IIC 2143 - Ingeniería de Software

Patrones de diseño III

M. Trinidad Vargas
mtvargas1@uc.cl

Clase “auspiciada” por Refactoring Guru

https://refactoring.guru/es/design-patterns/

Clasificación de patrones

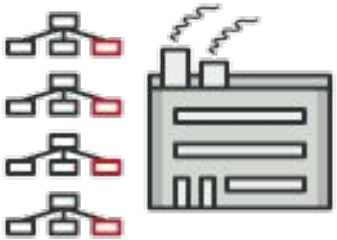
Según su propósito

- **Comportamiento:** brindan flexibilidad para agregar comportamiento.
- **Creacionales:** incrementan la flexibilidad y reusabilidad al momento de crear objetos complejos..
- **Estructurales:** mantiene la flexibilidad y reusabilidad al momento de componer estructuras de objetos.

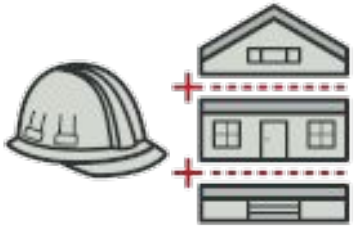
Patrones de diseño

Creacionales

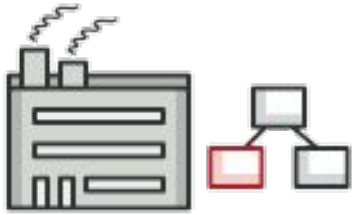
Abstract Factory



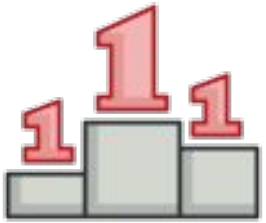
Builder



Factory

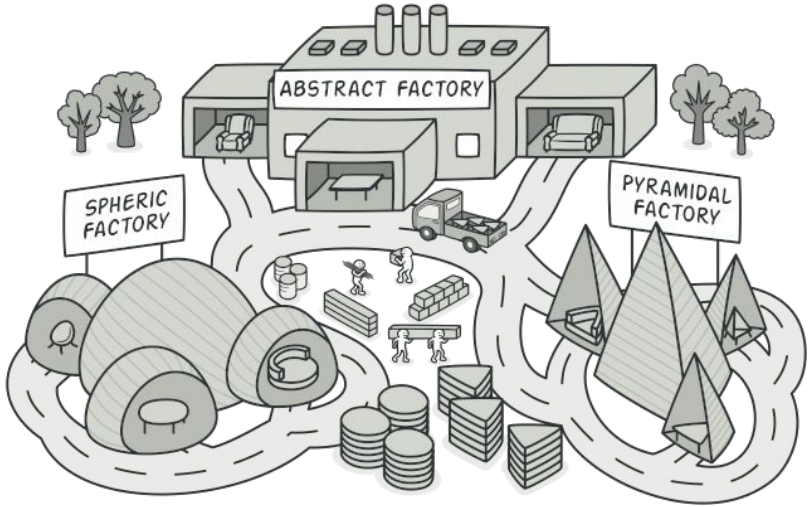


Singleton



Abstract Factory

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



https://refactoring.guru/design-patterns/abstract-factory

Abstract Factory

Caso

Tienes un simulador de tienda de muebles donde ofreces distintos productos relacionados como silla, sofá y mesa en tres estilos distintos: moderno, victoriano y art-deco.

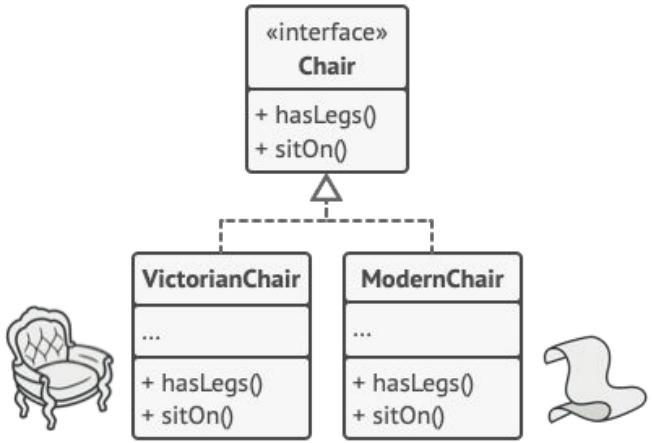
Problema

Necesitamos crear objetos individuales que combinen entre ellos y queremos agregar nuevos productos sin cambiar el código existente.

Abstract Factory

Solución

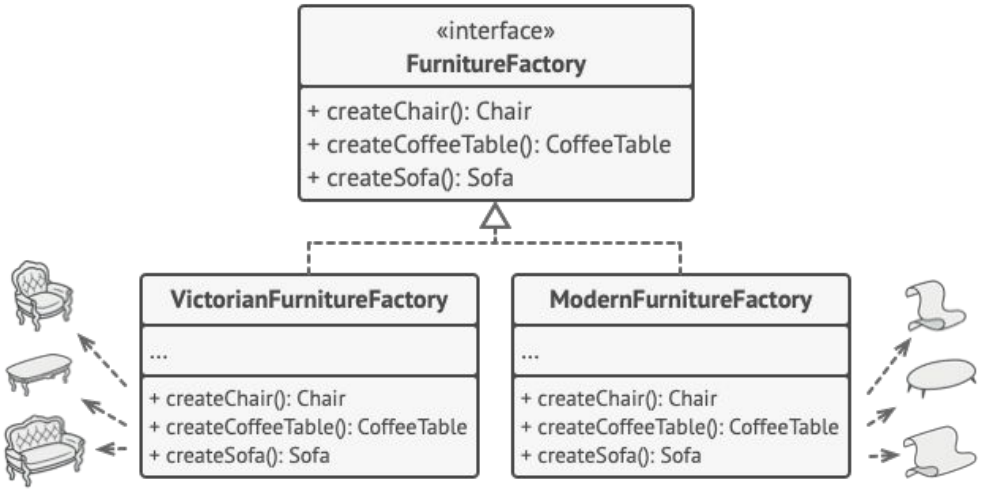
- Declarar una interfaz para cada producto diferente de la familia de productos. Todas las variantes de producto seguirán esta interfaz.



Abstract Factory

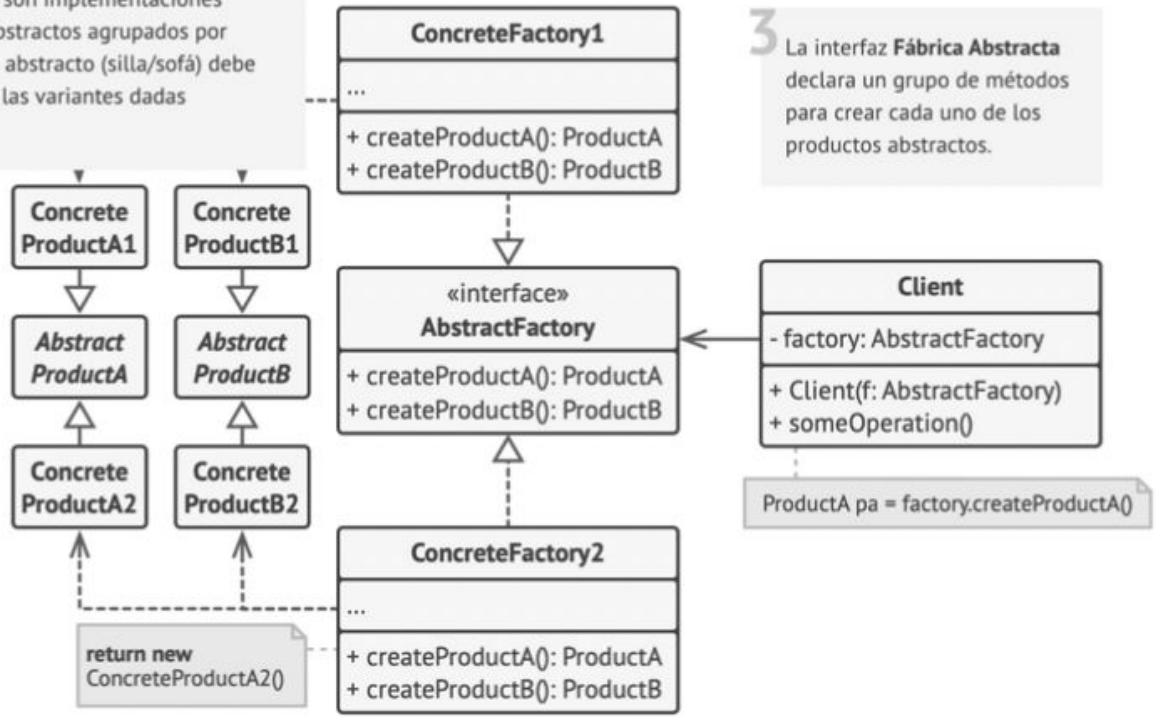
Solución

- Luego declaramos la fábrica abstracta: Una interfaz con una lista de métodos de creación para todos los productos parte de la familia de productos.
    - Los métodos devuelven productos abstractos.



2 Los **Productos Concretos** son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (silla/sofá) debe implementarse en todas las variantes dadas (victoriano/moderno).

1 Los **Productos Abstractos** declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.



3 La interfaz **Fábrica Abstracta** declara un grupo de métodos para crear cada uno de los productos abstractos.

4 Las **Fábricas Concretas** implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan solo dichas variantes de los productos.

5 Aunque las fábricas concretas instancian productos concretos, las firmas de sus métodos de creación deben devolver los productos *abstractos* correspondientes. De este modo, el código cliente que utiliza una fábrica no se acopla a la variante específica del producto que obtiene de una fábrica. El **Cliente** puede funcionar con cualquier variante fábrica/producto concreta, siempre y cuando se comunique con sus objetos a través de interfaces abstractas.

Ejemplo

Queremos construir una interfaz gráfica que funcione en distintas plataformas (Windows y Web), permitiendo crear botones adaptados a cada estilo sin modificar el código cliente.

Usaremos el patrón Abstract Factory para encapsular la creación de estos botones y mantener la independencia entre la lógica de negocio y la implementación específica de cada plataforma.

Ejemplo

```ruby
# 1. Interfaz de Button
class Button
 def render
   raise NotImplementedError, 'Subclases deben implementar el método render'
 end

 def on_click
   raise NotImplementedError, 'Subclases deben implementar el método on_click'
 end
end
```

```ruby
# 2. Productos concretos (WindowsButton y HTMLButton)
class WindowsButton < Button
 def render
   puts 'Renderizando botón estilo Windows'
 end
 def on_click
   puts 'Click en botón Windows'
 end
end
class HTMLButton < Button
 def render
   puts 'Renderizando botón HTML para web'
 end
 def on_click
   puts 'Click en botón HTML'
 end
end
```

Ejemplo

```ruby
# 3. Dialog: Cliente abstracto
class Dialog
 def render
   button = create_button
   button.on_click
   button.render
 end

 def create_button
   raise NotImplementedError, 'Subclases
deben implementar create_button'
 end
end
```

```ruby
# 4. Fábricas concretas
class WindowsDialog < Dialog
 def create_button
   WindowsButton.new
 end
end

class WebDialog < Dialog
 def create_button
   HTMLButton.new
 end
end
```

Ejemplo

```
# 5. Código del cliente
def client_code(dialog)
 dialog.render
end

# Simulando elección de plataforma
platform = 'web'  # o 'windows'

dialog = case platform
        when 'windows' then
WindowsDialog.new
        when 'web'     then
WebDialog.new
        else raise 'Plataforma no
soportada'
        end

client_code(dialog)
```

Ejemplo usando Abstract Factory

En lugar de que el cliente elija y cree la fábrica (WebDialog.new o WindowsDialog.new), vamos a inyectarla desde fuera, por ejemplo, al construir el objeto principal.

Resultado

- El objeto Dialog no sabe nada de WindowsButton, HTMLButton, ni de clases específicas.
- Puedes sustituir la fábrica fácilmente (por ejemplo, una falsa para tests).
- Es más fácil de extender a nuevas plataformas sin tocar el código cliente.

Ejemplo usando Abstract Factory

```ruby
# 1. interfaz de fábrica
class ButtonFactory
 def create_button
   raise NotImplementedError
 end
end

# Fabricas concretas
class WindowsButtonFactory <
ButtonFactory
 def create_button
   WindowsButton.new
 end
end

class HTMLButtonFactory < ButtonFactory
 def create_button
   HTMLButton.new
 end
end
```

```ruby
# Dialog que depende de la fabrica
class Dialog
 def initialize(factory)
   @factory = factory  # Inyección de
dependencia
 end

 def render
   button = @factory.create_button
   button.on_click
   button.render
 end
end

def client_code(dialog)
 dialog.render
end
```

Ejemplo usando Abstract Factory

```
# Elección externa (cliente o
contenedor)
platform = 'windows'  # o 'web'

factory = case platform
          when 'windows' then
WindowsButtonFactory.new
          when 'web'     then
HTMLButtonFactory.new
          else raise 'Plataforma no
soportada'
          end

dialog = Dialog.new(factory)
client_code(dialog)
```

Abstract Factory

¿Cuándo aplicar 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.

Abstract Factory

Pros

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.

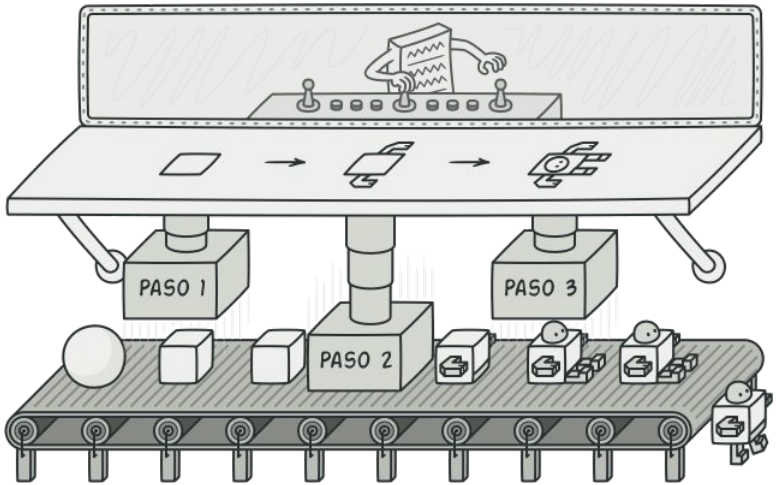
Principio de abierto/cerrado. Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente.

Contras

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

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.



https://refactoring.guru/design-patterns/builder

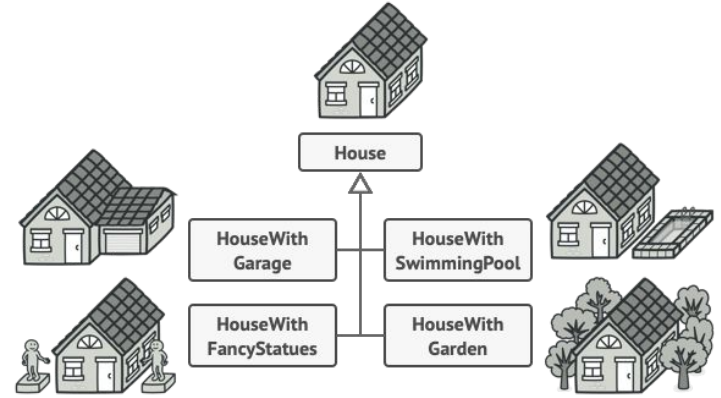
Builder

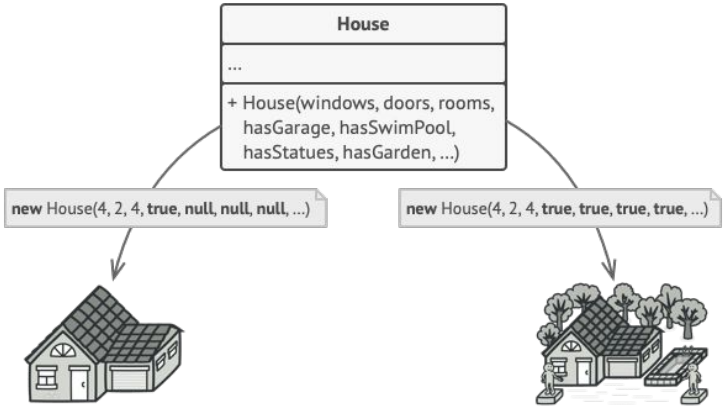
Caso

Tenemos un objeto complejo que requiere una inicialización con muchos parámetros y objetos anidados

Problema

Queremos diferentes tipos de casas y la primera opción es generar distintas subclases o parámetros lo que se vuelve inmantenible.

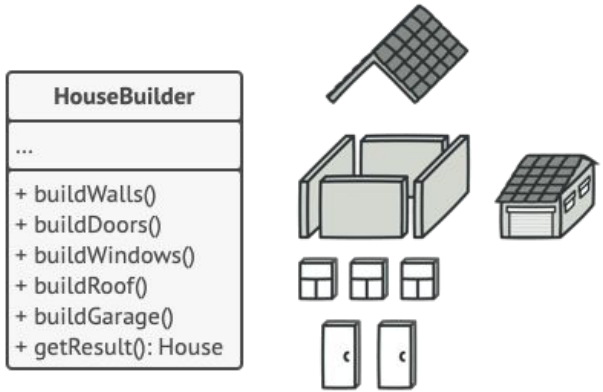


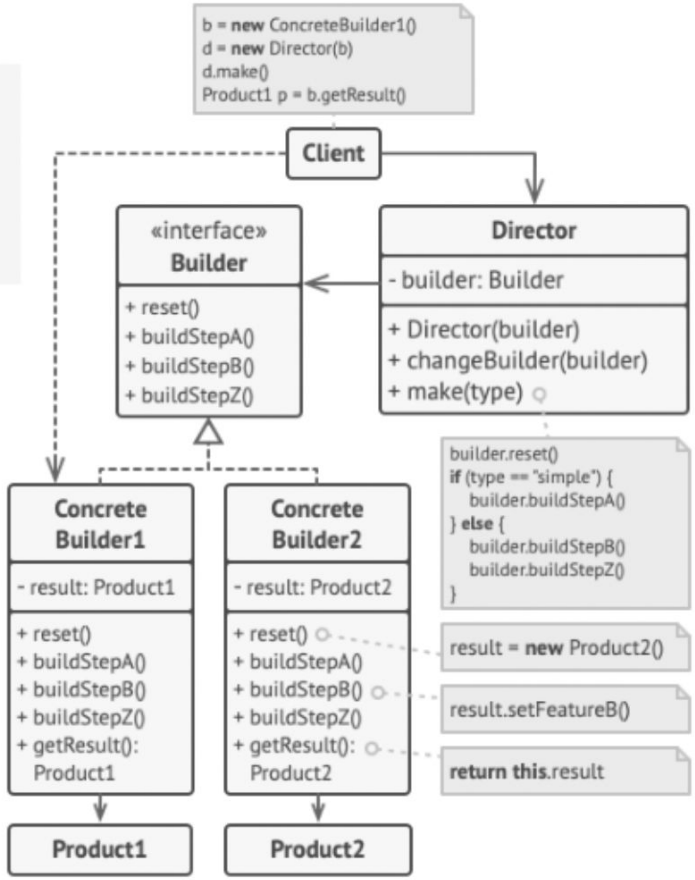


Builder

Solución

- Sacar el código de construcción del objeto de su propia clase y colocarlo dentro de objetos independientes, llamados constructores.





1 La interfaz **Constructora** declara pasos de construcción de producto que todos los tipos de objetos constructores tienen en común.

2 Los **Constructores Concretos** ofrecen distintas implementaciones de los pasos de construcción. Los constructores concretos pueden crear productos que no siguen la interfaz común.

3 Los **Productos** son los objetos resultantes. Los productos construidos por distintos objetos constructores no tienen que pertenecer a la misma jerarquía de clases o interfaz.

4 La clase **Directora** define el orden en el que se invocarán los pasos de construcción, por lo que puedes crear y reutilizar configuraciones específicas de los productos.

5 El **Cliente** debe asociar uno de los objetos constructores con la clase directora. Normalmente, se hace una sola vez mediante los parámetros del constructor de la clase directora, que utiliza el objeto constructor para el resto de la construcción. No obstante, existe una solución alternativa para cuando el cliente pasa el objeto constructor al método de producción de la clase directora. En este caso, puedes utilizar un constructor diferente cada vez que produzcas algo con la clase directora.

Builder

¿Cuándo aplicar Builder?

- Utiliza el patrón Builder 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 u otros objetos complejos.

Builder

Pros

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.

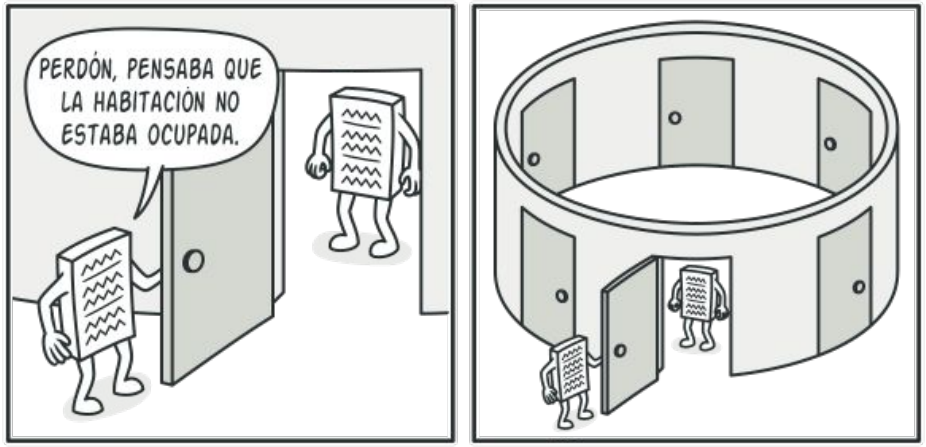
Principio de responsabilidad única. Puedes aislar un código de construcción complejo de la lógica de negocio del producto.

Contras

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

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.



https://refactoring.guru/design-patterns/singleton

Singleton

Caso

Imagina que has creado un objeto que es un recurso compartido, por ejemplo, una base de datos o un archivo.

Problema

Teniendo en cuenta este objeto, deseas

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

Singleton

Solución

Para garantizar que una clase tenga una única instancia:

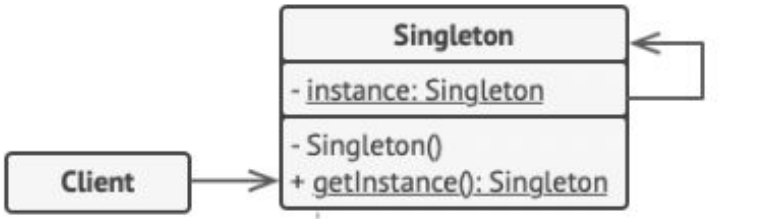
- Hacemos privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.
- Creamos un método de creación estático que actúa como constructor.

Para proporcionar un punto de acceso global a la instancia:

- La primera vez que llamemos al método de creación, se usará el método “new” y guardará el objeto en un atributo. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Singleton

Solución



1. La clase **Singleton** declara el método estático `obtenerInstancia` que devuelve la misma instancia de su propia clase.

   El constructor del Singleton debe ocultarse del código cliente. La llamada al método `obtenerInstancia` debe ser la única manera de obtener el objeto de Singleton.

```
if (instance == null) {
    // Nota: si estás creando una aplicación
    // que soporte el multihilo, debes
    // colocar un bloqueo de hilo aquí.
    instance = new Singleton()
}
return instance
```

Ejemplo

```ruby
class Singleton
 # Establece el método new como privado
 private_class_method :new
 # Creamos un método para acceder a la única instancia
 # de la clase
 def self.instance
   # Controlamos que solo se cree una instancia
   if @instance == nil
     @instance = new
   end
   return @instance
 end
end
```

```ruby
# la única forma de acceder a la instancia única es
# mediante el método instance
s1 = Singleton.instance
s2 = Singleton.instance
if s1.equal?(s2)
puts 'Singleton funciona, s1 y s2 contienen la misma
instancia.'
else
puts 'Singleton falla, s1 y s2 contienen distintas
instancias.'
end
```

Singleton

¿Cuándo aplicar 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.
- Cuando necesites un control más estricto de las variables globales.

Singleton

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.

Contras

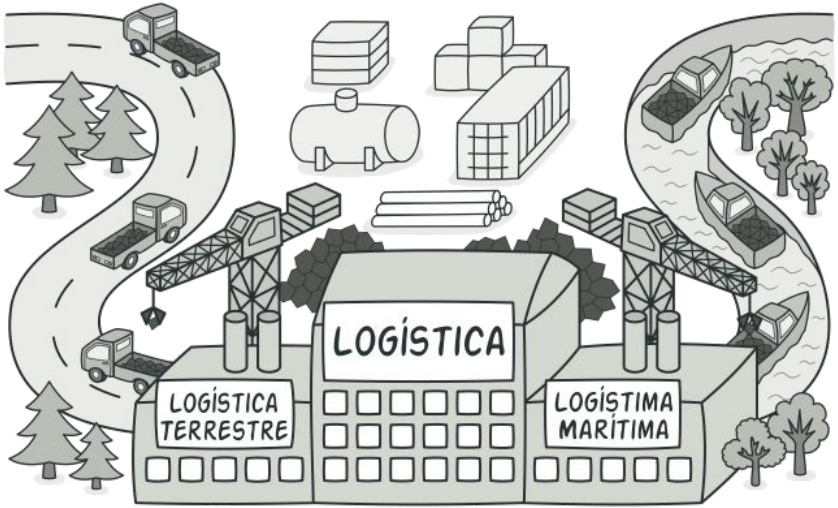
Vulnera el Principio de responsabilidad única.

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.

Factory Method

Factory Method 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.



https://refactoring.guru/es/design-patterns/factory-method

Factory Method

Caso Tienes una aplicación de gestión logística que incluye transporte en camión por lo que la mayor parte del código se encuentra dentro de la clase Camión. Dado el buen trabajo y éxito de tu app te piden crear la logística marítima.

Problema Actualmente tu código está completamente acoplado a la clase Camión y para agregar barcos a la app tienes que modificar toda la base de código.

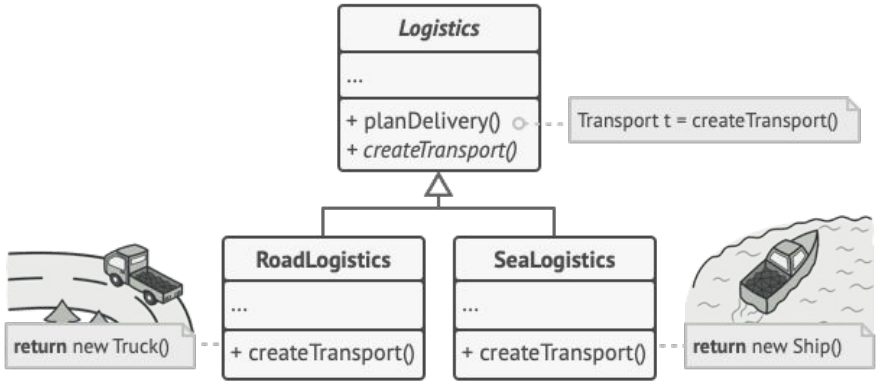
Factory Method

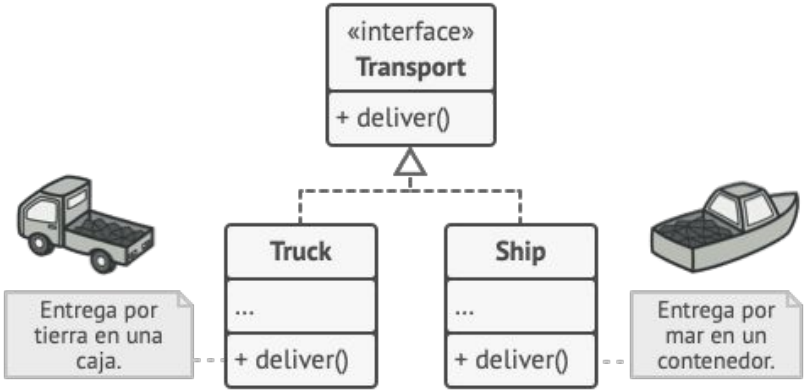
Solución

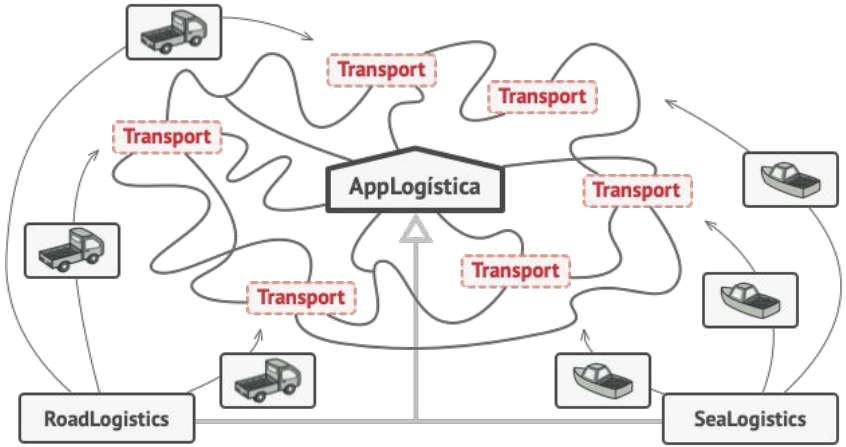
- En vez de utilizar el operador new para construir objetos, utilizar un método “fabrica”.
- Los objetos devueltos por un método de fábrica suelen denominarse productos.
- Todos los productos deben seguir la misma interfaz, para que el cliente pueda utilizarlos de manera indiferente.

Factory Method

Solución



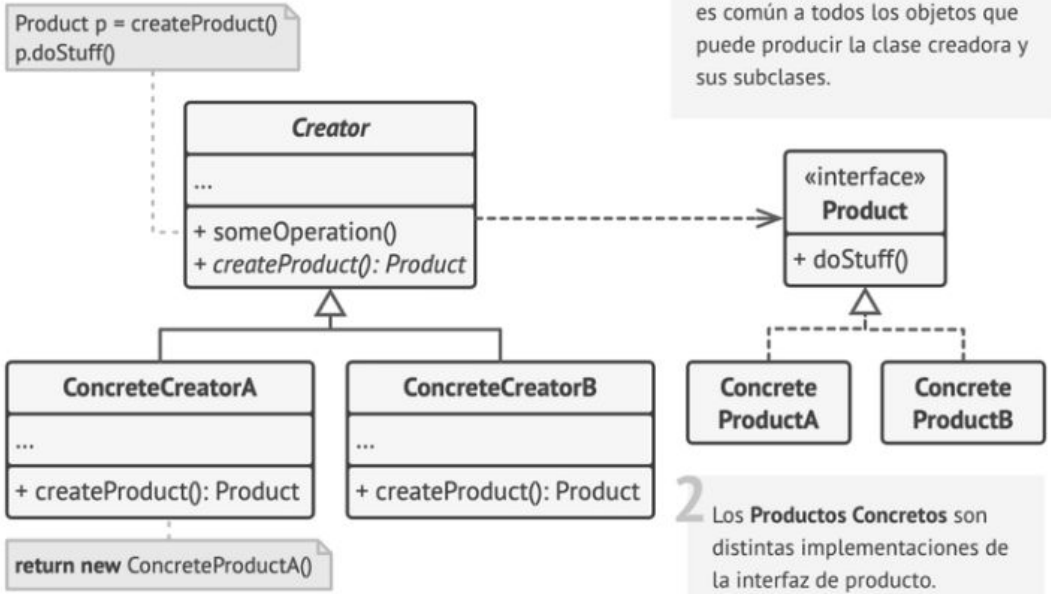




3 La clase **Creadora** declara el método fábrica que devuelve nuevos objetos de producto. Es importante que el tipo de retorno de este método coincida con la interfaz de producto.

Puedes declarar el patrón Factory Method como abstracto para forzar a todas las subclases a implementar sus propias versiones del método. Como alternativa, el método fábrica base puede devolver algún tipo de producto por defecto.

Observa que, a pesar de su nombre, la creación de producto **no** es la principal responsabilidad de la clase creadora. Normalmente, esta clase cuenta con alguna lógica de negocios central relacionada con los productos. El patrón Factory Method ayuda a desacoplar esta lógica de las clases concretas de producto. Aquí tienes una analogía: una gran empresa de desarrollo de software puede contar con un departamento de formación de programadores. Sin embargo, la principal función de la empresa sigue siendo escribir código, no preparar programadores.



1 El **Producto** declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.

2 Los **Productos Concretos** son distintas implementaciones de la interfaz de producto.

4 Los **Creadores Concretos** sobrescriben el Factory Method base, de modo que devuelva un tipo diferente de producto.

Observa que el método fábrica no tiene que **crear** nuevas instancias todo el tiempo. También puede devolver objetos existentes de una memoria caché, una agrupación de objetos, u otra fuente.

Ejemplo

¿Qué es necesario modificar para que el código siga el patrón Factory Method?

```ruby
class Entity
 def update_logic
   raise NotImplementedError
 end
end
class Boo < Entity
 def update_logic
   puts "Boo update logic..."
 end
end
class Goomba < Entity
 def update_logic
   puts "Goomba is ready to update..."
end end
class Koopa < Entity
 def update_logic
   puts "Koopa wants to update..."
 end
end
```

```ruby
class Game
 def initialize
   @enemies = []
 end
 def add_enemies(n, type)
   for i in (1..n)
     if type == "random"
       enemy = [Goomba, Boo, Koopa].sample.new
     elsif type == "boos"
       enemy = Boo.new
     elsif type == "goombas"
       enemy = Goomba.new
     end
     @enemies.push(enemy)
     enemy.update_logic
   end
 end
 def enemies
   @enemies
 end
end
```

```ruby
require_relative 'entities'
require_relative 'game'
game = Game.new
game.add_enemies(5,"random")
```

Factory Method

¿Cuándo aplicar Factory Method?

- Cuando no conoces de antemano los tipos exactos y las dependencias de los objetos con los que debería trabajar tu código.
- Cuando desees proporcionar a los usuarios de una biblioteca una forma de ampliar los componentes internos.

Factory Method

Pros

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.

Contras

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.

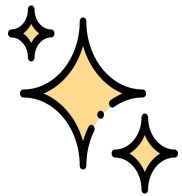
Ejercicio con décima

Extendiendo el ejemplo de abstract factory

Tu programa no solo necesita botones en múltiples plataformas, ahora necesitas soportar interfaces complejas con múltiples componentes como headers y checkbox.

¿Cuál es el patrón adecuado para extender este comportamiento?

Crea el diagrama de clases y el código en Ruby.



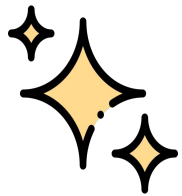
* con puts como en el ejemplo es suficiente

Ejercicio con décima

Extendiendo el ejemplo de abstract factory

Entregable

- Un archivo en pdf con el diagrama de clase de tu programa justificando por qué eligieron el patrón
- Un archivo .rb con el nuevo código
- Puedes entregar solo o con un compañero
- Disponible hasta el lunes a las 22 hrs



Ejercicios adicionales

1. Crea una tabla comparativa de todos los patrones visto con su categoría, propósito, cómo se relacionan las clases, los participantes claves y cuándo utilizar el patrón.
2. ¿Cuáles patrones puedes identificar en tu aplicación en Ruby on Rails? Identifica patrones que utiliza Rails y patrones que tu implementaste.
3. Crea los diagramas de clase para los distintos ejemplos en Ruby

Material Adicional

Existen más patrones creacionales

https://refactoring.guru/design-patterns/creational-patterns