Algoritmos y Estructuras de Datos II

Trabajo Práctico N° 1	Unidad 1
Modalidad: Semi -Presencial	**Estratégica Didáctica:** Trabajo individual
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Via Classroom.
Carácter de Trabajo: Obligatorio – Con Nota	**Fecha Entrega:** A confirmar por el Docente.

MARCO TEÓRICO

Responder el siguiente cuestionario en función de la bibliografía prueba obligatoria.

1. Describir los Tipos de Patrones.

Los patrones de diseño son soluciones probadas para problemas comunes en el desarrollo de software. Se pueden clasificar según su propósito. Los tipos principales mencionados en el material son:

Patrones de Creación: Se centran en los mecanismos para crear objetos. Su objetivo es incrementar la flexibilidad y la reutilización del código.

Patrones Estructurales: Explican cómo ensamblar objetos y clases en estructuras más grandes y complejas, manteniendo la eficiencia y flexibilidad.

Patrones de Comportamiento: Se encargan de la comunicación efectiva y la asignación de responsabilidades entre los distintos objetos del sistema.

Patrones Arquitectónicos: Son patrones de más alto nivel que se pueden usar para diseñar la arquitectura de una aplicación completa. Ejemplos como MVC, MVVM y Microservicios.

2. Que es un patrón de Software

Un patrón de software es, básicamente, una solución probada y documentada que se aplica a un problema que aparece de forma recurrente en el desarrollo de software.

No es un código terminado, sino más bien una pauta o descripción de cómo estructurar el código de manera efectiva para resolver un tipo específico de problema

Al usarlos, los desarrolladores evitan "reinventar la rueda" y aprovechan soluciones que ya han sido validadas en muchos proyectos.

3. Clasificar los Patrones de Software

Los patrones de software se pueden clasificar en varias categorías según su área de aplicación. Basándonos en el material, las clasificaciones principales son:

Patrones de diseño creacionales (Ej. Singleton, Builder, Factory Method, Abstract Factory, Prototype)

Patrones de diseño estructurales (Ej. Adapter, Decorator, Proxy, Composite, Facade)

Patrones de diseño de comportamiento (Ej. Observer, Strategy, Template Method, Command, State).

Patrones de diseño arquitectónicos (Ej. MVC, MVVM, Repositorio, Inyección de dependencias) .

Patrones de diseño de concurrencia (Ej. Bloqueo (Locking), Pool de objetos, Productor-Consumidor)

4. Clasificar los Patrones de Creación y realizar una Descripción

Los patrones de creación se ocupan de los mecanismos para instanciar objetos de forma flexible, aislando los detalles de la creación. Los principales patrones de creación descritos son:

Singleton: Garantiza que una clase tenga una única instancia en todo el sistema y proporciona un punto de acceso global a ella. Es útil para controlar el acceso a un recurso compartido, como una base de datos.

Builder: Permite construir objetos complejos paso a paso. Separa el proceso de construcción del objeto de su representación final, lo que es ideal cuando un objeto tiene muchas configuraciones y se quiere evitar un "constructor monstruoso".

Factory Method: Define una interfaz (un método) para crear objetos, pero deja que sean las subclases las que decidan qué clase concreta instanciar.

Abstract Factory: Proporciona una interfaz para crear familias de objetos que están relacionados entre sí (por ejemplo, productos de un mismo estilo, como Silla, Sofá y Mesilla), sin necesidad de especificar sus clases concretas.

Prototype: Permite crear nuevos objetos copiando o clonando un objeto existente, en lugar de tener que crearlos desde cero.

5. Dar un ejemplo de uso de cada uno.

Singleton: crea una clase Menu. Usa un constructor privado y un método estático **getInstance()** para asegurar que solo exista una instancia del menú en toda la aplicación, evitando duplicados.

Builder: Se usa para construir un MenuItem. La clase **MenuBuilder** permite configurar el objeto por partes (nombre, descripción, precio) y luego obtener el objeto final con el método **build().**

Factory Method: Una clase base **MenuCreator** que tiene el método **createMenuItem().** Las clases hijas, implementan ese método para decidir qué producto específico deben crear.

Abstract Factory: Se crean familias de productos (ej. "Comida Rápida" o "Muebles Modernos"). Asegurando que los productos creados sean de la misma familia y compatibles entre sí.

Prototype: Un método virtual **clone()**. La clase implementa clone() para crear una copia de sí misma. Esto permite crear nuevos objetos simplemente clonando un prototipo existente, en lugar de construir uno nuevo desde cero.

Marco Práctico Integrado Python - SQLite

Marco Práctico: Realizar en Python

Patrón de Diseño Abstract Factory

Abstract Factory es un patrón de diseño creacional, que te permite producir familias de objetos relaciondos con sus clases concretas. Para hacer más simple la explicación tomaremos el siguiente ejemplo.

Problema

Imagina que estás creando un simulador de una tienda de muebles. Tú código consiste en las clases que representan lo siguiente:

1. Una familia de productos relacionados tal como: Chair + Sofa + CoffeTable.
2. La colleción de objetos puede tener varias combinaciones, por ejemplo: Chair + Sofa + CoffeTable que están disponibles en sus diferentes variaciones: Modern, Victorian, ArcDeco.

Necesitas una forma de crear los objetos muebles individuales de tal forma que sea posible agruparse con los otros objetos de su misma familia, a razón es porque los clientes se enojan cuando no reciben una agrupación de muebles cuando lo requieren.

Considerando que tu no quieres cambiar el código existente cada que se añaden nuevos productos o familia de productos al programa y por su puesto esto no es para nada una buena práctica. Los vendedores de la mueblería actualizan los catálogos con mucha frecuencia y tu no debes hacer cambios al código cada vez que se esto sucede.

Implementar la Solución

3. Se Solicita Diseñar una Aplicación en C++ que de respuesta a este problema.

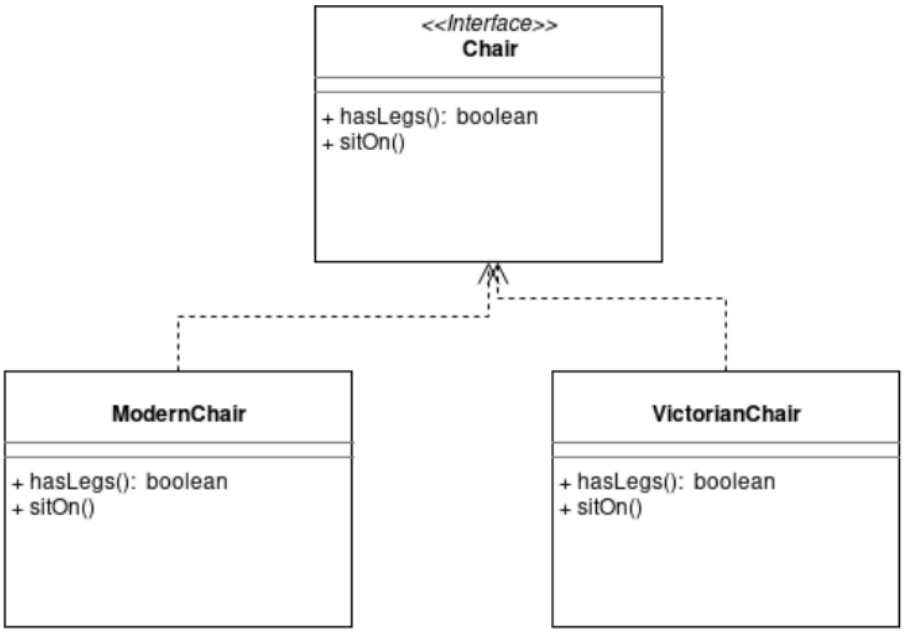
```cpp
#include <iostream>
#include <string>
using namespace std;

// ====================================
// 1. PRODUCTOS ABSTRACTOS (uno por tipo)
// ====================================
class Chair {
public:
    virtual void sitOn() const = 0;
    virtual ~Chair() = default;
};

class Sofa {
public:
    virtual void lieOn() const = 0;
    virtual ~Sofa() = default;
};

class CoffeeTable {
public:
    virtual void placeCup() const = 0;
    virtual ~CoffeeTable() = default;
};

// ====================================
// 2. PRODUCTOS CONCRETOS
// ====================================
// --- MODERN ---
class ModernChair : public Chair { void sitOn() const override { cout << "Silla MODERNA: minimalista\n"; } };
class ModernSofa : public Sofa { void lieOn() const override { cout << "Sofá MODERNO: líneas limpias\n"; } };
class ModernCoffeeTable : public CoffeeTable { void placeCup() const override { cout << "Mesa MODERNA: vidrio\n"; } };

// --- VICTORIAN ---
class VictorianChair : public Chair { void sitOn() const override { cout << "Silla VICTORIANA: tallada\n"; } };
class VictorianSofa : public Sofa { void lieOn() const override { cout << "Sofá VICTORIANO: botones\n"; } };
class VictorianCoffeeTable : public CoffeeTable { void placeCup() const override { cout << "Mesa VICTORIANA: madera\n"; } };

// --- ART DECO ---
class ArtDecoChair : public Chair { void sitOn() const override { cout << "Silla ART DECO: geométrica\n"; } };
class ArtDecoSofa : public Sofa { void lieOn() const override { cout << "Sofá ART DECO: cromo\n"; } };
class ArtDecoCoffeeTable : public CoffeeTable { void placeCup() const override { cout << "Mesa ART DECO: mármol\n"; } };

```

```cpp

// ======================================
// 3. ABSTRACT FACTORY
// ======================================
class FurnitureFactory {
public:
    virtual Chair* createChair() const = 0;
    virtual Sofa* createSofa() const = 0;
    virtual CoffeeTable* createCoffeeTable() const = 0;
    virtual ~FurnitureFactory() = default;
};

// ======================================
// 4. CONCRETE FACTORIES
// ======================================
class ModernFactory : public FurnitureFactory {
public:
    Chair* createChair() const override { return new ModernChair(); }
    Sofa* createSofa() const override { return new ModernSofa(); }
    CoffeeTable* createCoffeeTable() const override { return new ModernCoffeeTable(); }
};

class VictorianFactory : public FurnitureFactory {
public:
    Chair* createChair() const override { return new VictorianChair(); }
    Sofa* createSofa() const override { return new VictorianSofa(); }
    CoffeeTable* createCoffeeTable() const override { return new VictorianCoffeeTable(); }
};

class ArtDecoFactory : public FurnitureFactory {
public:
    Chair* createChair() const override { return new ArtDecoChair(); }
    Sofa* createSofa() const override { return new ArtDecoSofa(); }
    CoffeeTable* createCoffeeTable() const override { return new ArtDecoCoffeeTable(); }
};

```

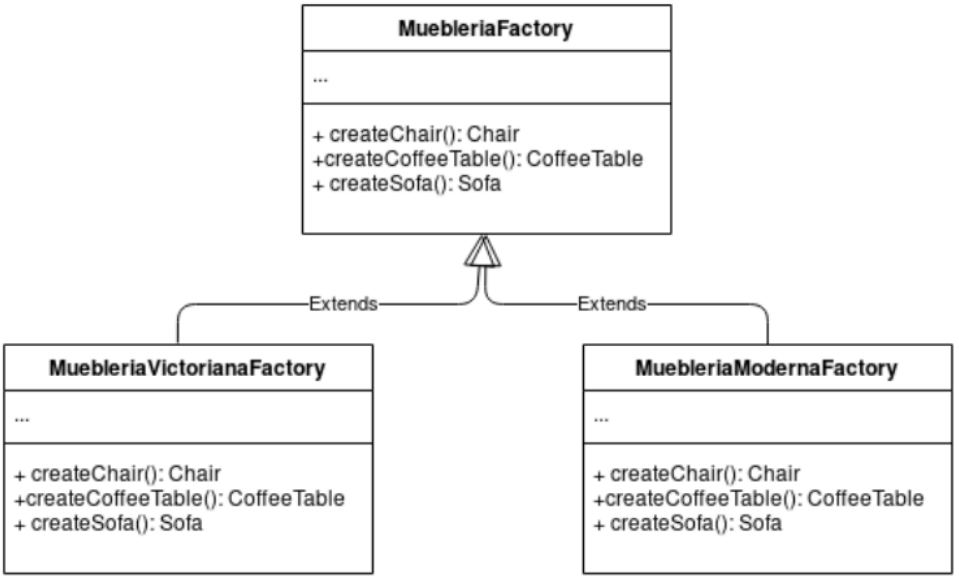
```cpp

// ========================================
// 5. CLIENTE (Tienda)
// ========================================
void orderFurnitureSet(const FurnitureFactory& factory) {
    Chair* chair = factory.createChair();
    Sofa* sofa = factory.createSofa();
    CoffeeTable* table = factory.createCoffeeTable();

    cout << "\n=== CONJUNTO COMPLETO ===\n";
    chair->sitOn();
    sofa->lieOn();
    table->placeCup();

    delete chair; delete sofa; delete table;
}

// ========================================
// 6. MAIN - INTERACTIVO
// ========================================
int main() {
    string style;
    cout << "Estilo (modern/victorian/artdeco): ";
    cin >> style;

    FurnitureFactory* factory = nullptr;

    if (style == "modern") factory = new ModernFactory();
    else if (style == "victorian") factory = new VictorianFactory();
    else if (style == "artdeco") factory = new ArtDecoFactory();
    else { cout << "Estilo no válido.\n"; return 1; }

    orderFurnitureSet(*factory);
    delete factory;
    return 0;
}
```

Material Xtra.

Lo primero que el patrón Abstract Factory sugiere es que explicitamente las interfaces se declaren por cada producto distinto de la familia del producto. Después puedes hacer las variantes de los productos siguiendo estás interfaces. Por ejemplo, todas las variantes que pueden implementar la interface Chair

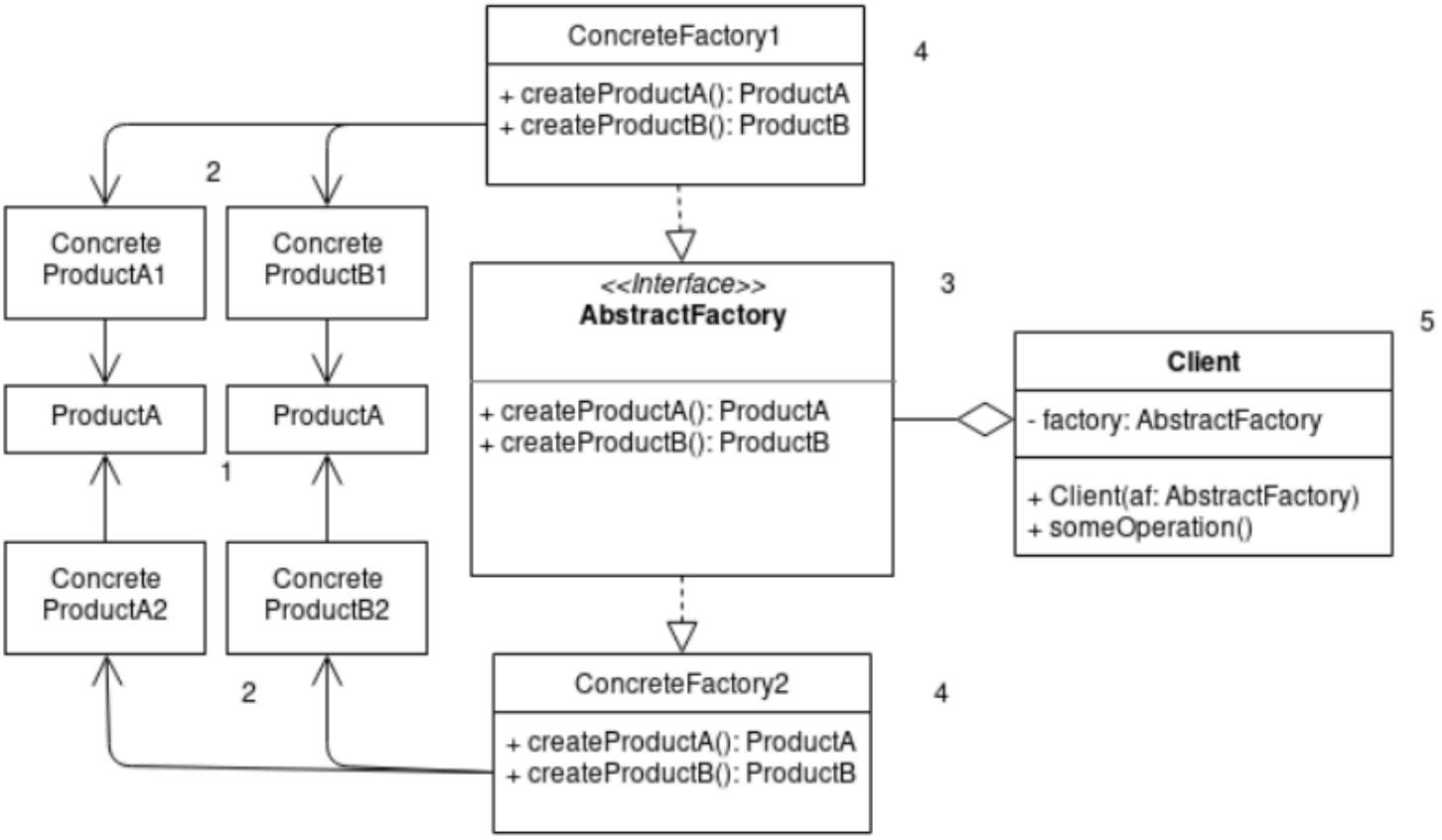


4. En el siguiente diagrama podrás ver la interfaz Abstract Factory, una interfaz con una lista de métodos para todos los productos que son parte de la familia de productos. Estos métodos regresan un tipo de producto abstracto representado por las interfaces que se han mostrado previamente: Chair, Sofa, etc.



Por cada variante de una familia de productos, creamos una clase fábrica separada basada en la interfaz AbstractFactory. Una Fábrica es una clase que regresa productos de un tipo particular. Por ejemplo, MuebleriamodernaFactory puede crear objetos de tipo ModernChair, ModernSofa y ModernCoffeeTable.

Estructura base



5. Abstract Products declara interfaces para un conjunto de productos distintos pero relacionados cada cual con su familia.

6. Concrete Products: son varias implementaciones de productos abstractos, agrupados por variantes. Cada producto abstracto debe ser implementado en todas las variantes dadas (Victorian/Modern).

7. Abstract Factory: Interfaz que declara un conjunto de métodos para crear cada uno de los productos abstractos.

8. Concrete Factories: Implementa la creación de métodos de abstract factory. Cada fábrica concreta corresponde a una variante específica de productos y crea solo esas variantes de productos.

9. Usa la Abstract Factory cuando tú código necesite trabajar con varias familias de productos, pero no quieras que dependa de las clases concretas de estos productos, pueden ser desconocidos de antemano o simplemente permitir extensibilidad futura.

Lic. Oemig José Luis.