UNSA
UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS
DEPARTAMENTO ACADÉMICO DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
ESCUELA PROFESIONAL DE CIIENCIAS DE LA COMPUTACIÓN

CIENCIAS DE LA COMPUTACIÓN I

Tema 30: Diseño orientado a objetos

Tema 31: Patrones creacionales

Tema 32: Patrones estructurales

UNIDAD IV
Capítulo XIV: Patrones de Diseño
Periodo Académico: 2020C
Semestre: 02

Docente: Mg. Maribel Molina Barriga

Agenda

- Tema 30: Diseño orientado a objetos
- Tema 31: Patrones creacionales
- Tema 32: Patrones estructurales

Tema 30: Diseño orientado a objetos

Requisitos para implementar patrones de diseño

- Programación en un Lenguaje de Programación como el C++
- Programación OO
  - Herencia
  - Polimorfismo
  - Clases Abstractas
- UML básico

Los Patrones de Diseño (Designs Patterns)

- Son modelos de trabajo enfocados a dividir un problema en partes, para así abordar cada parte del problema por separado y simplificar la resolución.

- Un patrón de diseño es un conjunto de reglas que describen cómo afrontar tareas y solucionar problemas que surgen durante el desarrollo de software.

Los Patrones de Diseño (Designs Patterns)

- Los patrones de Diseño encapsulan forman comunes de resolver problemas
- Solución simple, reusable, fácil mantenimiento
- Los patrones de diseño dados a conocer con: Erich Gamma, Richard Helm, Ralph Johnson y John Vissides en el libro ***"Desing Patterns: Elements of Reusable Objet-Oriented Software"***
- Actualmente existen 23 patrones esenciales a conocer
- Es un catálogo de soluciones que muestran interacciones entre los objetos, que los programadores encuentran útiles
- Nuevos patrones se han creado, algunos para áreas especializadas

Clasificación

- Se dividen en 3 grupos:
  - **Patrones Creacionales:** crean los objetos en un lugar de que los instanciemos directamente. Dan flexibilidad sobre que objeto necesitamos en que momento.
  - **Patrones de Estructura:** Ayudan agrupar los objetos en estructuras más grande.
  - **Patrones de comportamiento:** Ayudan a definir la comunicación entre los objetos y como se controla el flujo del programa

Ventajas

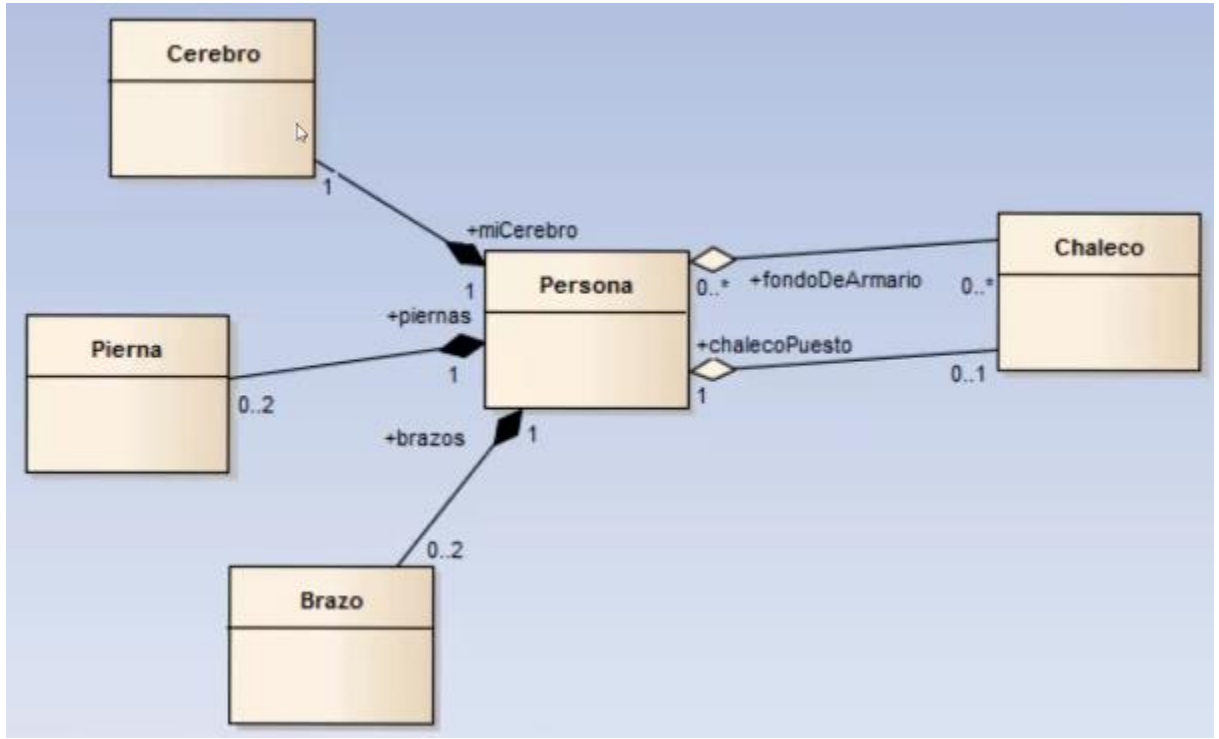
- Va ha proteger nuestro software contra el cambio (por los cambios son constantes porque ahora existen muchas modificaciones, esto para no trabajar extra)
- Mantienen a las clases separadas
- Hacen que el código sea más fácil de entender para otros programadores (esto para dar mantenimiento)
- Se utilizan principios de diseño OO bien conocidos

Algunos Principios de Diseño OO

- Existen varios principios, pero explicaremos los siguientes:
- Programar a una interfaz y no a una implementación
  - En lo alto de la jerarquía debe de existir una clase abstracta o a una interfaz
  - Así se tiene más flexibilidad de poner la implementación que sea necesaria
- Favorecer la composición (y agregación) sobre la herencia
  - La herencia puede adicionar comportamientos no deseados en las clases hijas
  - Con la composición creamos un objeto a partir de otros
  - De esta forma solo adquirimos los comportamiento de la interfaz y no todos como en la herencia

La relación composición / Agregación

	Agregación	Composición
Representación	Rombo transparente	Rombo negro
Varias asociaciones comparten los componentes	Sí	No
Destrucción de los componentes al destruir el compuesto	No	Sí
Cardinalidad del compuesto	Cualquiera	0..1 ó 1



Agregación: Es una relación en la que una de las clases representa un todo y la otra representa parte de ese todo

Composición: Es una forma más fuerte de la agregación, en la que el todo no puede existir sin sus partes

¿Cómo se implementan?
¿Cuál es la diferencia con las asociaciones?

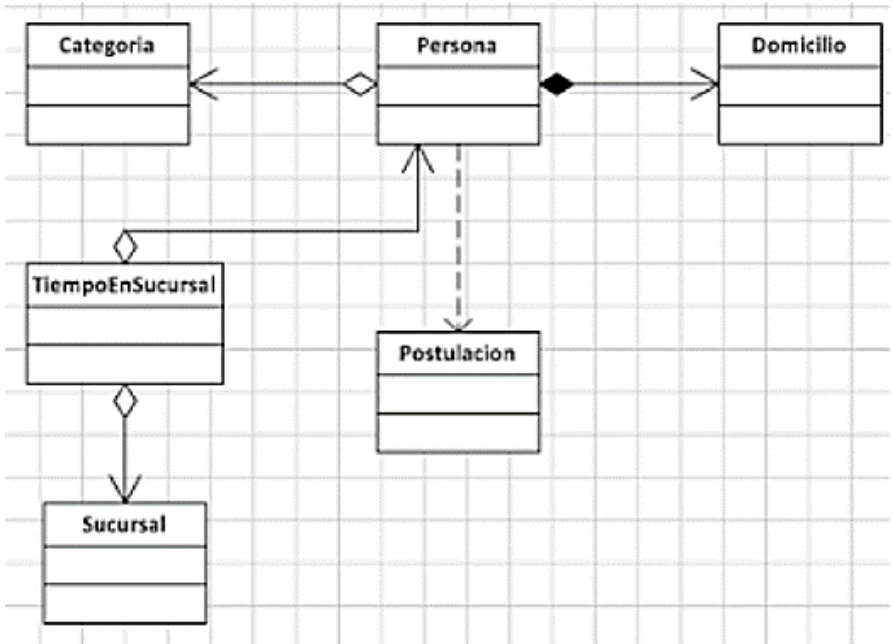
```
class Persona
{
    //Relación de composición: hago el new dentro de la clase
    Domicilio dom = new Domicilio();

    //Relación de agregación: no hago el new, ya viene desde afuera resuelto.
    Categoria cat;
    public Persona(Categoria c)
    {
        cat = c;
    }

    //Atributos de la clase
    public string Apellido { get; set; }
    public string Nombre { get; set; }
    public int Id { get; set; }

    //Relación de dependencia: no forma parte de la clase, es utilizada para hacer alguna de sus operaciones
    public void Postularse()
    {
        Postulacion p = new Postulacion();
        p.Enviar();
    }

}
```




```
class Categoria
{
    public int Id { get; set; }
    public string IdCategoria { get; set; }
}

class Domicilio
{
    public int PersonaId { get; set; }
    public string Calle { get; set; }
    public int Numero { get; set; }
}

class Postulacion
{
    public int Id { get; set; }
    public void Enviar() { }

}

class Sucursal
{
    public int Id { get; set; }
    public string NombreSucursal { get; set; }
}

class TiempoEnSucursal
{
    //Atributos relacionales
    public Sucursal Sucursal { get; set; }
    public Persona Persona { get; set; }

    //Atributos de la clase
    public DateTime FechaInico { get; set; }
    public DateTime FechaFin { get; set; }
}
```

Tema 31: Patrones creacionales

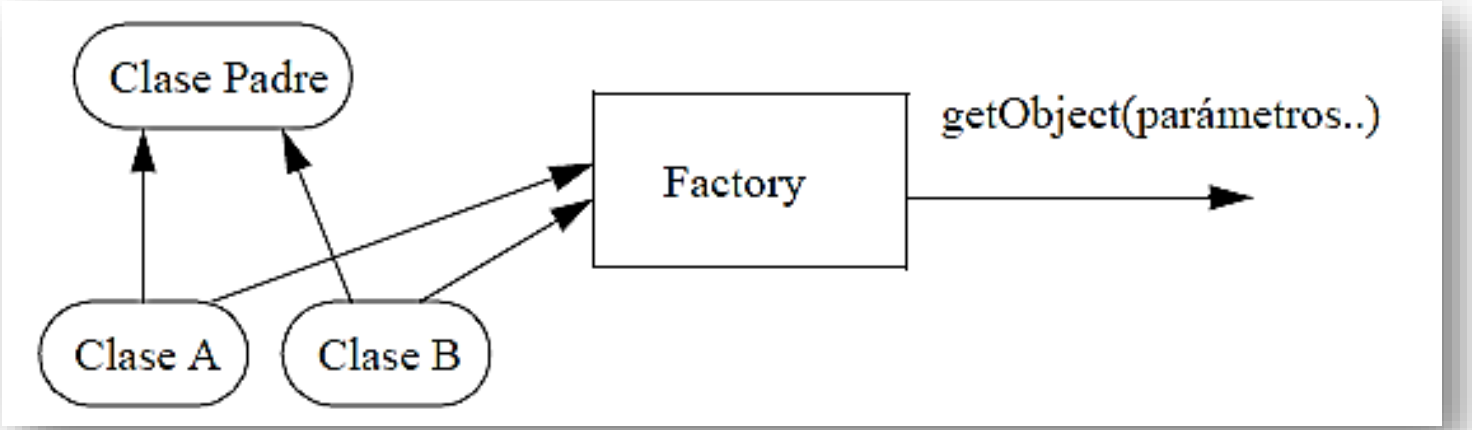
Patrones de Creación

- Se encargan de crear instancias de objetos. Los patrones de creación más conocidos son:

- Factory
- Abstract Factory
- Builder
- Prototype
- Singleton

Factory

- El objetivo de este patrón de diseño es devolver una instancia de múltiples tipos de objetos, generalmente heredan de una misma clase padre y se diferencian entre ellos por su comportamiento.
- El objeto Factory es el encargado de decidir, según los parámetros que le pasemos, el tipo de objeto que devolverá



Abstract Factory

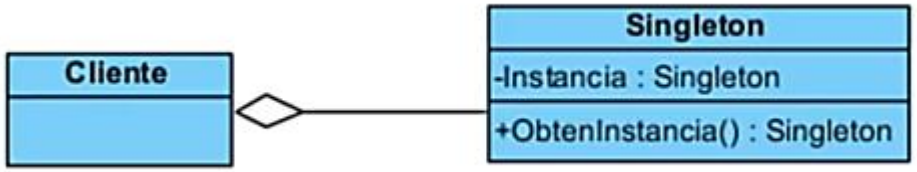
- Este patrón añade un nivel más de complejidad. Anteriormente la clase Factory devolvía objetos de diferentes tipos, en este caso, devolverá diferentes clases Factory según el parámetro enviado.

- 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.

Singleton

- Un Singleton es una clase de la que tan sólo puede haber una única instancia. Ejemplos típicos de esto son spools de impresión, servidores de bases de datos, etc..

- Estas son las formas más comunes de resolver estos problemas:
  - **Crear una variable estática** dentro de la clase que indique si una instancia ha sido o no creada. Esta solución tiene el problema de como avisar desde el constructor de que no se ha podido crear una nueva instancia.
  - **Crear una clase final:** El objetivo de esto es crear una clase final que tan sólo tenga métodos estáticos. De este modo la clase no se podrá extender. Un ejemplo de esto es la clase java.lang.Math, que agrupa métodos matemáticos de utilidad de manera que sólo haya una única forma de acceder a los mismos.
  - **Crear el Singleton con un método estático:** Esta aproximación lo que hace es hacer privado el constructor de la clase de manera que la única forma de conseguir una instancia de la misma sea con un método estático.



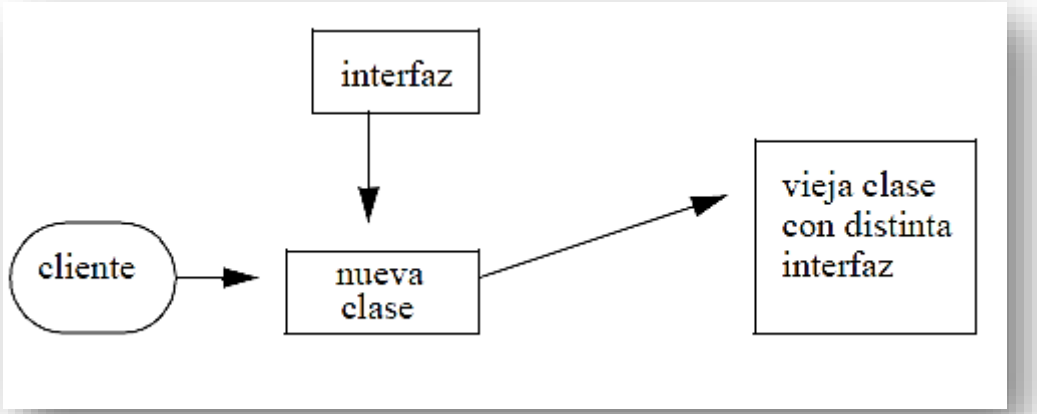
Tema 32: Patrones estructurales

Patrones Estructurales

- Los patrones estructurales describen cómo formar estructuras complejas a partir de elementos más simples. Existen dos tipos: de clase y de objeto.
- Los patrones de clase muestran cómo la herencia puede ser utilizada para proporcionar mayor funcionalidad mientras que los patrones de objeto utilizan composición de objetos o inclusión de objetos dentro de otros para proporcionar también una mayor funcionalidad.
- Los patrones más conocidos son:
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - FaÇade
  - Flyweight
  - Proxy

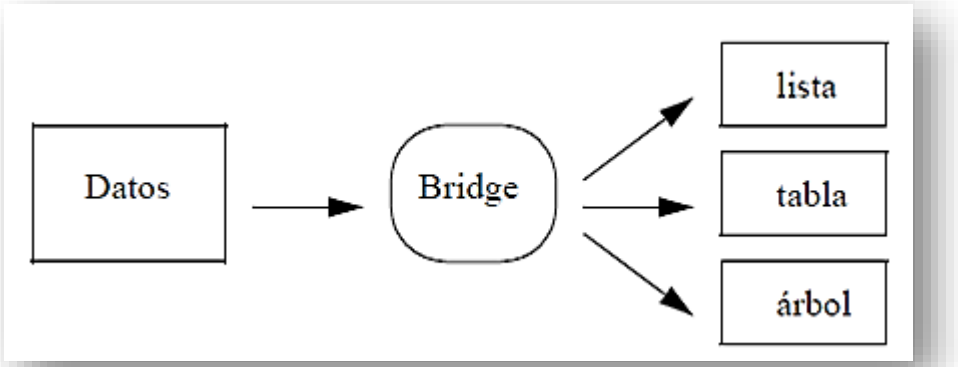
Adapter

- Su finalidad es transformar la interfaz de programación de una clase en otra. Se utilizan adaptadores cuando se desea que clases que no tienen nada que ver funcionen de la misma manera para un programa determinado.
- El concepto es escribir una nueva clase con la interfaz de programación deseada y hacer que se comunique con la clase cuya interfaz de programación era diferente.
- Existen dos formas de realizar esto, con herencia o con composición de objetos.
  - En el primer caso vamos a crear una nueva clase que heredará de la que se desea adaptar y a esta nueva clase se le agregarán los métodos necesarios para que su interfaz de programación se corresponda con la que se desea utilizar.
  - En la segunda aproximación se incluye la clase original dentro de la nueva y se crean los métodos de manera que accedan a la clase que se agregaron como atributo.



Bridge

- Un Bridge se utiliza para separar la interfaz de una clase de su implementación de forma que ambas puedan ser modificadas de manera separada, el objetivo es poder modificar la implementación de la clase sin tener que modificar el código del cliente de la misma.
- El funcionamiento del Bridge es simple, Ejemplo:
  - Supongamos datos de clientes de nuestra empresa que se desean mostrar en pantalla. En una opción de nuestro programa queremos mostrar esos datos en una lista con sus nombres, mientras que en otra queremos mostrar los datos en una tabla con nombre y apellidos.
- Lo importante de este patrón es que si ahora quisiésemos mostrar información de nuestros clientes en un árbol, no tendríamos que modificar el cliente para nada sino que en nuestro Bridge añadiríamos un parámetro que crearía el árbol.



Tema 33: Patrones de comportamiento

Patrones de Comportamiento

- Los patrones de comportamiento fundamentalmente especifican el comportamiento entre los objetos del sistema. Los patrones más conocidos son:
  - Strategy
  - Chain
  - Observer
  - Mediator
  - Template
  - Interpreter
  - Visitor
  - State
  - Command
  - Iterator

Patrón de estrategia (strategy)

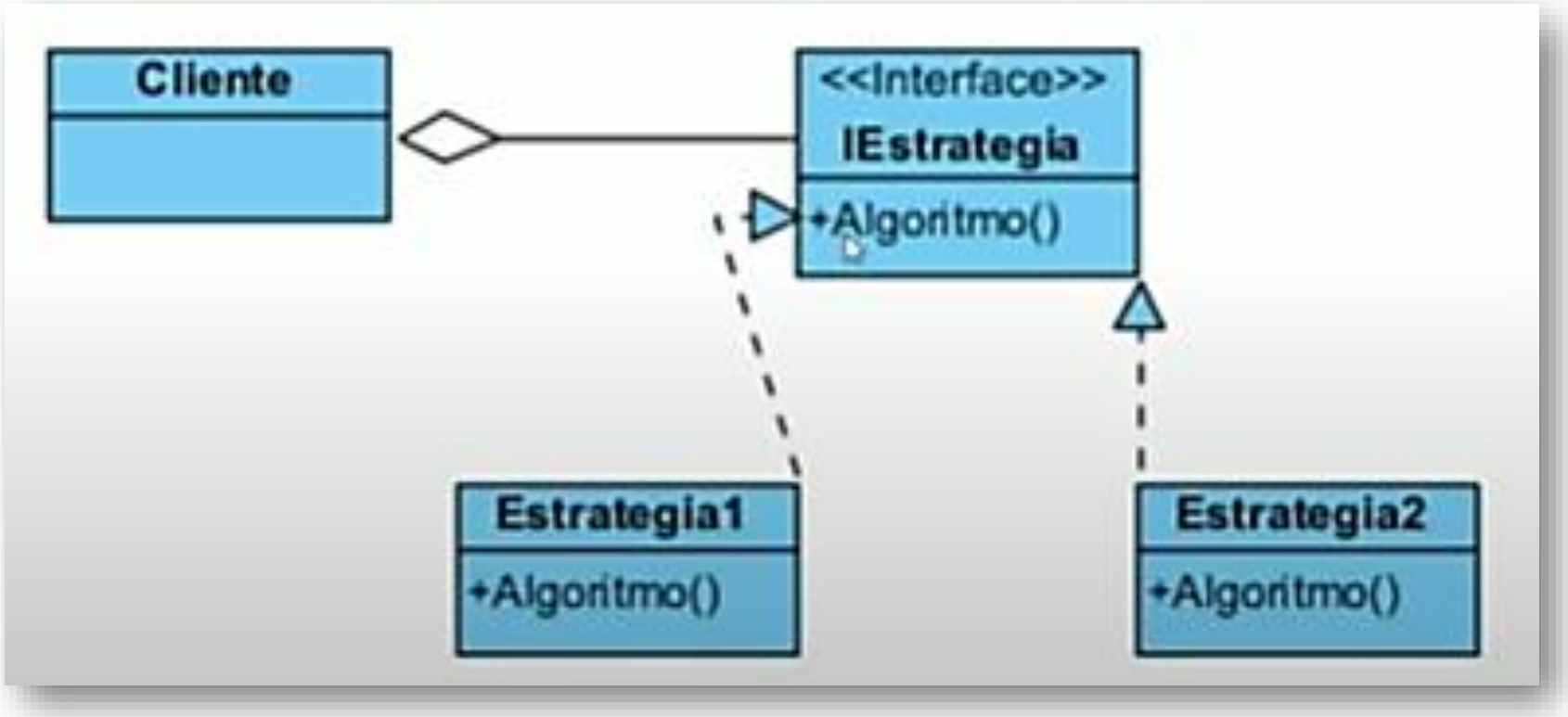
- Es el uso del polimorfismo
- Es sencillo
- Se forma con una familia de algoritmos que están encapsulados
- El cliente selecciona que algoritmo utilizar
- El objetivo es hacer esos algoritmos intercambiables y usar el mejor para cada caso
- Si tenemos un programa que provee una funcionalidad, pero existen varias formas de llevarla a cabo, se puede usar la estrategia
- Puede hacerse vía herencia o con implementación de interfaz

Donde se puede aplicar

- Salvar un archivo en diferentes formatos
- Comprensión con diferentes algoritmos
- Formas de representar información
- Partes importantes:
  - Contexto
  - Interfaz estrategia
    - Una clase que contiene la información de contexto sobre la cual trabajaran los algoritmos
    - Define la interfaz común a todas las estrategias (algoritmos)
  - Estrategia n
    - Implementación de un algoritmo en particular

Estrategia

- Diagrama a implementar



Actividades

1. Contestar las preguntas del foro en el DUTIC

Videos Complementarios

Patrón Estrategia:

https://www.youtube.com/watch?v=CEkdKz1EqOI&list=PLM-p96nOrGcbqbL_A29b0z3KUXdq2_fpn&index=3

Patrón del Singleton:

https://www.youtube.com/watch?v=Q6HJpgdkAK8&list=PLM-p96nOrGcbqbL_A29b0z3KUXdq2_fpn&index=22

Bibliografía

- C++ How to Program (10th Edition) 10th Edition, Paul J. Deitel, Harvey Deitel.
- The C++ Programming Language, Stroustrup Bjarne.

Link interesante:

- https://refactoring.guru/es/design-patterns/abstract-factory/cpp/example#example-0
- https://www.soloentendidos.com/patrones-de-diseno-de-programacion-orientada-a-objetos-1771
- https://issuu.com/aldivadyer/docs/5_uml_rev1
- **Ejemplo patron prototype:**
  https://gitlab.com/UAI-TCTD/patr-n-prototype-en-csharp/-/tree/master/Patrones.Prototype

GRACIAS …!

Ing. Maribel Molina Barriga

✉ mmolinab@unsa.edu.pe