SUBDIRECCIÓN ACADÉMICA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN
AGOSTO - DICIEMBRE 2019

INGENIERÍA EN SISTEMAS COMPUTACIONALES

MATERIA:
PATRONES DE DISEÑO

SERIE:
ADF-1701 SC8B

CATEDRÁTICO:
MARTHA ELENA PULIDO

TÍTULO:
EXPOSICIÓN: FÁBRICA ABSTRACTA

INTEGRANTES:
AGUIAR SOLIS PABLO 16211958
NAVARRO GONZALEZ ESMERALDA 16212052

FECHA DE ENTREGA:
10 DE FEBRERO DEL 2020

Introducción **2**

Desarrollo de tema **3**

Definiciones 3

Ventajas y desventajas 4

Características 4

Metodología 5

Estructura 5

Componentes 5

Aplicaciones 6

Ejemplos 7

Practicas 9

Conclusiones **10**

Referencias **10**

Introducción

Los patrones de diseño tienen su origen en la Arquitectura, cuando en 1979 el Arquitecto Christopher Alexander publicó el libro Timeless Way of Building, en el cual hablaba de una serie de patrones para la construcción de edificios, comparando la arquitectura moderna con la antigua y cómo la gente había perdido la conexión con lo que se considera calidad. Él utilizaba las siguientes palabras: "Cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez."

Este documento se centra en el patrón de diseño fábrica abstracta, el cual pertenece a una categoría definida como patrones creacionales, y como su nombre lo indica, se encuentran relacionados con la creación o construcción de objetos. Estos patrones intentan controlar la forma en que los objetos son creados implementando mecanismos que eviten la creación directa de objetos.

En el transcurso del documento, se detalla el patrón de diseño fábrica abstracta, mencionando aspectos relevantes como sus características, ventajas, desventajas, aplicaciones y finalmente, un ejemplo con la finalidad de una mejor comprensión.

Desarrollo de tema

Definiciones

En primer lugar, previo a definir el patrón de diseño fábrica abstracta, es necesario mencionar la categoría a la que pertenece, siendo un tipo de patrón creacional. Los patrones creacionales sirven para controlar la forma en que se crean los objetos, de entrada puede parecer un poco extraño, ya que se tiene la costumbre de crear libremente los objetos, sin embargo, existen situaciones donde por conveniencia es necesario establecer un mecanismo que permita crear instancias de una forma controlada. Esta necesidad puede nacer debido a que se requiera que sólo exista una instancia de una clase o no se sabe exactamente qué objeto se debe instanciar sino hasta en tiempo de ejecución o porque se busca que las clases sean creadas de una forma más simple mediante una clase de utilidad. Pueden existir cientos de motivos por los cuales se necesita que las clases sean creadas de forma controlada, sin embargo, lo importante es tener la visión de identificarlas y utilizar patrones creacionales que se adapten al problema.

Abstract Factory es un patrón de diseño creacional para el desarrollo de software. Provee una interfaz para crear familias de objetos relacionados o dependientes entre ellos sin especificar una clase en concreto. Los patrones de Abstract Factory funcionan en torno a una súper fábrica que crea otras fábricas. La implementación del patrón abstracto de fábrica nos proporciona un marco que nos permite crear objetos que siguen un patrón general. Entonces, en tiempo de ejecución, la fábrica abstracta se combina con cualquier fábrica de concreto deseada que pueda crear objetos del tipo deseado.

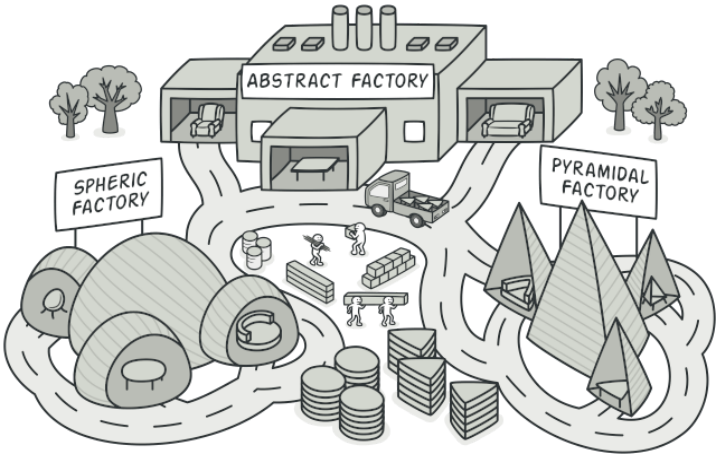


Imagen 1. Ilustración de abstract factory.

Ventajas y desventajas

Ventajas	Desventajas
Principio de responsabilidad única. Puede extraer el código de creación del producto en un solo lugar, haciendo que el código sea más fácil de soportar.	El código puede volverse más complicado de lo que debería ser, ya que se introducen muchas nuevas interfaces y clases junto con el patrón.
Evita el acoplamiento estrecho entre productos concretos y el código del cliente.	Difícil de agregar nuevos tipos de productos: No es fácil ampliar las fábricas abstractas para producir nuevos tipos de productos. Esto se debe a que la interfaz Abstract Factory corrige el conjunto de productos que se pueden crear. La compatibilidad con nuevos tipos de productos requiere ampliar la interfaz de la fábrica, lo que implica cambiar la clase Abstract Factory y todas sus subclases.
Los productos que obtienen de una fábrica son compatibles entre sí.	
Principio abierto / cerrado. Puede introducir nuevas variantes de productos sin romper el código de cliente existente.	

Características

El patrón Abstract Factory tiene los siguientes beneficios y responsabilidades:

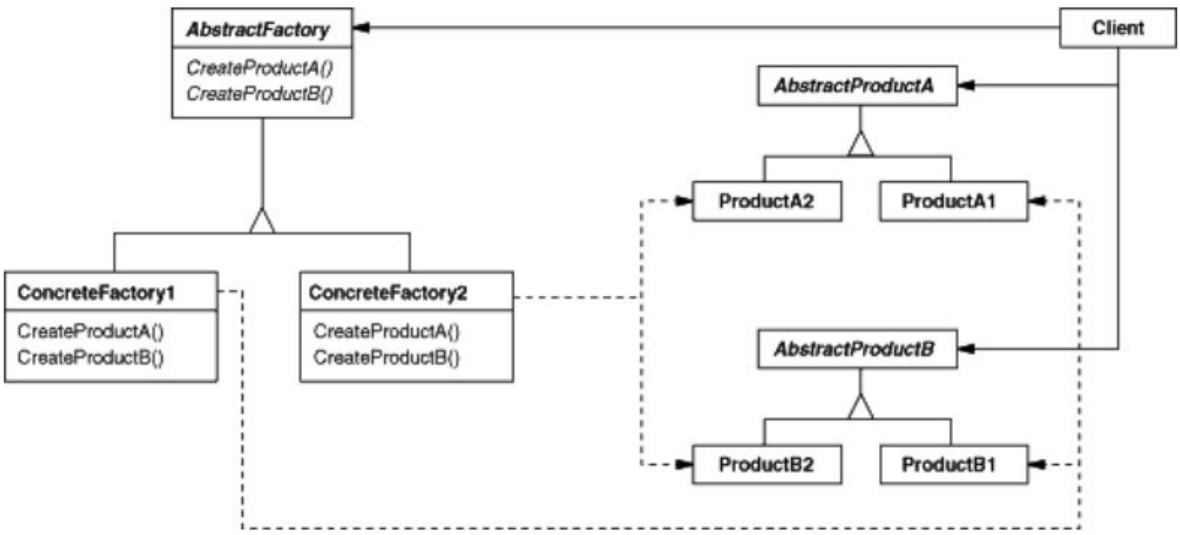
1. Aísla clases concretas. El patrón Abstract Factory ayuda a controlar las clases de objetos que crea una aplicación. Debido a que una fábrica encapsula la responsabilidad y el proceso de creación de objetos de producto, aísla a los clientes de las clases de implementación. Los clientes manipulan las instancias a través de sus interfaces abstractas. Los nombres de clase de producto están aislados en la implementación de la fábrica concreta; No aparecen en el código del cliente.
2. Facilita el intercambio de familias de productos. La clase de una fábrica concreta aparece solo una vez en una aplicación, es decir, donde se instancia. Esto facilita cambiar la fábrica concreta que utiliza una aplicación. Puede usar diferentes configuraciones de productos simplemente cambiando la fábrica concreta. Debido a que una fábrica abstracta crea una familia completa de productos, toda la familia de productos cambia a la vez.

3. Promueve la consistencia entre los productos. Cuando los objetos de productos en una familia están diseñados para trabajar juntos, es importante que una aplicación use objetos de una sola familia a la vez. AbstractFactory hace que sea fácil de aplicar.

Metodología

1. Definir los productos abstractos.
2. Definir los productos concretos.
3. Definir las fábricas concretas o familia de productos.
4. Definir la fábrica abstracta.
5. Implementar las llamadas o instancias de objetos desde el cliente.
6. Verificar la salida.

Estructura



Componentes

La estructura de Abstract Factory puede resultar confusa ya que tiene muchos componentes y éstos aparentemente se mezclan entre sí. Para comprender mejor cómo funciona este patrón, a continuación se explica cada componente:

- **Cliente (Client):** La clase que llamará a la fábrica adecuada ya que necesita crear uno de los objetos que provee la fábrica, es decir, Cliente lo que quiere es obtener una instancia de alguno de los productos (ProductoA, ProductoB).
- **Fábricas Abstractas (Abstract Factory):** Es la definición de las interfaces de las fábricas. Debe de proveer un método para la obtención de cada objeto o producto abstracto que pueda crear. ("crearProductoA()" y "crearProductoB()")
- **Fábricas Concretas (Concrete Factory):** Estas son las diferentes familias de productos. Provee de la instancia concreta de la que se encarga de crear. De esta forma podemos tener una fábrica que cree los elementos gráficos para Windows y otra que los cree para Linux, pudiendo poner fácilmente (creando una nueva) otra que los cree para MacOS, por ejemplo.
- **Producto abstracto (Abstract Product):** Definición de las interfaces para la familia de productos genéricos. En el diagrama son "ProductoA" y "ProductoB". En un ejemplo de interfaces gráficas podrían ser todos los elementos: Botón, Ventana, Cuadro de Texto, Combo, etc. El cliente trabajará directamente sobre esta interfaz, que será implementada por los diferentes productos concretos.
- **Producto concreto (Concrete Product):** Implementación de los diferentes productos abstractos. Podría ser por ejemplo "BotónWindows" y "BotónLinux". Como ambos implementan "Botón" el cliente no sabrá si está en Windows o Linux, puesto que trabajará directamente sobre la superclase o interfaz.

Aplicaciones

El patrón Abstract Factory está aconsejado cuando se prevé la inclusión de nuevas familias de productos, pero puede resultar contraproducente cuando se añaden nuevos productos o cambian los existentes, puesto que afectaría a todas las familias creadas. Algunos casos específicos son:

- Cuando el sistema necesita ser independiente de cómo se crea, compone y representa su objeto.
- Cuando la familia de objetos relacionados tiene que usarse en conjunto, entonces esta restricción debe hacerse cumplir.
- Cuando desee proporcionar una biblioteca de objetos que no muestre implementaciones y solo revele interfaces.

- Cuando el sistema necesita ser configurado con uno de una familia múltiple de objetos.

Ejemplos

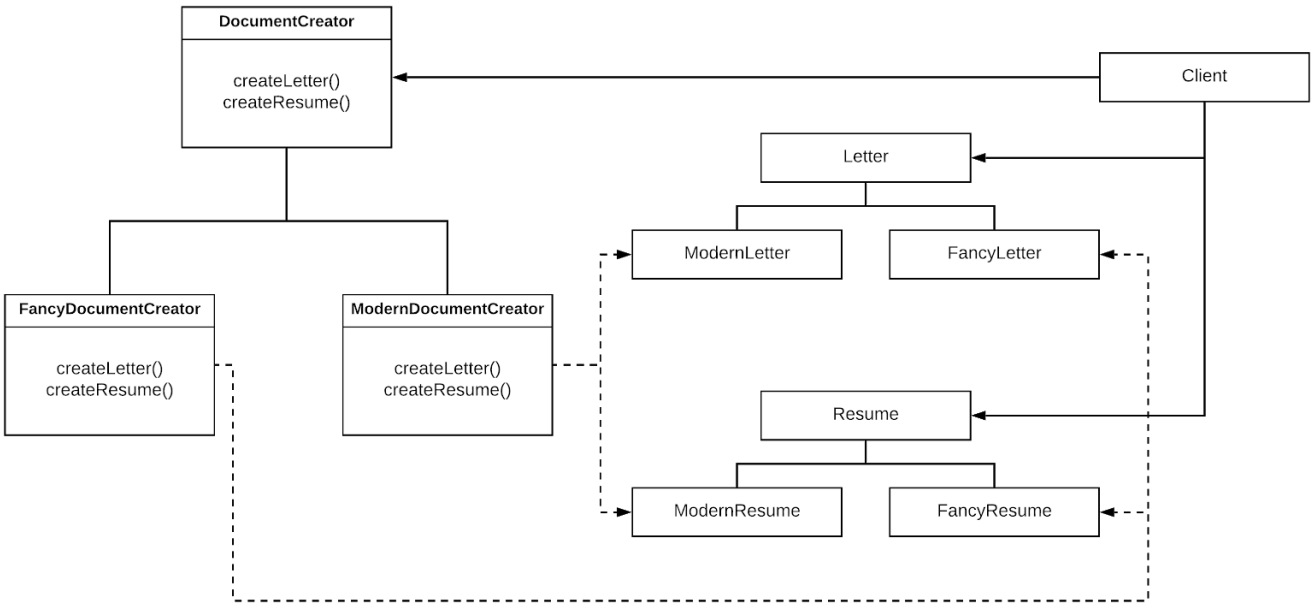
El siguiente programa ayuda a implementar el patrón de fábrica abstracta.

```
class Window:
    __toolkit = ""
    __purpose = ""

    def __init__(self, toolkit, purpose):
        self.__toolkit = toolkit
        self.__purpose = purpose

    def getToolkit(self):
        return self.__toolkit

    def getType(self):
        return self.__purpose

class GtkToolboxWindow(Window):
    def __init__(self):
        Window.__init__(self, "Gtk", "ToolboxWindow")

class GtkLayersWindow(Window):
    def __init__(self):
        Window.__init__(self, "Gtk", "LayersWindow")

class GtkMainWindow(Window):
    def __init__(self):
        Window.__init__(self, "Gtk", "MainWindow")

class QtToolboxWindow(Window):
    def __init__(self):
        Window.__init__(self, "Qt", "ToolboxWindow")

class QtLayersWindow(Window):
    def __init__(self):
        Window.__init__(self, "Qt", "LayersWindow")

class QtMainWindow(Window):
    def __init__(self):
        Window.__init__(self, "Qt", "MainWindow")

# Abstract factory class
class UIFactory:
```

```
    def getToolboxWindow(self): pass
    def getLayersWindow(self): pass
    def getMainWindow(self): pass

class GtkUIFactory(UIFactory):
    def getToolboxWindow(self):
        return GtkToolboxWindow()
    def getLayersWindow(self):
        return GtkLayersWindow()
    def getMainWindow(self):
        return GtkMainWindow()

class QtUIFactory(UIFactory):
    def getToolboxWindow(self):
        return QtToolboxWindow()
    def getLayersWindow(self):
        return QtLayersWindow()
    def getMainWindow(self):
        return QtMainWindow()

if __name__ == "__main__":
    gnome = True
    kde = not gnome

    if gnome:
        ui = GtkUIFactory()
    elif kde:
        ui = QtUIFactory()

    toolbox = ui.getToolboxWindow()
    layers = ui.getLayersWindow()
    main = ui.getMainWindow()

    print "%s:%s" % (toolbox.getToolkit(), toolbox.getType())
    print "%s:%s" % (layers.getToolkit(), layers.getType())
    print "%s:%s" % (main.getToolkit(), main.getType())
```

Practicas

Un ejemplo de esto sería una clase de fábrica abstracta DocumentCreator que proporciona interfaces para crear una serie de productos (por ejemplo, createLetter() y createResume()). El sistema tendría cualquier número de versiones concretas derivadas de la clase DocumentCreator como FancyDocumentCreator o ModernDocumentCreator, cada una con una implementación diferente de createLetter() y createResume() que crearía un objeto correspondiente como FancyLetter o ModernResume. Cada uno de estos productos se deriva de una clase abstracta simple como Carta o Currículum vitae que el cliente conoce. El código del cliente obtendría una instancia apropiada de DocumentCreator y llamaría a sus métodos de fábrica. Cada uno de los objetos resultantes se crearía a partir de la misma implementación de DocumentCreator y compartiría un tema común (todos serían objetos elegantes o modernos). El cliente solo necesitaría saber cómo manejar la clase abstracta de Letter o Resume, no la versión específica que obtuvo de la fábrica de concreto.



Conclusiones

Con el desarrollo de este presente documento e investigación correspondiente, se concluye que los patrones de diseño resultan ser muy importantes, debido a que, cada uno de ellos está orientado a un problema que ocurre infinidad de veces, así como su correspondiente solución. De modo que la implementación de los patrones de diseño pueden ayudar a utilizar esa misma solución para dicho problema en particular, sin la necesidad de complicaciones pensándolo desde cero.

Los patrones ayudan a estandarizar el código, haciendo que el diseño sea más comprensible para otros programadores. Son muy buenas herramientas, y como programadores, siempre deberíamos usar las mejores herramientas a nuestro alcance.

Referencias

[1] Oscar Javier Blancarte (2016). Introducción a los Patrones de Diseño. Ciudad de México, México.

[2] John Vlissides, Ralph Johnson, Richard Helm, Erich Gamma (1994). Design Patterns: Elements of Reusable Object-Oriented Software

[3] Ecured.”Abstract factory”. Consultado en: https://www.ecured.cu/Abstract_Factory

[4] Javatpoint. ”Abstract Factory Pattern”. Consultado en: https://www.javatpoint.com/abstract-factory-pattern