Patrones de Diseño

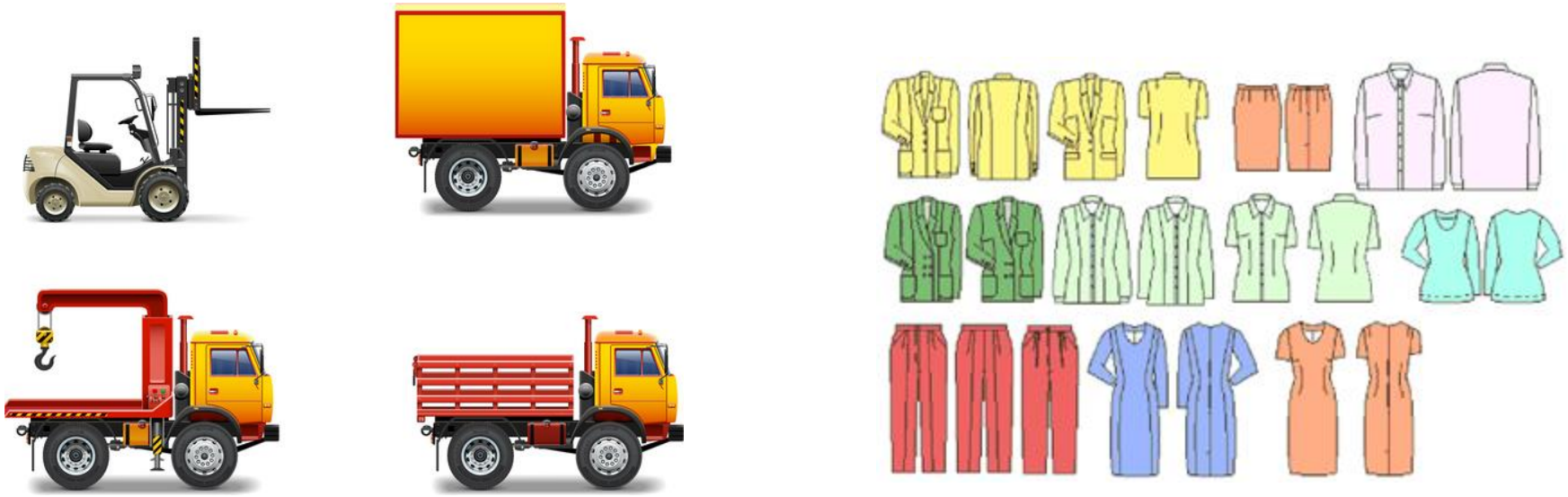
Semana 5

Agenda

- Introducción a los patrones de diseño.
- Elementos de un patrón de diseño.
- Taxonomía de patrones de diseño.
- Patrones creacionales.

Introducción a los patrones de diseño

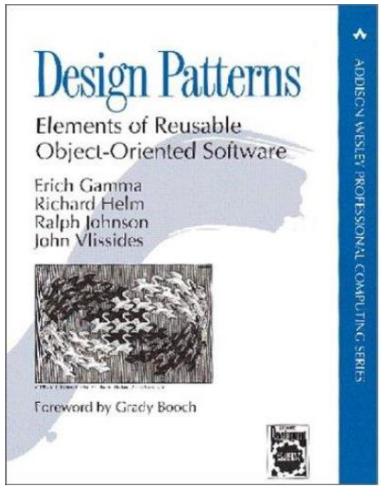
¿Qué es un patrón?



Es un modelo que sirve de muestra para sacar otra cosa igual.

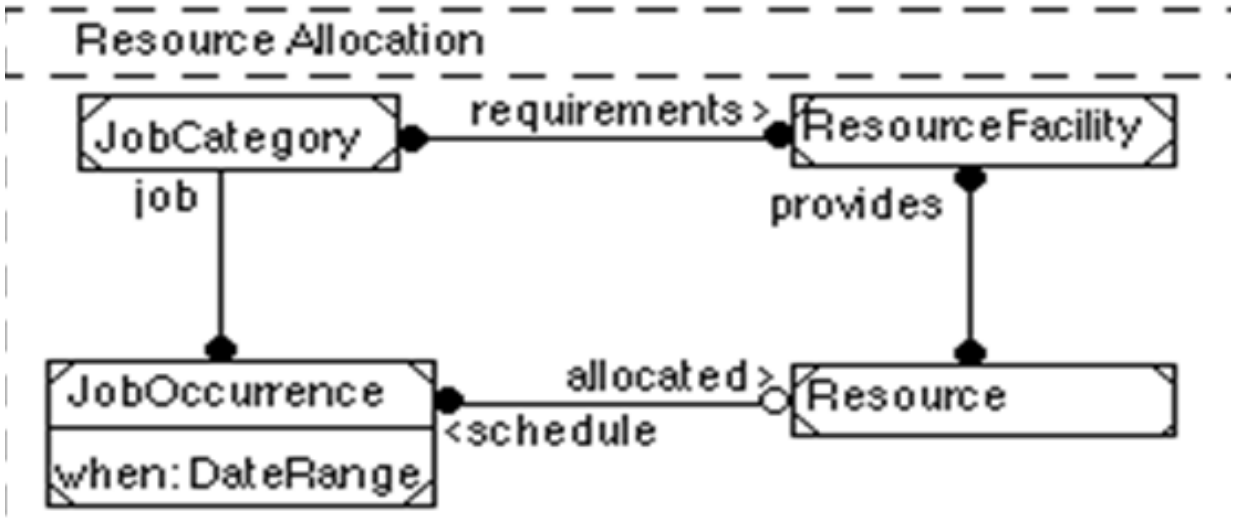
Breve historia

- 1977 – Christopher Alexander – identificó patrones de diseño en Arquitectura.
- 1987 – Cunningham and Beck – Utilizaron las ideas de Alexander para desarrollar Smalltalk-80
- 1995 – Gang of Four (GoF) publicaron su libro de Patrones de diseño.

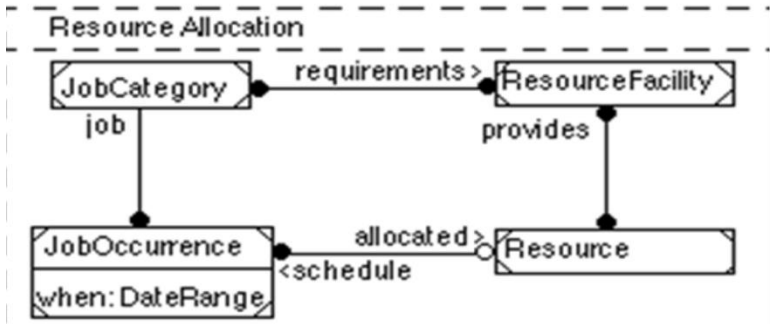


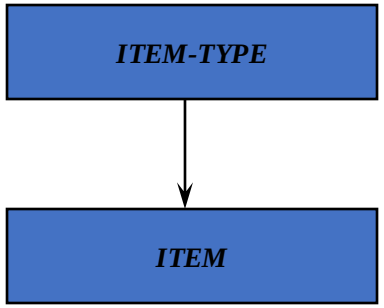
Design Pattern

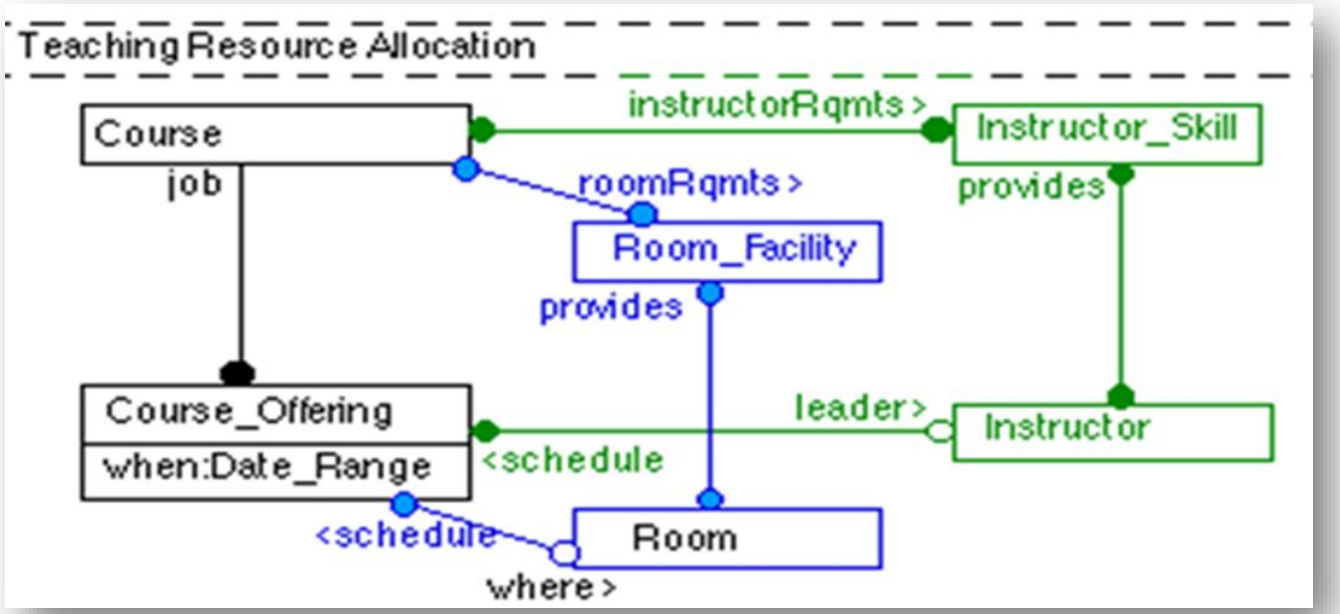
- Dupla (problema, solución)
- Los patrones facilitan la ***reutilización*** de diseños y arquitecturas de software exitosos.



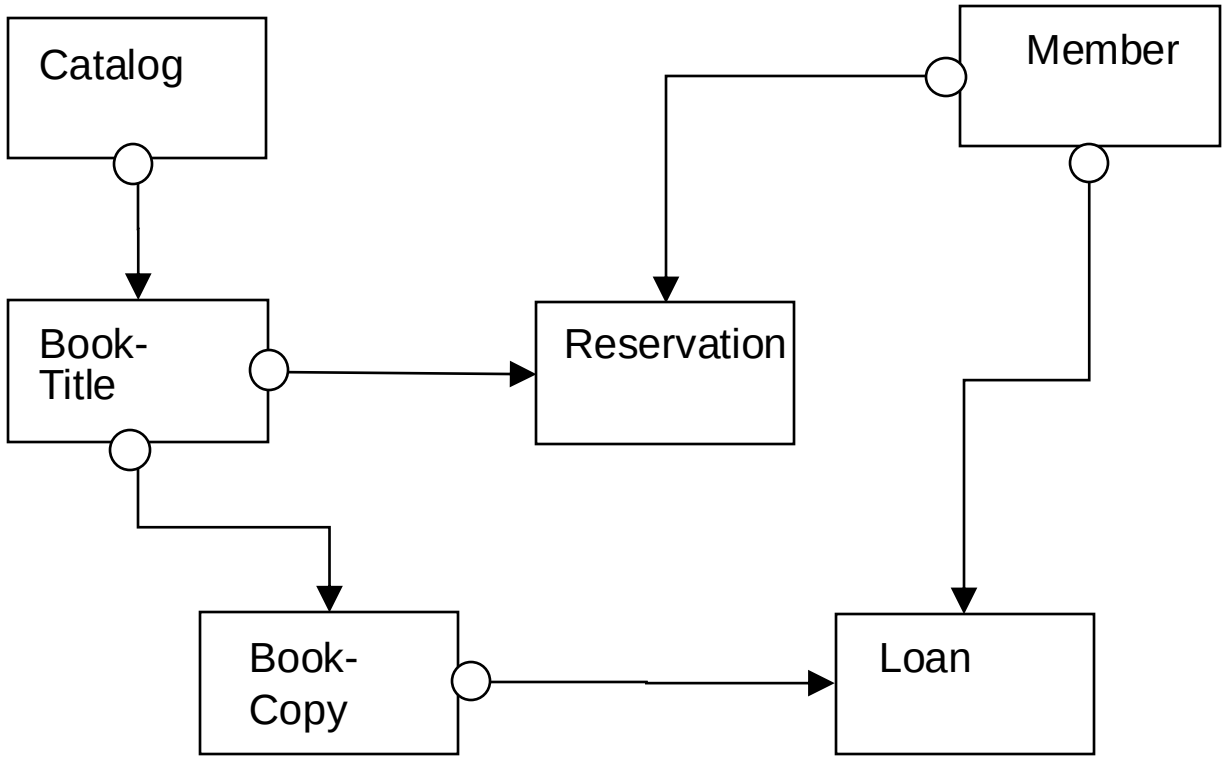
Reuse: "Type" Classes



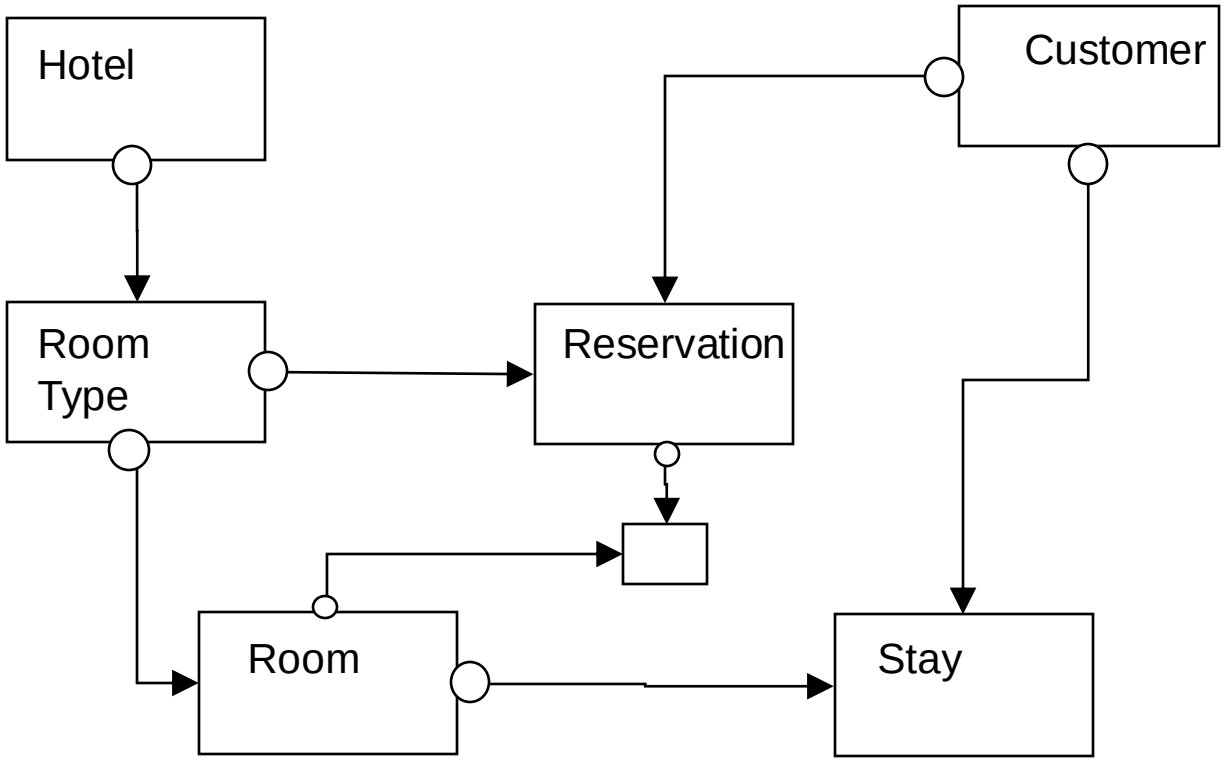




A Library Model ...

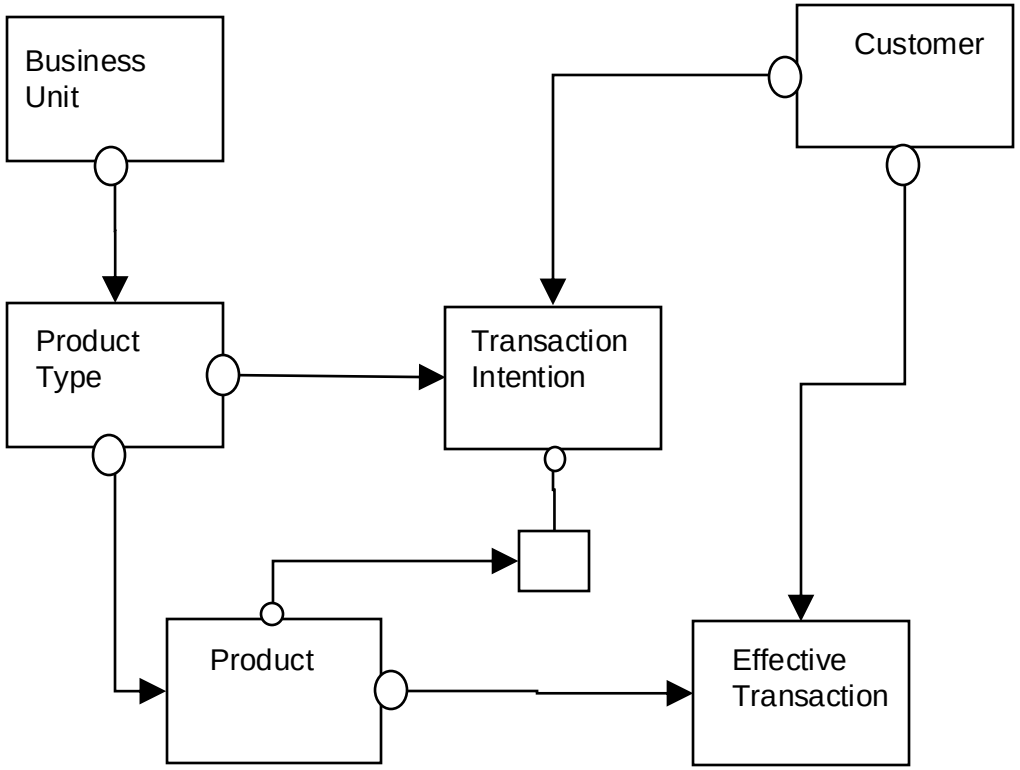


… is similar to a Hotel Model



Generic Business Model

- Avoids to re-invent the wheel over and over again.
- Allows to benefit from existing knowledge



Elementos de un patrón de diseño

Componentes de un patrón DE DISEÑO

- Nombre
  - Identifica el patrón
  - Define terminología
- Alias
- Ejemplo – del mundo real
- Contexto
- Problema
- Solución
  - Descripción en lenguaje natural
- Estructura
  - Clases, objetos y diagramas de la solución
- Diagrama de interacción (opcional)
- Consecuencias
  - Ventajas y desventajas del uso del patrón.
- Implementación
  - Parte crítica del código.
- Usos conocidos
  - Algunos sistemas que inspiraron al patrón.

Taxonomía de patrones de diseño

Tipos de patrones de diseño

- Creacionales
  - Concernientes al proceso de creación de objetos.
- Estructurales
  - Tratan sobre la composición de las clases o de los objetos.
- Comportamiento
  - Caracterizan formas en como interactúan clases u objetos y distribuyen responsabilidades.

TIPOS DE PATRONES DE DISEÑO - GOF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope: dominio sobre el cual se aplica un patrón

Purpose: que hace el patrón

Consideraciones al elegir un patrón

- **Resistencia al cambio:** Nos cuesta cambiar nuestros hábitos de desarrollo de software.
  - **Reutilización:** No estamos acostumbrados a reutilizar.

Ventajas del uso de patrones

- Proporcionan una estructura conocida por todos los programadores.
- Permiten tener una estructura de código común a todos los proyectos que implementan una funcionalidad genérica.
- Permite ahorrar tiempo en la construcción de software de características similares.
- El software construido es más fácil de comprender, mantener y extender.

Desventajas del uso de patrones

- Dependiente del paradigma a objetos.
- Difíciles de usar si **las similitudes** con los patrones creados **son escasas o si no existen.**

Patrones creacionales

Descripción

- Son patrones para abstraer y controlar la forma de como se crean los objetos en un software.
- Intentan hacer que los sistemas sean independientes de la creación, composición y representación de los objetos.
- **La parte** del sistema **encargado de la instanciación** de clases debe utilizar una interfaz común de creación **sin la necesidad de conocer** realmente la forma en **como son creados los objetos.**

Patrones creacionales

- Builder
  - Separa la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda crear representaciones diferentes.
- Prototype
  - Permite crear nuevos objetos a partir de instancias previamente creadas (duplicados).
- Singleton
  - Asegura que una clase sólo tenga una instancia y provee una forma global para acceder a esta.
- Typesafe Enum
  - Generaliza Singleton: Asegura que una clase sólo tenga un número fijo de instancias unicas.

Patrones creacionales

- Abstract Factory
  - Provee una interfaz para crear familias de objetos relacionados o dependientes pero sin especificar sus clases concretas.
- Factory Method
  - Provee una interfaz para crear un objeto pero sin mostrar la lógica de creación al cliente.

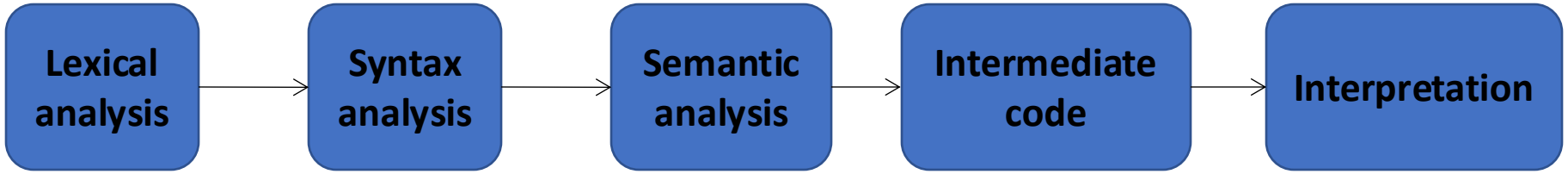
Builder

Builder

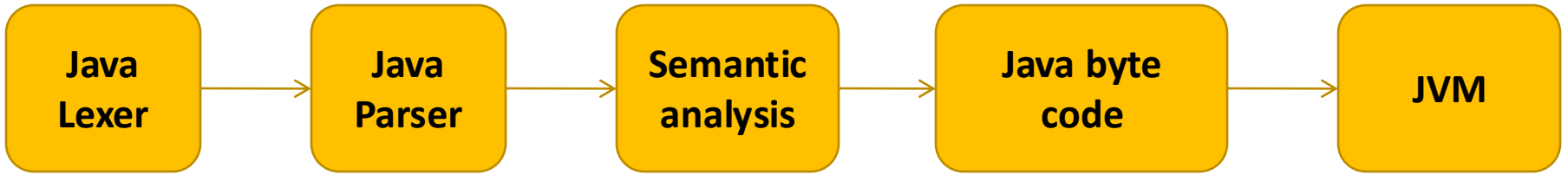
- Construye un objeto complejo mediante un paso a paso.
- El proceso puede construir distintas representaciones.

Ejemplos - Builder

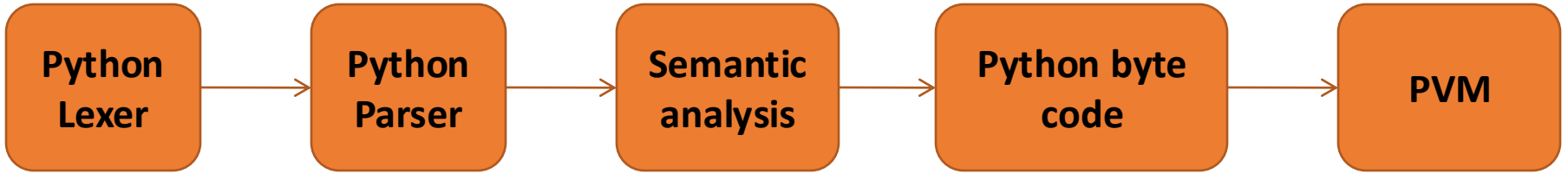
Compiler process



Java Compiler



Python Compiler



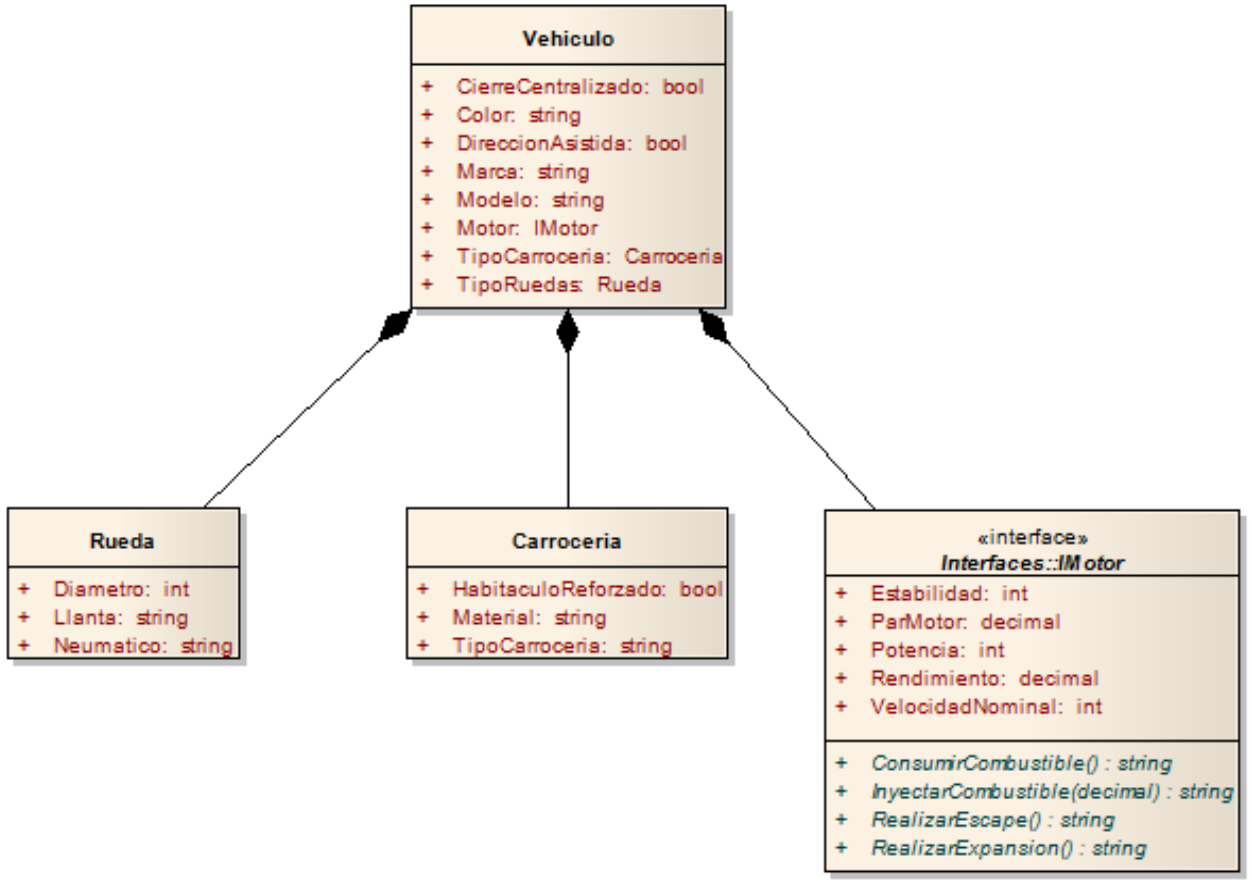
Sistema de Creación de Vehículos

- Se desea implementar un sistema que permita la construcción de vehículos.
- Si sabemos que cada vehículo tiene marca, color, modelo, motor, etc.

¿Cómo puedo comenzar?

Diagrama de clase

- Después del análisis de requerimientos se obtuvo lo siguiente:



Codificando… 1/3

```
public class Vehiculo
{
    public bool CierreCentralizado { get; set; }
    public string Color { get; set; }
    public bool DireccionAsistida { get; set; }
    public string Marca { get; set; }
    public string Modelo { get; set; }
    public IMotor Motor { get; set; }
    public Carroceria TipoCarroceria { get; set; }
    public Rueda TipoRuedas { get; set; }

    public string GetPrestaciones()
    {
        StringBuilder sb = new StringBuilder();
        string nl = Environment.NewLine;

        sb.Append("El presente vehiculo es un ").Append(Marca).Append(" ").Appen
        sb.Append(" estilo ").Append(TipoCarroceria.TipoCarroceria).Append(nl);
        sb.Append("Color: ").Append(Color).Append(nl);
        sb.Append(DireccionAsistida ? "Con " : "Sin ").Append("direccion asistid
        sb.Append(CierreCentralizado ? "Con " : "Sin ").Append("cierre centraliz
        sb.Append("Carroceria de ").Append(TipoCarroceria.Material);
        sb.Append(TipoCarroceria.HabitaculoReforzado ? " con " : " sin ").Append
        sb.Append("Ruedas con llantas ").Append(TipoRuedas.Llanta).Append(" de "
        sb.Append("Neumaticos ").Append(TipoRuedas.Neumatico);
        sb.Append("Respuesta del motor: ").Append(Motor.InyectarCombustible(100)

        return sb.ToString();
    }
}
```

Codificando… 2/3

```
// Tipo de rueda: diámetro, llanta y neumático
public class Rueda
{
    public int Diametro { get; set; }
    public string Llanta { get; set; }
    public string Neumatico { get; set; }
}
// Tipo de carrocería
public class Carroceria
{
    public bool HabitaculoReforzado { get; set; }
    public string Material { get; set; }
    public string TipoCarroceria { get; set; }
}
// Interfaz que expone las propiedades del motor
public interface IMotor
{
    string ConsumirCombustible();
    string InyectarCombustible(int cantidad);
    string RealizarEscape();
    string RealizarExpansion();
}
```

Codificando… 3/3

```
// Motor diesel, que implementará la interfaz IMotor
public class MotorDiesel : IMotor
{
    #region IMotor Members

    public string ConsumirCombustible()
    {
        return RealizarCombustion();
    }

    public string InyectarCombustible(int cantidad)
    {
        return string.Format("MotorDiesel: Inyectados {0} ml. de Gasoil.", canti
    }

    public string RealizarEscape()
    {
        return "MotorDiesel: Realizado escape de gases";
    }

    public string RealizarExpansion()
    {
        return "MotorDiesel: Realizada expansion";
    }

    #endregion

    private string RealizarCombustion()
    {
        return "MotorDiesel: Realizada combustion del Gasoil";
    }
}
```

Crear clase Abstracta 1/4


```
public abstract class VehiculoBuilder
{
    // Declaramos la referencia del producto a construir
    protected Vehiculo v;

    // Declaramos el método que recuperará el objeto
    public Vehiculo GetVehiculo()
    {
        return v;
    }

    #region Métodos Abstractos

    public abstract void DefinirVehiculo();
    public abstract void ConstruirRuedas();
    public abstract void ConstruirHabitaculo();
    public abstract void ConstruirMotor();
    public abstract void DefinirExtras();

    #endregion
}
```

Podemos crear estos dos vehículos

Citroen Xsara

A3 SportBack

Código – citroen 2/4


```
public class CitroenXsaraBuilder : VehiculoBuilder
{
    public override void DefinirVehiculo()
    {
        v = new Vehiculo();
        v.Marca = "Citroen";
        v.Modelo = "Xsara Picasso";
    }

    // Método que construirá las ruedas
    public override void ConstruirRuedas()
    {
        v.TipoRuedas = new Rueda();
        v.TipoRuedas.Diametro = 15;
        v.TipoRuedas.Llanta = "hierro";
        v.TipoRuedas.Neumatico = "Firestone";
    }

    public override void ConstruirMotor()
    {
        v.Motor = new MotorDiesel();
    }

    // Método que construirá el habitaculo
    public override void ConstruirHabitaculo()
    {
        v.TipoCarroceria = new Carroceria();
        v.TipoCarroceria.TipoCarroceria = "monovolumen";
        v.TipoCarroceria.HabitaculoReforzado = false;
        v.TipoCarroceria.Material = "acero";
        v.Color = "negro";
    }

    public override void DefinirExtras()
    {
        v.CierreCentralizado = false;
        v.DireccionAsistida = false;
    }
}
```

Código – AUDI 3/4

```
public class A3SportbackBuilder : VehiculoBuilder
{
    public override void DefinirVehiculo()
    {
        v = new Vehiculo();
        v.Marca = "Audi";
        v.Modelo = "A3 Sportback";
    }

    // Método que construirá las ruedas
    public override void ConstruirRuedas()
    {
        v.TipoRuedas = new Rueda();
        v.TipoRuedas.Diametro = 17;
        v.TipoRuedas.Llanta = "aluminio";
        v.TipoRuedas.Neumatico = "Michelin";
    }

    // Método que construirá el motor
    public override void ConstruirMotor()
    {
        v.Motor = new MotorDiesel();
    }

    // Método que construirá el habitaculo
    public override void ConstruirHabitaculo()
    {
        v.TipoCarroceria = new Carroceria();
        v.TipoCarroceria.TipoCarroceria = "deportivo";
        v.TipoCarroceria.HabitaculoReforzado = true;
        v.TipoCarroceria.Material = "fibra de carbono";
        v.Color = "plata cromado";
    }

    public override void DefinirExtras()
    {
        v.CierreCentralizado = true;
        v.DireccionAsistida = true;
    }
}
```

Definir al Director 4/4

```
public class VehiculoDirector
{
    private VehiculoBuilder builder;

    // Constructor que recibirá un Builder concreto y lo asociará al director
    public VehiculoDirector(VehiculoBuilder builder)
    {
        this.builder = builder;
    }

    public void ConstruirVehiculo()
    {
        // Construimos el vehiculo definiendo el orden del proceso
        builder.DefinirVehiculo();
        builder.ConstruirHabitaculo();
        builder.ConstruirMotor();
        builder.ConstruirRuedas();
        builder.DefinirExtras();

        // Se realizan comprobaciones y validaciones
        if ((builder.GetVehiculo().TipoCarroceria.TipoCarroceria == "deportivo")
            (builder.GetVehiculo().DireccionAsistida == false))
            throw new Exception("Error en el ensamblado: Un deportivo no puede c
    }

    public Vehiculo GetVehiculo()
    {
        return builder.GetVehiculo();
    }
}
```

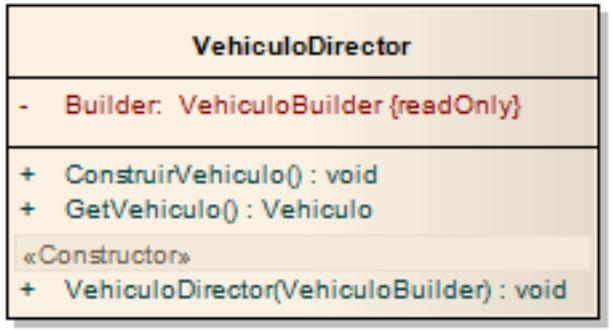
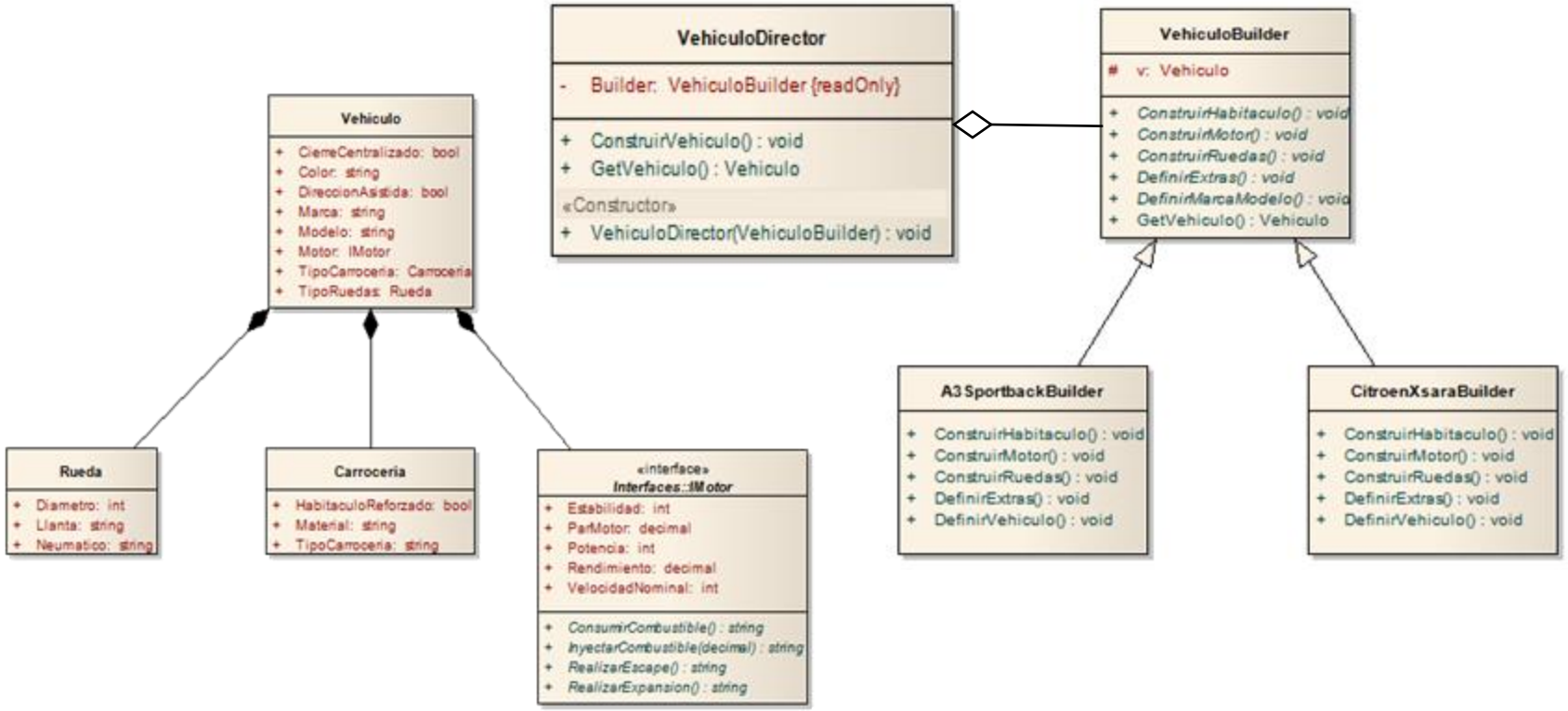


Diagrama de clases final

- A partir de nuestra clase constructora (VehiculoBuilder) creamos los constructores concretos (A3SportbackBuilder y CitroenXaraBuilder).

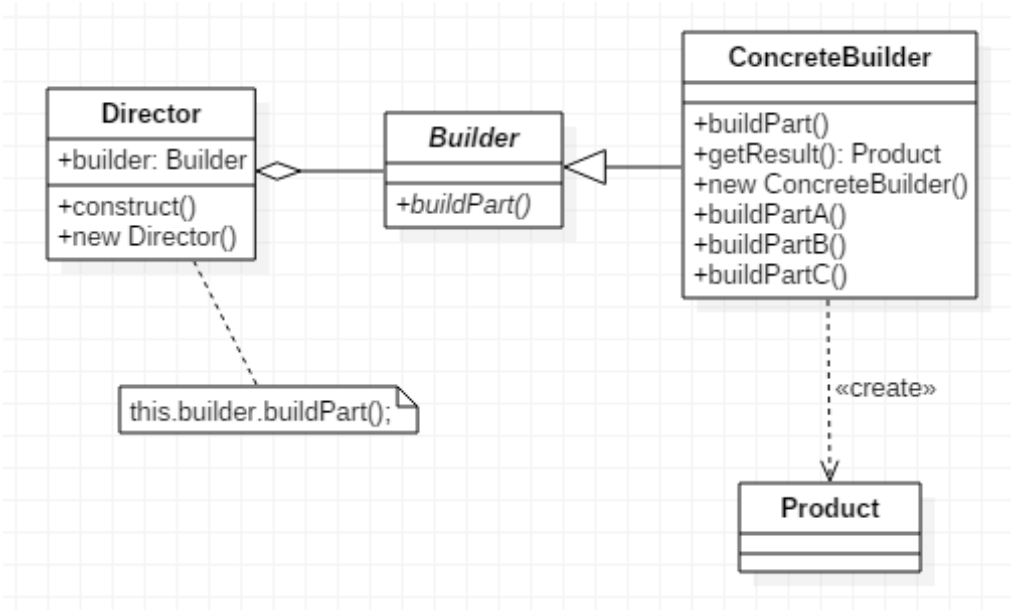


Ejecutando el código

```
// Definimos un director, pasándole un constructor de Audi como parámetro
VehiculoDirector directorAudi = new VehiculoDirector(new A3SportbackBuilder());

// El director construye el vehiculo, delegando en el constructor la tarea de
// creación de cada una de las piezas
directorAudi.ConstruirVehiculo();

// Obtenemos el vehículo directamente del director, aunque la instancia del vehículo
// se encuentra en el constructor.
Vehiculo audiA3 = directorAudi.GetVehiculo();

// Repetimos el proceso con un constructor distinto.
VehiculoDirector directorCitroen = new VehiculoDirector(new CitroenXsaraBuilder());
directorCitroen.ConstruirVehiculo();
Vehiculo citroen = directorCitroen.GetVehiculo();

// Mostramos por pantalla los dos vehiculos:
Console.WriteLine("PRIMER VEHICULO:" + Environment.NewLine);
Console.WriteLine(audiA3.GetPrestaciones());

Console.WriteLine("SEGUNDO VEHICULO:" + Environment.NewLine);
Console.WriteLine(citroen.GetPrestaciones());

Console.ReadLine();
```

Diagrama de Builder

- Director: Se encarga de iniciar el proceso.
- Builder: Clase abstracta que indica el proceso de construcción.
- ConcreteBuilder: Una clase para crear un tipo de producto específico.



Singleton

Singleton

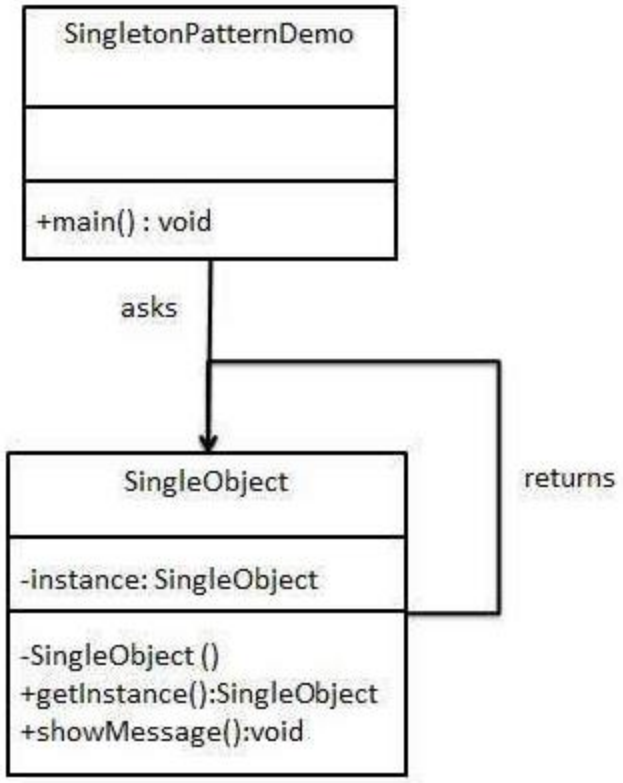
- **Propósito**
  - Asegurar que una clase sólo tenga una instancia, y proporcionar un punto de acceso global a ésta.
- **Motivación**
  - Algunas clases sólo necesitan exactamente una instancia.
    - Un sólo sistema de archivos
    - Un sólo gestor de ventanas
  - En vez de tener una variable global para acceder a ese ejemplar único, la clase se encarga de proporcionar un método de acceso

Singleton

- **Aplicación**
  - Cuando sólo puede haber un ejemplar de una clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido.
  - Cuando el único ejemplar pudiera ser extensible por herencia, y los clientes deberían usar el ejemplar de una subclase sin modificar su código.

Singleton - Implementación

Los clientes acceden al ejemplar de Singleton únicamente a través del método getInstance()



Codificando 1/2

SingleObject.java

```java
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Codificando 2/2

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Consecuencias

- Acceso controlado a una instancia única.

- Espacio de nombres reducido
  - Evita la necesidad de usar variables globales

- Permite refinar las operaciones y la representación
  - Podríamos heredar de la clase Singleton para configurar una instancia concreta.

Factory Method

Factory method

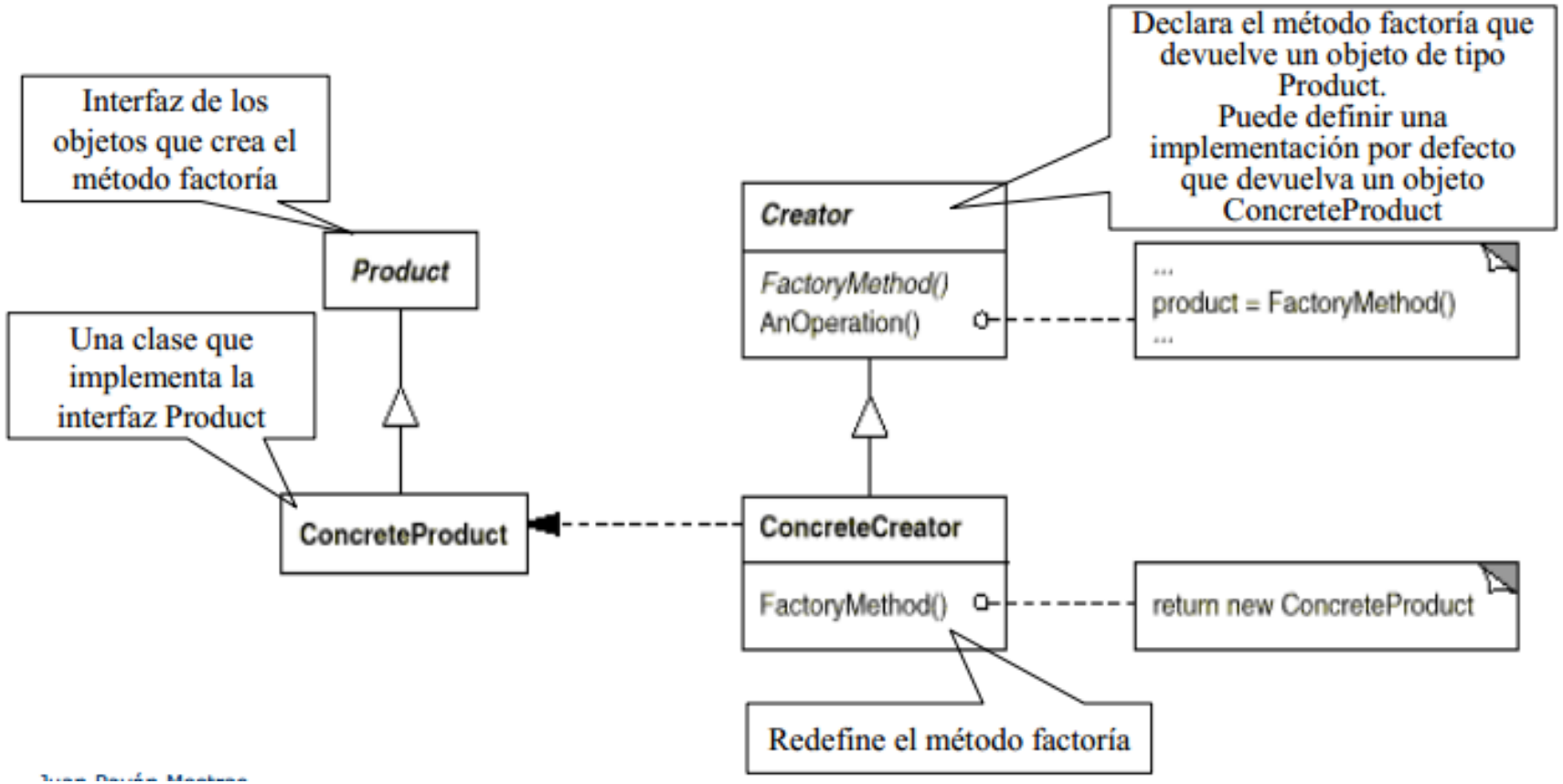
- **Propósito**
  - Permite que una clase difiera la instanciación de objetos a las subclases (son éstas las que deciden qué clase instanciar.
  - Definir un constructor virtual.

- **Ventajas**
  - Centralización de la creación de objetos
  - Facilita la escalabilidad del sistema
  - El cliente se abstrae de la instancia a crear

Factory Method

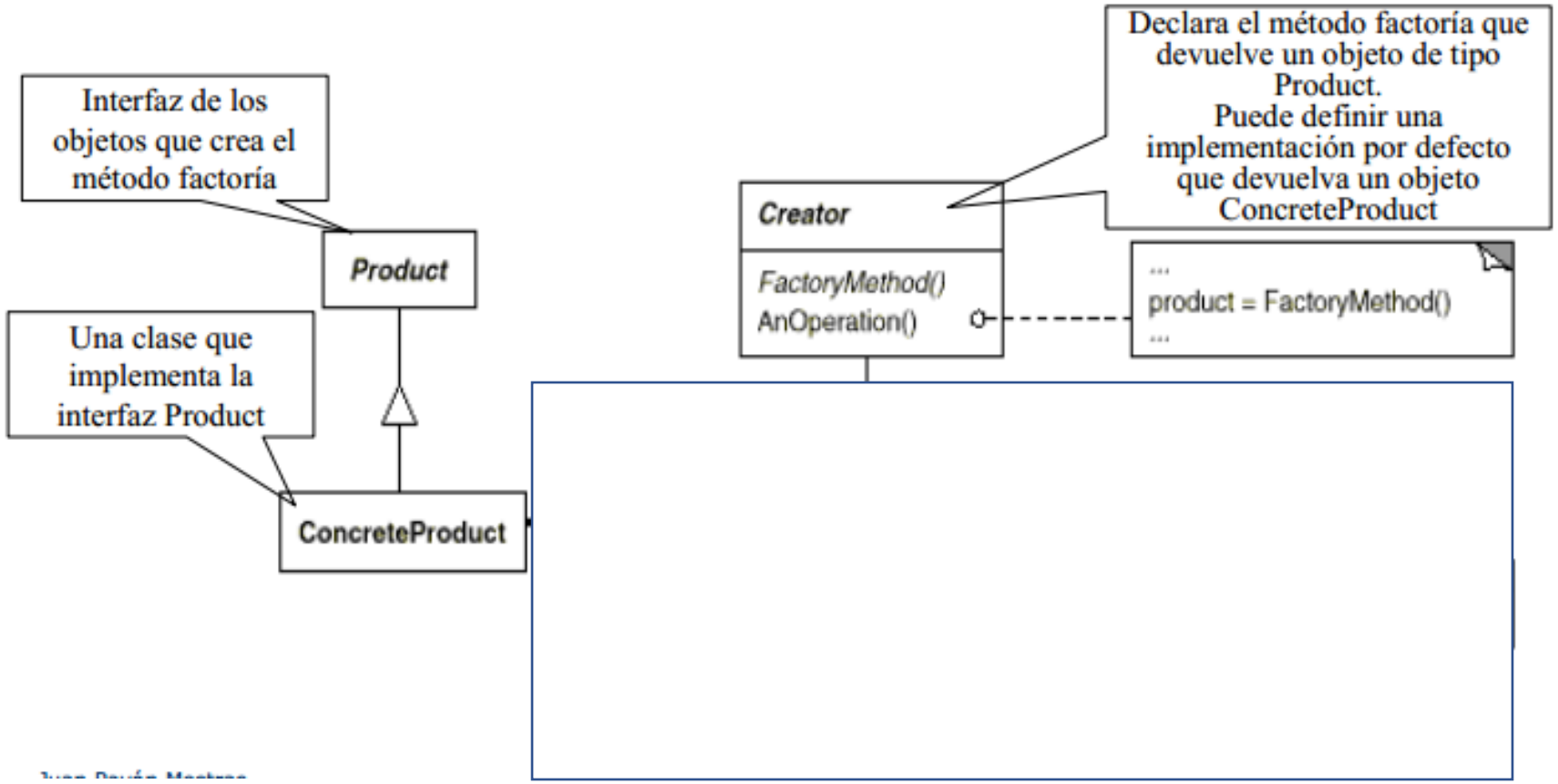
- **Aplicación:**
  - Cuando una clase no puede anticipar el tipo de objetos que debe crear.
  - Cuando una clase pretende que sus subclases especifiquen los objetos que ella crea.
  - Cuando una clase delega su responsabilidad hacia una de entre varias subclases auxiliares y queremos tener localizada a la subclase delegada.

Implementación



Implementación

- Variante: Simple Factory



Codificando 1/3

Definimos el producto y su implementación concreta:

```
public interface Product{
    public void operacion();
}

public class ConcreteProduct implements Product{
    public void operacion(){
        System.out.println("Una operación de este
producto");
    }
}
```

Codificando 2/3

```
abstract class Creator{
    // Definimos método abstracto
    public abstract Product factoryMethod();
}
```

Ahora definimos el creador concreto:

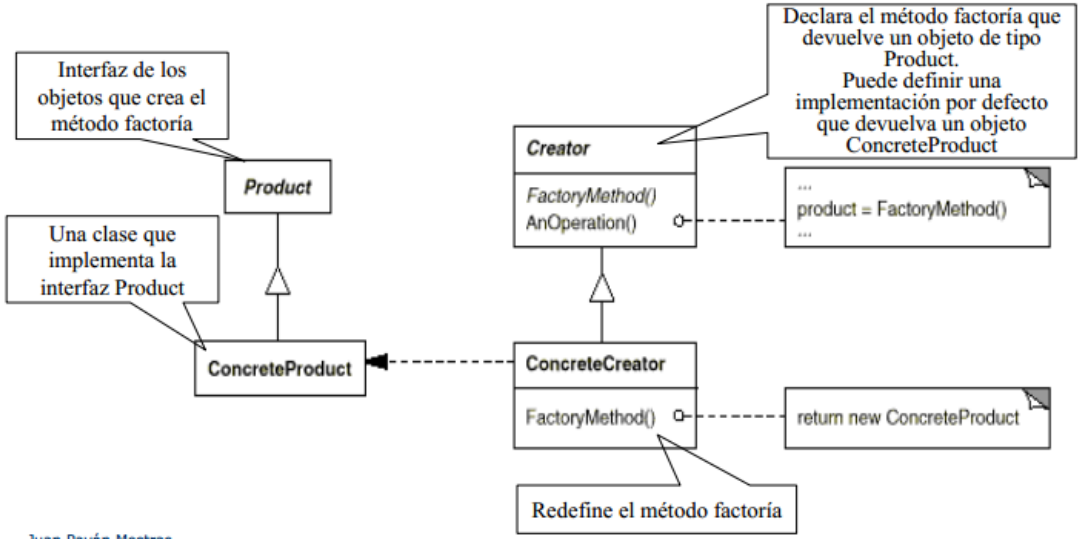
```
public class ConcreteCreator extends Creator{
    public Product factoryMethod() {
        return new ConcreteProduct();
    }
}
```

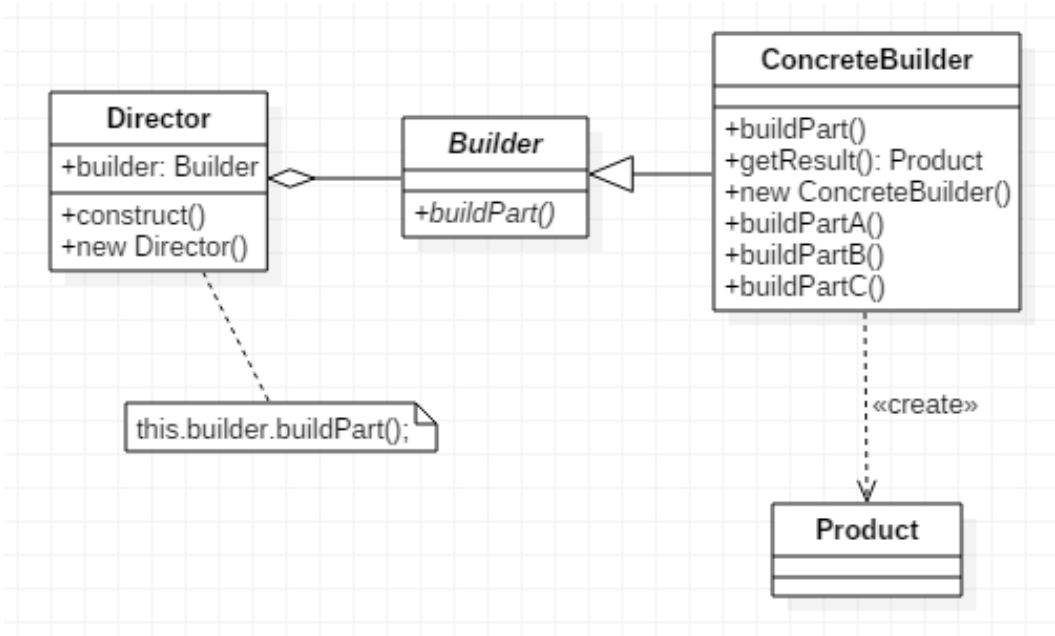
Codificando 3/3

Ejemplo de uso:

```
public static void main(String args[]){
    Creator aCreator;
    aCreator = new ConcreteCreator();
    Product producto = aCreator.factoryMethod();
    producto.operacion();
}
```

Builder vs. Factory Method





Builder vs Factory method

- **Builder** se enfoca en la construcción de objetos complejos, que se construyen paso a paso, que tienen estructura similar pero diferentes características. Generalmente se obtiene como resultado una composición de objetos
  - Ejemplo: Un auto que tiene distintas partes, una pizza con diferentes sabores y toppings.

- **Factory Method** se enfoca en la creación de una familia de objetos (diferentes pero relacionados)
  - Ejemplo: Carro, Bus, Avión son Vehículos

Builder vs Factory method

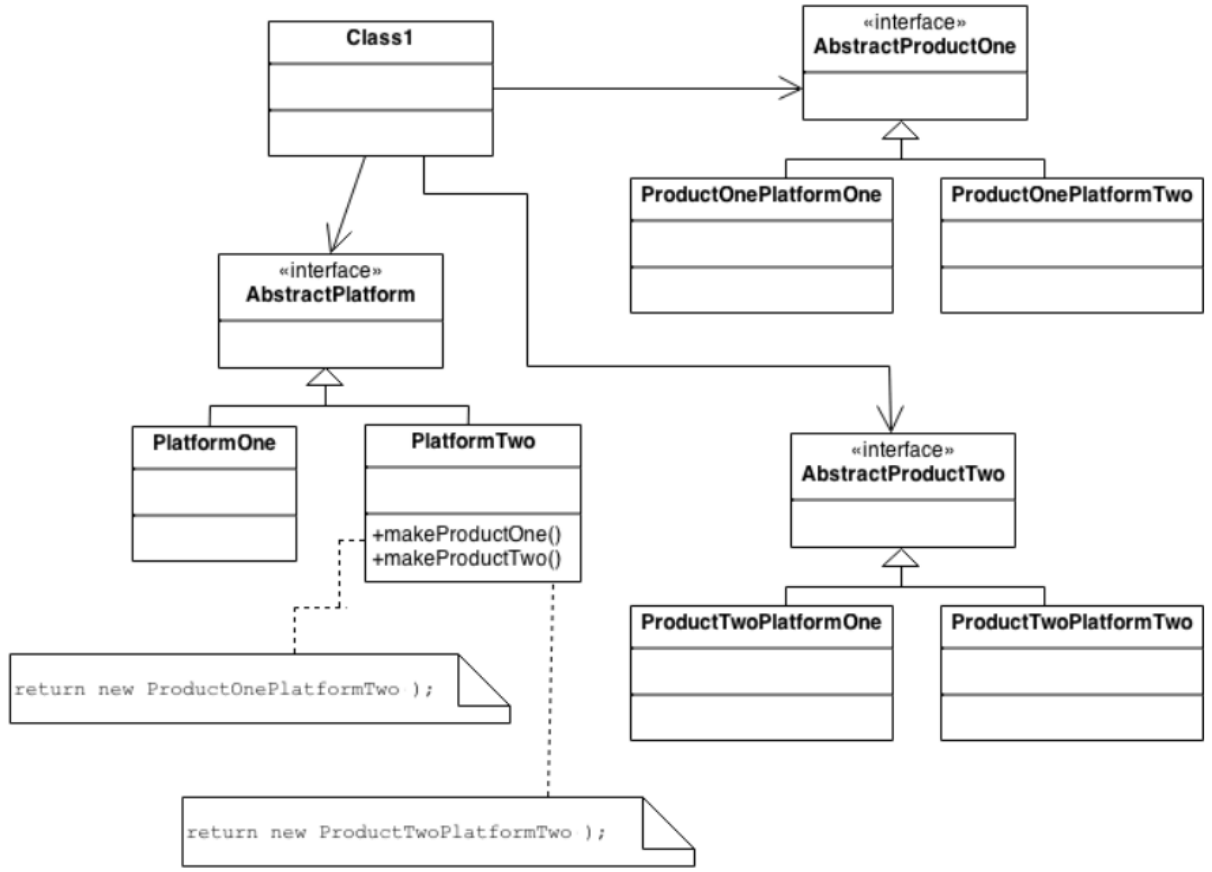
- Frecuentemente, los diseños empiezan con Factory Method y evolucionan a diseños más complejos como Builder.

- Ambos patrones pueden usarse en conjunto, dependiendo de la flexibilidad deseada

Abstract Factory

Abstract Factory

- ¿Cuál es la principal diferencia de Abstract Factory con Factory Method?

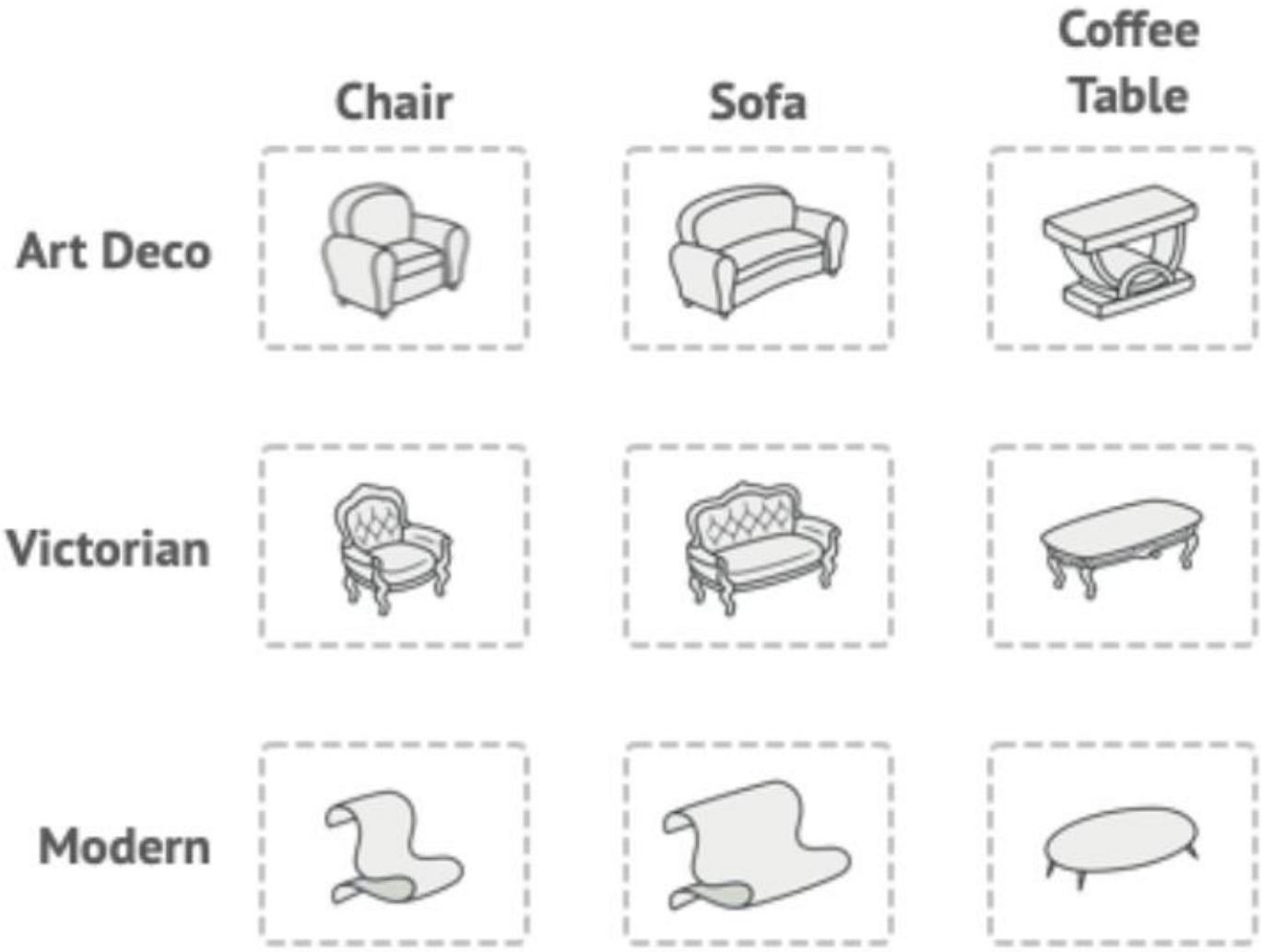


```java
public interface Currency {}

class Zloty implements Currency {}
class Euro implements Currency {}
class Dolar implements Currency {}

public class Currencies {
 public static Currency dollar() {
 return new Dollar();
 }

 public static Currency euro() {
 return new Euro();
 }

 public static Currency zloty() {
 return new Zloty();
 }
}
```

Identifique las ventajas de cada implementación

```java
public interface OSComponents {
 Window aWindow();
 Menu aMenu();
 Panel aPanel();
}

public interface Window {}
public interface Panel {}
public interface Menu {}

public class UnixComponents implements OSComponents {
 @Override
 public Window aWindow() {
 return new UnixWindow();
 }

 @Override
 public Menu aMenu() {
 return new UnixMenu();
 }

 @Override
 public Panel aPanel() {
 return new UnixPanel();
 }
}

public class WindowsComponents implements OSComponents {
 // code
}

public class iOSComponents implements OSComponents {
 // code
}
```

- Familias de objetos:
  - Chair
  - Sofa
  - Coffee Table
- Fábricas:
  - Art Deco
  - Victorian
  - Modern
- Productos Concretos:
  - ArtDecoChair
  - ArtDecoSofa
  - ArtDecoCoffeeTable
  - VictorianChair
  - VictorianSofa
  - …



Product families and their variants.

Antes de finalizar

Puntos para recordar

- ¿Qué es un patrón de diseño?
  - ¿Para qué sirve?
  - ¿Cuál es la taxonomía de patrones?
- Asocie un ejemplo que le resulte fácil de recordar para cada patrón de diseño

Lectura adicional

- Gamma et al. , “Design Patterns: Elements of Reusable Object-Oriented Software”
- Shalloway and Trott, "Design Patterns Explained"
- Source Making, “Design Patterns”
  - https://sourcemaking.com/design_patterns

- Patrones estructurales