemas de desacoplamiento:** El arquitecto se asegura de que el sistema no dependa de implementaciones concretas, facilitando los cambios futuros.

Patrón Builder



- Cómo comenzamos a diseñar la arquitectura de una aplicación para una pizzería?
- Qué clases uso?
  - Una para cada pizza
  - Una clase con una cantidad de constructores sobrecargados
- Herencia, Polimorfismo?

El proceso de construcción depende del producto

Patrón Builder

Builder Design es un patrón creacional que permite construir objetos complejos paso a paso, donde el proceso de construcción puede cambiar en función del tipo de producto que se esté construyendo.

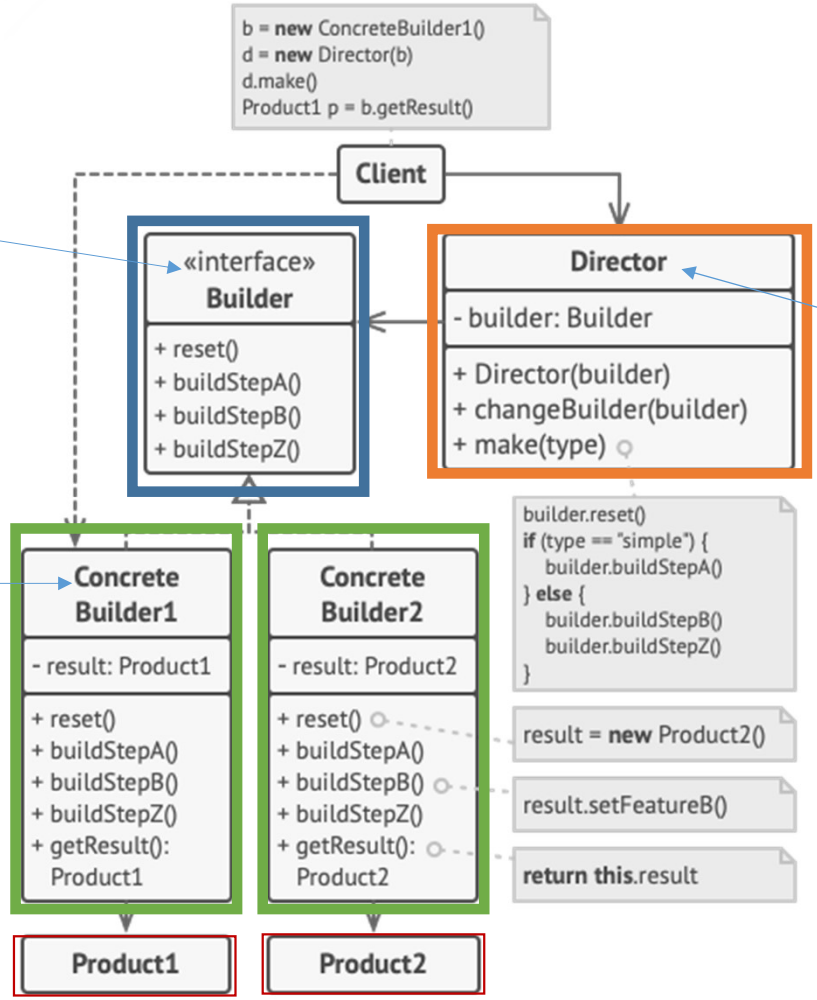
Estructura Builder

Interfaz Builder: Define los pasos para construir el producto

Director: Define la secuencia de construcción de diferentes tipos de productos.

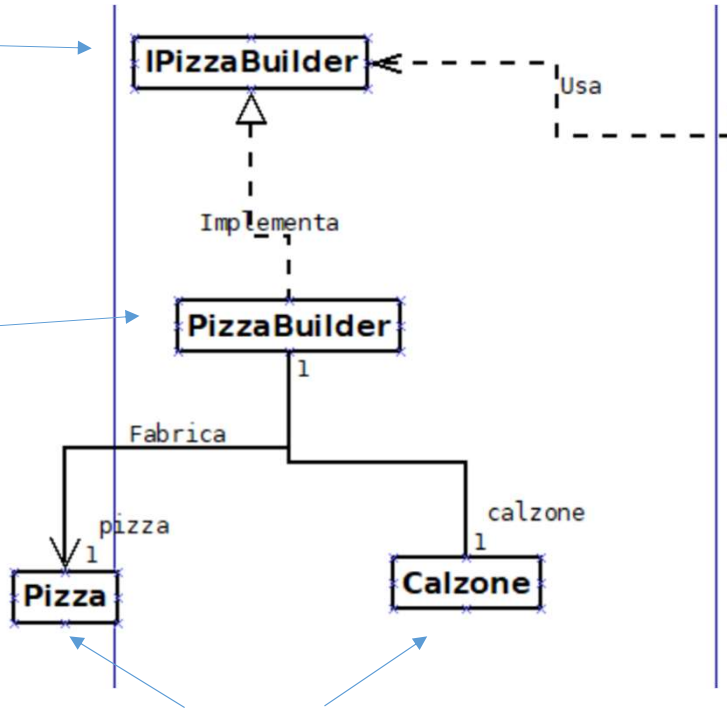
Constructor Concreto: Es la clase concreta que implementa la interfaz

Producto: Representa el objeto que se está construyendo.



Implementación de la Estructura Builder

Interfaz Builder



Implementemos

Cuándo se usa el patrón Builder

- **Construcción de objetos complejos:** cuando se tiene un objeto con muchos componentes o configuraciones opcionales y deseas proporcionar una separación clara entre el proceso de construcción y la representación real del objeto.
- **Construcción paso a paso:** cuando la construcción de un objeto implica un proceso paso a paso donde es necesario establecer diferentes configuraciones u opciones en diferentes etapas.
- **Cómo evitar constructores con múltiples parámetros:** cuando la cantidad de parámetros en un constructor se vuelve demasiado grande y el uso de constructores telescópicos (constructores con múltiples parámetros) se vuelve difícil de manejar y propenso a errores.
- **Creación de objetos configurables:** cuando necesita crear objetos con diferentes configuraciones o variaciones y desea una forma más flexible y legible de especificar estas configuraciones.
- **Interfaz común para múltiples representaciones:** cuando desea proporcionar una interfaz común para construir diferentes representaciones de un objeto.

Cuándo NO se usa el patrón Builder

Construcción de objetos simples: No usar para objetos con pocos parámetros o configuraciones simples

Preocupaciones sobre el rendimiento: La sobrecarga adicional que introduce el patrón Builder puede ser un problema en aplicaciones con rendimiento crítico. Puede afectar el rendimiento si la construcción de objetos es frecuente.

Objetos inmutables con campos final o const: No usar si va a trabajar con un lenguaje que admite objetos inmutables con campos finales (por ejemplo, final o const) y la estructura del objeto es relativamente simple

Cuando se quiere evitar introducir mayor complejidad del código: La introducción de una clase constructora para cada objeto complejo puede generar un aumento en la complejidad del código.

Acoplamiento estrecho con el producto: Si el constructor está estrechamente vinculado al producto que construye, y los cambios en el producto requieren modificaciones correspondientes en el constructor, podría reducir la flexibilidad y la capacidad de mantenimiento del código.

Preguntas que Resuelve un Arquitecto con el patrón Builder

¿Cómo simplificar la construcción de objetos con múltiples parámetros?
Se evita el uso de constructores largos con demasiados parámetros.

¿Cómo hacer que la construcción de objetos sea más flexible?
Se pueden crear distintas configuraciones de un objeto sin modificar su estructura.

¿Cómo desacoplar el proceso de construcción de la representación final del objeto?
El director define los pasos de construcción sin saber detalles de la implementación específica.

¿Cómo hacer que el código sea más mantenible y extensible?
Si se agregan nuevas opciones, solo se modifican los métodos del Builder, sin afectar el cliente.

¿Cómo permitir diferentes representaciones de un objeto sin modificar su código?
Se pueden agregar más builders para diferentes tipos de automóviles (híbridos, eléctricos, etc.).

Patrón Prototype

Prototype es un patrón creacional que permite clonar objetos existentes sin acoplar un código a sus clases específicas.

- El objeto se clona a partir de una instancia ya existente, lo que permite una mayor flexibilidad y facilidad de mantenimiento.
- No habría dependencia directa de los constructores ni de la implementación concreta de una clase.

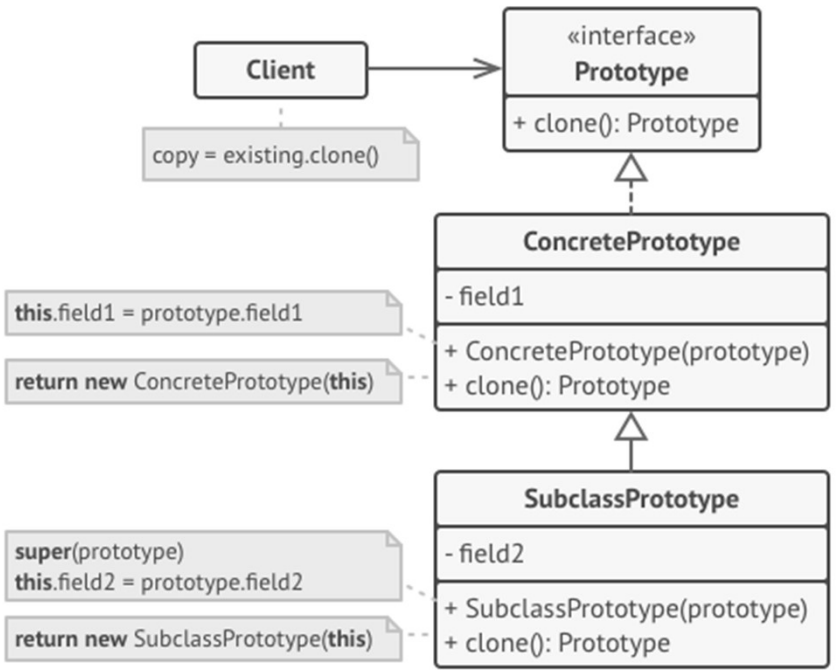
Porqué se quisiera clonar un Objeto?

Puede ocurrir que el costo computacional y de recursos para generar una nueva instancia es alto en comparación al costo de realizar una copia de una instancia ya existente. Algunos ejemplos:

- Instanciar un objeto cuyos datos han sido obtenidos previamente. Debería volver a usar los recursos para obtener los datos nuevamente y asignarlos al objeto recientemente instanciado. Sale mejor, clonarlo.
- Un sistema de encuestas permite generar formularios personalizados con base en plantillas predefinidas. En lugar de construir cada formulario desde cero, se usa un prototipo (FormularioBase) que se clona y se personaliza.
- Un sistema empresarial que maneja objetos costosos (en recursos) de inicializar (conexiones a bases de datos, configuraciones de usuario). Se mantiene un objeto en caché y, cuando se necesita, se clona en lugar de instanciarlo de nuevo.
- En sistemas de IA o Machine Learning, se requiere generar múltiples copias de un modelo de datos para experimentos o simulaciones. Se crea un modelo base (ModeloBase) y se clona varias veces con parámetros distintos, en lugar de reentrenarlo desde cero.

Estructura

Cliente, la clase que solicita al prototipo que se clone.



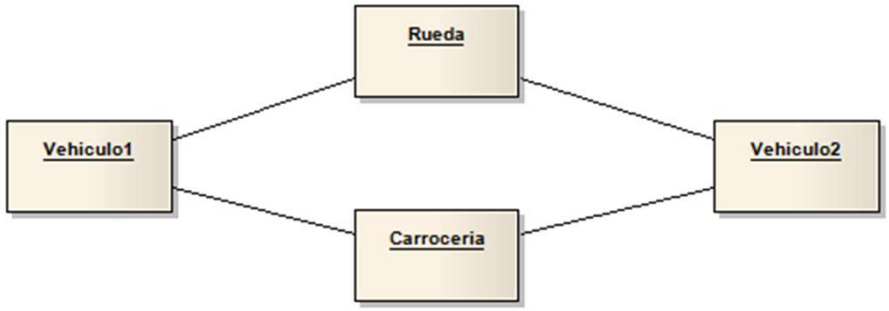
Interface que define la operación de clonado. Normalmente firma un solo método clone

Prototipo Concreto, implementa la interface y su método Clone() para proceder al clonado del objeto.

Tipos de Clonación

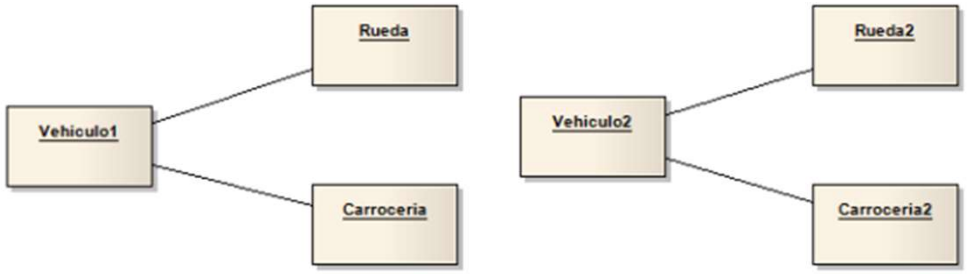
Contexto: Vehiculo es un todo compuesto por Carrocería y Rueda

Clonación superficial (shallow copy)



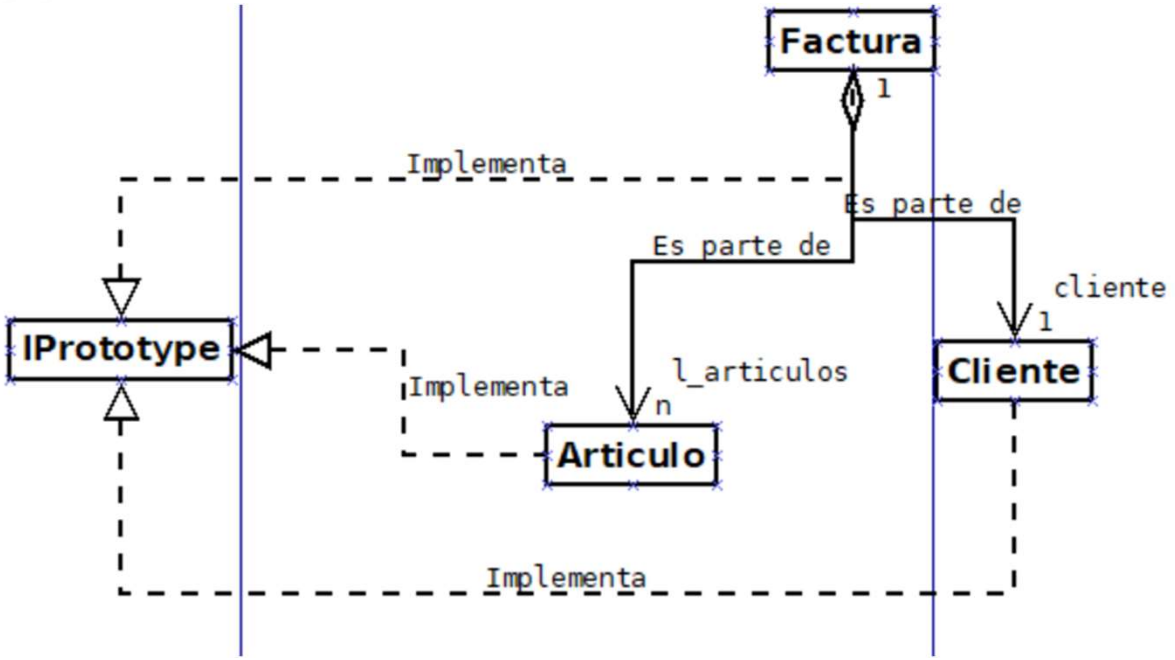
- Se clona bit a bit, por lo tanto lo único que se clona es el todo y los clones comparten las partes
- Los clones del todo son instancias independientes, si se cambia un valor de un atributo, no afecta a los otros
- Las partes son las mismas para los clones del todo, por lo tanto un cambio en los atributos de las partes, cambian para los clones relacionados

Clonación Profunda (deep copy)



- En esta clonación, se clona la estructura Todo Partes completa.

https://danielggarcia.wordpress.com/



Implementemos

Un Arquitecto de Software usa Prototype cuando:

- Se requiere la creación eficiente de objetos similares sin pasar por un proceso costoso de configuración.
- Se necesita evitar el uso de new repetidamente, lo cual mejora la modularidad y la capacidad de prueba del código.
- Se requiere flexibilidad para duplicar objetos sin conocer sus clases exactas.
- Se trabaja con configuraciones complejas donde es más sencillo copiar un prototipo en lugar de configurar uno nuevo desde cero.

Cuando un equipo de desarrollo considera usar Prototype, estas son algunas preguntas clave:

✓ ¿Los objetos son costosos de crear?
El uso de Prototype puede ahorrar recursos.

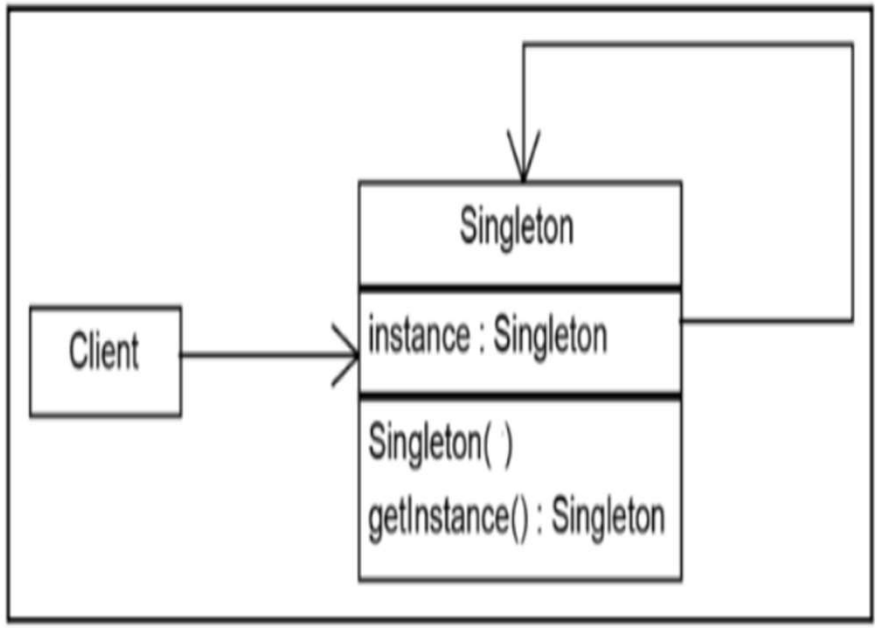
✓ ¿Necesitamos muchas variaciones de un mismo objeto con pequeñas diferencias?
Prototype permite clonar y modificar solo lo necesario.

✓ ¿Los objetos contienen estructuras complejas o referencias anidadas?
Se debe definir si la clonación debe ser superficial o profunda.

✓ ¿Es un problema usar new directamente en múltiples lugares del código?
Si se quiere evitar la creación repetitiva y mejorar el mantenimiento, Prototype es una buena solución.

✓ ¿Queremos reducir la dependencia de clases concretas y mejorar la flexibilidad del código?
Con Prototype, el código depende menos de los constructores y más de la clonación de instancias ya configuradas.

Patrón Singleton



Tomado de GeekforGeeks

Singleton es un patrón creacional que asegura que una clase tenga una única instancia y proporciona un punto global de acceso a esa instancia.

Es muy útil para situaciones que requieren un control centralizado, como ejemplo:

- La gestión de conexiones de bases de datos
- Una clase Configuración que todos la deben usar
- Una clase que recibe los datos de varios sensores y todos pueden acceder a esta a consultarlos

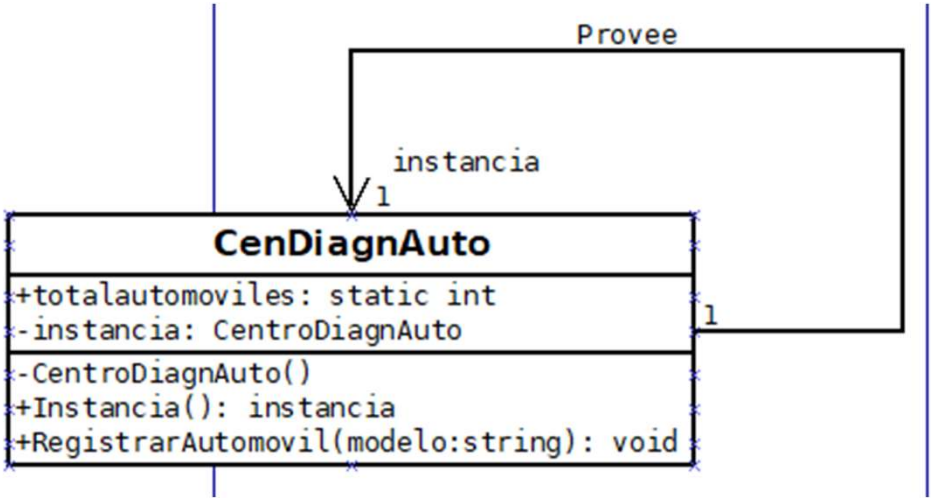
No se puede usar un constructor normal.

Evita que otro objeto sobreescriba la instancia

Atributo Estático

En C# se usa el delegado Lazy<T>, permite la creación de objetos de forma diferida

Accesor de tipo get



Ventajas de Singleton:

- El patrón Singleton garantiza que solo haya una instancia con un identificador único, lo que ayuda a prevenir problemas de nombres.
- Este patrón admite tanto la inicialización ansiosa (crear la instancia cuando se carga la clase) como la inicialización perezosa (crearla cuando se necesita por primera vez), lo que proporciona adaptabilidad en función del caso de uso.
- Cuando se implementa correctamente, un Singleton puede ser seguro para subprocesos, lo que garantiza que varios subprocesos no creen accidentalmente instancias duplicadas.
- Al mantener solo una instancia, el patrón Singleton puede ayudar a reducir el uso de memoria en aplicaciones donde los recursos son limitados.

Desventajas de Singleton:

- Los singletons pueden dificultar las pruebas unitarias, ya que introducen un estado global, su estado puede influir en los resultados de las pruebas.
- En entornos multiproceso, el proceso de creación e inicialización de un Singleton puede generar “alta concurrencia” si varios subprocesos intentan crearlo simultáneamente.
- Si más adelante descubre que necesita varias instancias o desea modificar la forma en que se crean las instancias, es posible que se requieran cambios importantes en el código.
- El patrón Singleton crea una dependencia global, lo que puede complicar el reemplazo del Singleton con una implementación diferente o el uso de inyección de dependencia.
- Crear subclases de un Singleton puede ser complicado, ya que el constructor suele ser privado. Esto requiere un manejo cuidadoso y puede no ajustarse a las prácticas de herencia estándar.

Singleton y Atributos estáticos

Cuando una clase Singleton tiene atributos estáticos en C#, estos atributos se comparten en toda la aplicación y son accesibles sin necesidad de instanciar el Singleton.

Comportamiento de Atributos Estáticos en un Singleton

1.Pertenecen a la clase, no a la instancia:

- ✓ Los atributos estáticos se almacenan en memoria **una sola vez** y no dependen de la instancia del Singleton.

2.Se inicializan en el primer acceso:

- ✓ En C#, los atributos estáticos se inicializan la **primera vez que se accede a la clase**, y **su valor se mantiene a lo largo de toda la ejecución del programa**.

3.Independencia de la instancia del Singleton:

- ✓ Aunque un Singleton **garantiza que solo hay una instancia**, los atributos estáticos existen **independientemente de la instancia**.

En el contexto de desarrollo de software automotriz, un **arquitecto de software** podría usar el Singleton en:

- **Un módulo de diagnóstico de vehículos**: Centraliza la recopilación de datos de sensores de todos los automóviles en producción.
- **Un gestor de configuración del sistema**: Evita que múltiples instancias configuren parámetros críticos de los sistemas electrónicos.
- **Un controlador de acceso a una base de datos de vehículos**: Asegura que todas las consultas pasen por un único punto de acceso.

Bibliografía

- Shvets Alexander. Dive into Design Patterns. Refactoring.guru. 2019.
- Perera Srinath. Software Architecture and Decision-Making_ Leveraging Leadership, Technology, and Product Management to Build Great Products. Addison-Wesley. 2024
- Pacheco, Diego_Sgro, Sam_ - Principles of Software Architecture Modernization _ Delivering engineering excellence with the art of fixing microservices, mono. BPB Publicatio. 2024
- Gabriel Baptista, Francesco Abbruzzese - Software Architecture with C_ 12 and .NET 8 - Fourth Edition_ Build enterprise applications using microservices, DevOps, EF Core. 2024
- Oliver Goldman.Effective Software Architecture_ Building Better Software Faster (2024, Addison-Wesley Professional)
- John Gilbert - Software Architecture Patterns for Serverless Systems_ Architecting for innovation with event-driven microservices (2024)

Cibergrafía

- Freecodecamp
- Openwebinars.net
- YouTube: Software Architecture Monday
- refactoring.guru
- Github
- Coursera "Software Architecture”. University of Alberta
- https://www.geeksforgeeks.org/