UNSA

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERÍA Y PRODUCCIÓN DE SERVICIOS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

“AÑO DEL BICENTENARIO DEL PERÚ: 200 AÑOS DE INDEPENDENCIA”

CIENCIA DE LA COMPUTACIÓN

GRUPO: A SEMESTRE: 2021 - A

Patrones de Diseño

INTEGRANTES:

Miguel Angel Deza Cuela

Katherine Nikole Béjar Román

Josue Gabriel Sumare Uscca

Yanira Angie Suni Quispe

DOCENTE: Alvaro Henry Mamani Aliaga

Arequipa – Perú

2021

Prototype

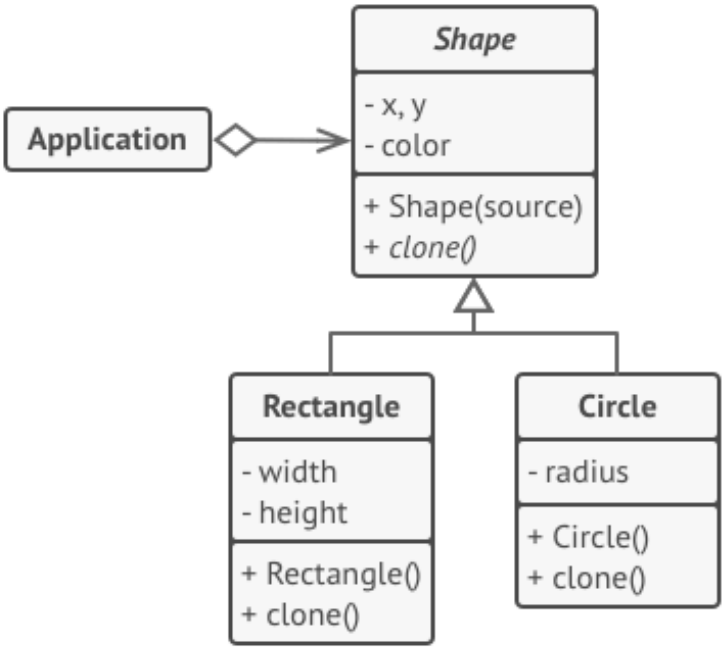
Prototype es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método `clonar`.

Un objeto que soporta la clonación se denomina *prototipo*. Cuando tus objetos tienen decenas de campos y miles de configuraciones posibles, la clonación puede servir como alternativa a la creación de subclases.

En este ejemplo, el patrón Prototype nos permite producir copias exactas de objetos geométricos sin acoplar el código a sus clases.

Puntos Clave:



- **Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.**

- El patrón Prototype proporciona al código cliente una interfaz general para trabajar con todos los objetos que soportan la clonación. Esta interfaz hace que el código cliente sea independiente de las clases concretas de los objetos que clona.

- **Utiliza el patrón 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.**

Relaciones con otros patrones

- Muchos diseños empiezan utilizando el Factory Method (menos complicado y más personalizable mediante las subclases) y evolucionan hacia Abstract Factory, Prototype, o Builder (más flexibles, pero más complicados)
  .
- Las clases del Abstract Factory a menudo se basan en un grupo de métodos de fábrica, pero también puedes utilizar Prototype para escribir los métodos de estas clases.

- Prototype puede ayudar cuando necesitas guardar copias de Comandos en un historial.

Ventajas de su Uso

- Puedes clonar objetos sin acoplarlos a sus clases concretas.

- Puedes evitar un código de inicialización repetido clonando prototipos prefabricados.

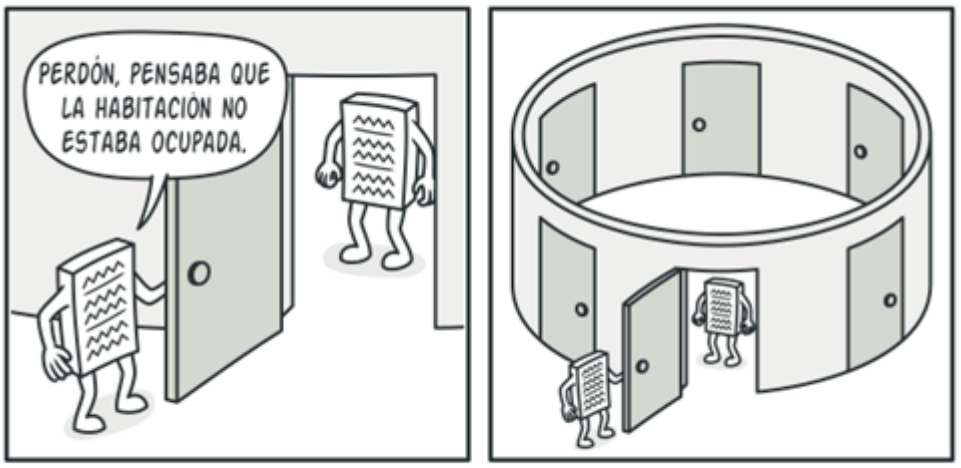
- Puedes crear objetos complejos con más facilidad.

Singleton

Singleton 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.

El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el Principio de responsabilidad única:

- Garantizar que una clase tenga una única instancia.
- Proporcionar un punto de acceso global a dicha instancia



Puede ser que los clientes ni siquiera se den cuenta de que trabajan con el mismo objeto todo el tiempo.

Pasos:

Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

1. Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador `new` con la clase Singleton.

2. Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

Aplicabilidad:

- Utiliza el patrón Singleton 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.

  El patrón Singleton deshabilita el resto de las maneras de crear objetos de una clase, excepto el método especial de creación. Este método crea un nuevo objeto, o bien devuelve uno existente si ya ha sido creado.

- Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.

  Al contrario que las variables globales, el patrón Singleton garantiza que haya una única instancia de una clase. A excepción de la propia clase Singleton, nada puede sustituir la instancia en caché.

  Ten en cuenta que siempre podrás ajustar esta limitación y permitir la creación de cierto número de instancias Singleton. La única parte del código que requiere cambios es el cuerpo del método `getInstance`.

Ventajas:

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

Desventajas:

1. Vulnera el Principio de responsabilidad única. El patrón resuelve dos problemas al mismo tiempo.
2. El patrón Singleton puede enmascarar un mal diseño, por ejemplo, cuando los componentes del programa saben demasiado los unos sobre los otros.
3. 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.

4. 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.

Relaciones con otros patrones de diseño:

- Una clase Facade a menudo puede transformarse en una Singleton, ya que un único objeto fachada es suficiente en la mayoría de los casos.

- Flyweight podría asemejarse a Singleton si de algún modo pudieras reducir todos los estados compartidos de los objetos a un único objeto flyweight. Pero existen dos diferencias fundamentales entre estos patrones:

    1. Solo debe haber una instancia Singleton, mientras que una clase Flyweight puede tener varias instancias con distintos estados intrínsecos.
    2. El objeto Singleton puede ser mutable. Los objetos flyweight son inmutables.

- Los patrones Abstract Factory, Builder y Prototype pueden todos ellos implementarse como Singletons.

Abstract Factory

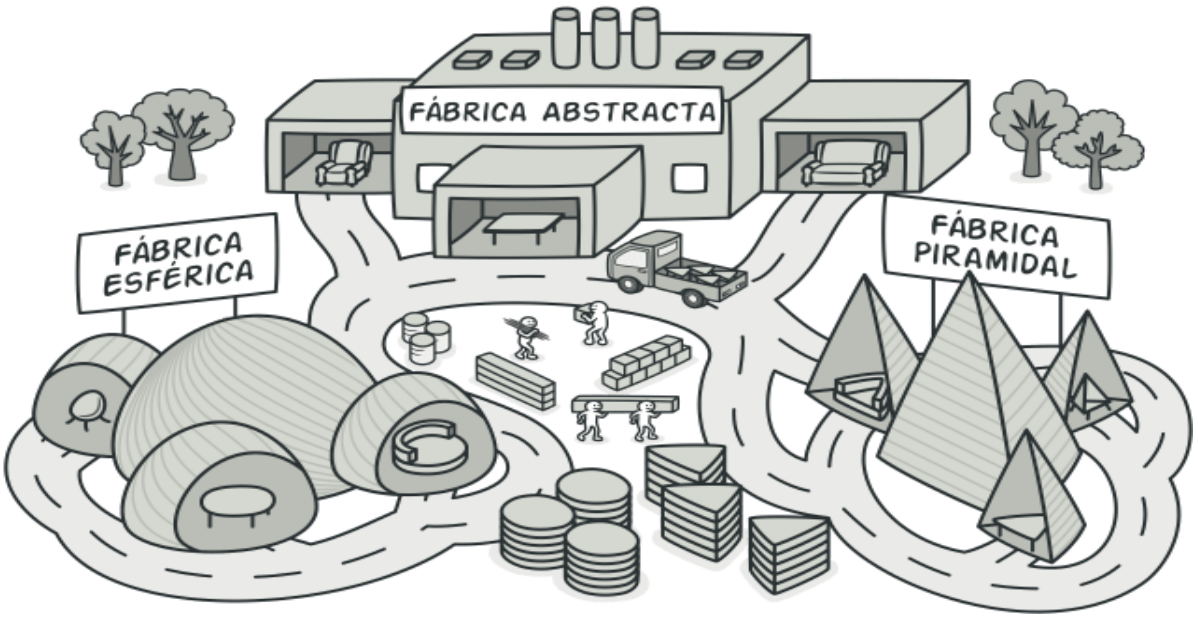
Este patrón añade un nivel más de complejidad. Anteriormente la clase Factory devolvía objetos de diferentes tipos, en este caso, devolverá diferentes clases Factory según el parámetro enviado.

Por ejemplo: En un sistema podemos tener una clase Abstract Factory que devuelva diferentes objetos Look&Feel específicos para una plataforma (Windows, Linux, Mac, ...). A su vez, estos objetos pueden ser clases Factory que devuelven los diferentes componentes correspondientes a cada una de esas plataformas.

El patrón Abstract Factory define una interfaz para crear todos los productos, pero deja la propia creación de productos para las clases de fábrica concretas. Cada tipo de fábrica se corresponde con cierta variedad de producto.

El código cliente invoca los métodos de creación de un objeto de fábrica en lugar de crear los productos directamente con una llamada al constructor (operador `new`). Como una fábrica se corresponde con una única variante de producto, todos sus productos serán compatibles.

El código cliente trabaja con fábricas y productos únicamente a través de sus interfaces abstractas. Esto permite al mismo código cliente trabajar con productos diferentes. Simplemente, creas una nueva clase fábrica concreta y la pasas al código cliente.



Aplicabilidad:

- Utiliza el patrón Abstract Factory 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.
- El patrón Abstract Factory nos ofrece una interfaz para crear objetos a partir de cada clase de la familia de productos. Mientras tu código cree objetos a través de esta interfaz, no tendrás que preocuparte por crear la variante errónea de un producto que no combine con los productos que ya ha creado tu aplicación.
- Considera la implementación del patrón Abstract Factory cuando tengas una clase con un grupo de métodos de fábrica que nublen su responsabilidad principal.
- En un programa bien diseñado cada clase es responsable tan solo de una cosa. Cuando una clase lidia con varios tipos de productos, puede ser que valga la pena extraer sus métodos de fábrica para ponerlos en una clase única de fábrica o una implementación completa del patrón Abstract Factory.

Ventajas:

- 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 abierto/cerrado*. Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente.

Desventaja:

- Puede ser que el código se complique más de lo que debería, ya que se introducen muchas nuevas interfaces y clases junto al patrón.

Relaciones con otros patrones:

- Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).
- **Builder** se enfoca en construir objetos complejos, paso a paso. **Abstract Factory** se especializa en crear familias de objetos relacionados. *Abstract Factory* devuelve el producto inmediatamente, mientras que *Builder* te

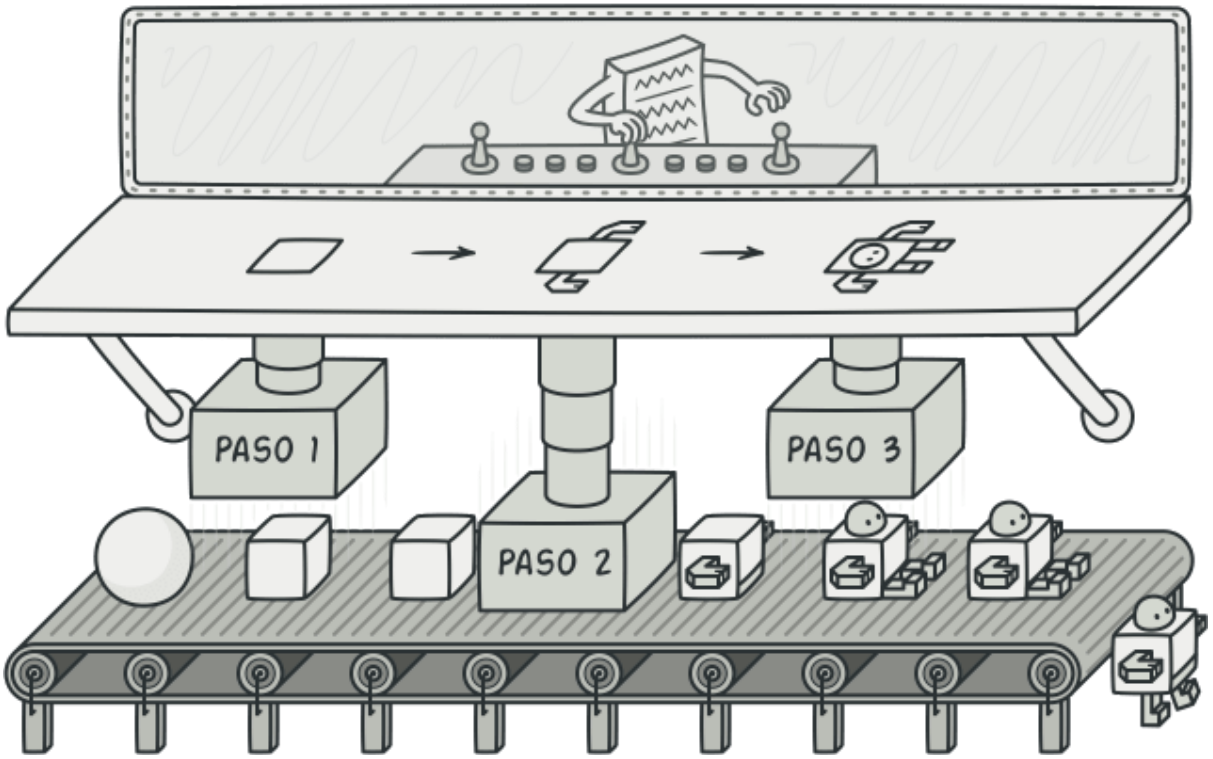
permite ejecutar algunos pasos adicionales de construcción antes de extraer el producto.

- Las clases del **Abstract Factory** a menudo se basan en un grupo de **métodos de fábrica**, pero también puedes utilizar **Prototype** para escribir los métodos de estas clases.
- **Abstract Factory** puede servir como alternativa a **Facade** cuando tan solo deseas esconder la forma en que se crean los objetos del subsistema a partir del código cliente.
- Puedes utilizar **Abstract Factory** junto a **Bridge**. Este emparejamiento resulta útil cuando algunas abstracciones definidas por *Bridge* sólo pueden funcionar con implementaciones específicas. En este caso, *Abstract Factory* puede encapsular estas relaciones y esconder la complejidad al código cliente.
- Los patrones **Abstract Factory**, **Builder** y **Prototype** pueden todos ellos implementarse como **Singletons**.

Builder

Builder 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.

El patrón Builder es muy conocido en el mundo C++. Resulta especialmente útil cuando debes crear un objeto con muchas opciones posibles de configuración.



Aplicabilidad:

- Imagina un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados. Normalmente, este código de inicialización está sepultado dentro de un monstruoso constructor con una gran cantidad de parámetros. O, peor aún: disperso por todo el código cliente.

Ventajas:

- 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.

Desventaja:

- La complejidad general del código aumenta, ya que el patrón exige la creación de varias clases nuevas.

Relaciones con otros patrones:

- Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).
- **Builder** se enfoca en construir objetos complejos, paso a paso. **Abstract Factory** se especializa en crear familias de objetos relacionados. *Abstract Factory* devuelve el producto inmediatamente, mientras que *Builder* te permite ejecutar algunos pasos adicionales de construcción antes de extraer el producto.
- Puedes utilizar **Builder** al crear árboles **Composite** complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.
- Puedes combinar **Builder** con **Bridge**: la clase *directora* juega el papel de la abstracción, mientras que diferentes *constructoras* actúan como *implementaciones*.
- Los patrones **Abstract Factory**, **Builder** y **Prototype** pueden todos ellos implementarse como **Singletons**.

Factory Method

El patrón Factory, o patrón de diseño Método Factoría, describe un enfoque de programación que sirve para crear objetos sin tener que especificar su clase exacta. Esto quiere decir que el objeto creado puede intercambiarse con flexibilidad y facilidad. Para implementar este método, los desarrolladores utilizan el Factory Method, que da nombre a este patrón. Su uso puede especificarse en una interfaz o implementarse mediante la clase hijo o la clase base y opcionalmente sobrescribirse (mediante clases derivadas). Al hacerlo, el patrón o método toma el lugar del constructor de clase normal para separar la creación de objetos de los propios objetos. Como resultado, es posible seguir los principios SOLID.

Aplicabilidad:

- Usa el patrón Factory method cuando:
- Una clase no puede prever la clase de objetos que tiene que crear
- Una clase quiere que sus subclases decidan qué objetos crean
- Las clases delegan responsabilidades a una de entre varias subclases auxiliares, y queremos localizar en qué subclase concreta se ha delegado

Ventajas:

- Se gana en flexibilidad, pues crear los objetos dentro de una clase con un "Método de Fábrica" es siempre más flexible que hacerlo directamente,

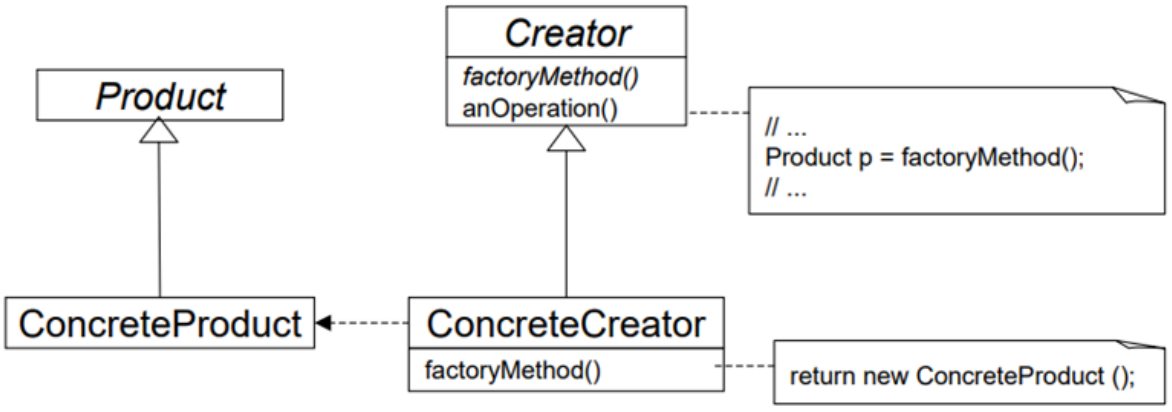
debido a que se elimina la necesidad de atar nuestra aplicación a unas clases de productos concretos.

- Se facilitan futuras ampliaciones, puesto que se ofrece a las subclases la posibilidad de proporcionar una versión extendida de un objeto, con sólo aplicar en los Productos la misma idea del "Método de Fábrica".
- Se facilita, en cuanto a que se hace natural, la conexión entre jerarquías de clases paralelas, que son aquellas que se generan cuando una clase delega algunas de sus responsabilidades en una clase aparte. Ambas jerarquías de clases paralelas son creadas por un mismo cliente y el patrón Método de Fábrica establece la relación entre parejas de subclases concretas en las mismas.

Desventaja:

- Se obliga al cliente a definir subclases de la clase "Creador" sólo para crear un producto concreto y esto puede no ser apropiado siempre.

Estructura:



Product: Define la interfaz de los objetos que crea el método factoría.
ConcreteProduct: Implementa la interfaz de Product.
Creator:

- ➢ Declara el método factoría que devuelve un objeto de tipo Product. Puede definir una implementación por defecto de dicho método, que devuelva un objeto de algún producto concreto ConcreteProduct.
- ➢ Puede llamar al método factoría para crear un objeto de tipo Product.

ConcreteCreator: Sobrescribe el método factoría para devolver un objeto de algún ConcreteProduct.

Relaciones con otros patrones de diseño:

- Abstract Factory es frecuentemente implementado con el Método de Fábrica.
- Factory Method habitualmente se utiliza dentro del Método Plantilla.
- Prototype no requiere hacer subclases de "Creador", pero sí una operación de "Inicializar" en la clase "Producto", que no le hace falta a Factory Method.