Patrones de Diseño

Alejandro Mancilla
@alxmancilla

Agenda del curso

- Patrones ¿qué son y para qué sirven?
- Categorías de Patrones
- Clasificación de Patrones de Diseño
- Patrones de Creación - ¿Cómo creo un objeto?
- Patrones de Estructura - ¿Cómo pueden trabajar una clase y un objeto?
- Patrones de Comportamiento - ¿Cómo interactúo entre objetos?
- Patrones de Arquitectura - ¿Cómo diseño una aplicación?

Clasificación de Patrones de Diseño (GoF)

		Propósito		
		Creación	Estructura	Comportamiento
Alcance	Clase	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Patrones de Diseño de Creación - ¿Cómo creo un objeto?

- Factory Method
  Crea una instancia de varias clases derivadas
- Abstract Factory
  Crea una instancia de varias familias de clases
- Builder
  Separa la construcción de un objeto de su representación
- Prototype
  Una instancia inicializada para ser copiada o clonada
- Singleton
  Una clase de la que sólo puede existir una instancia

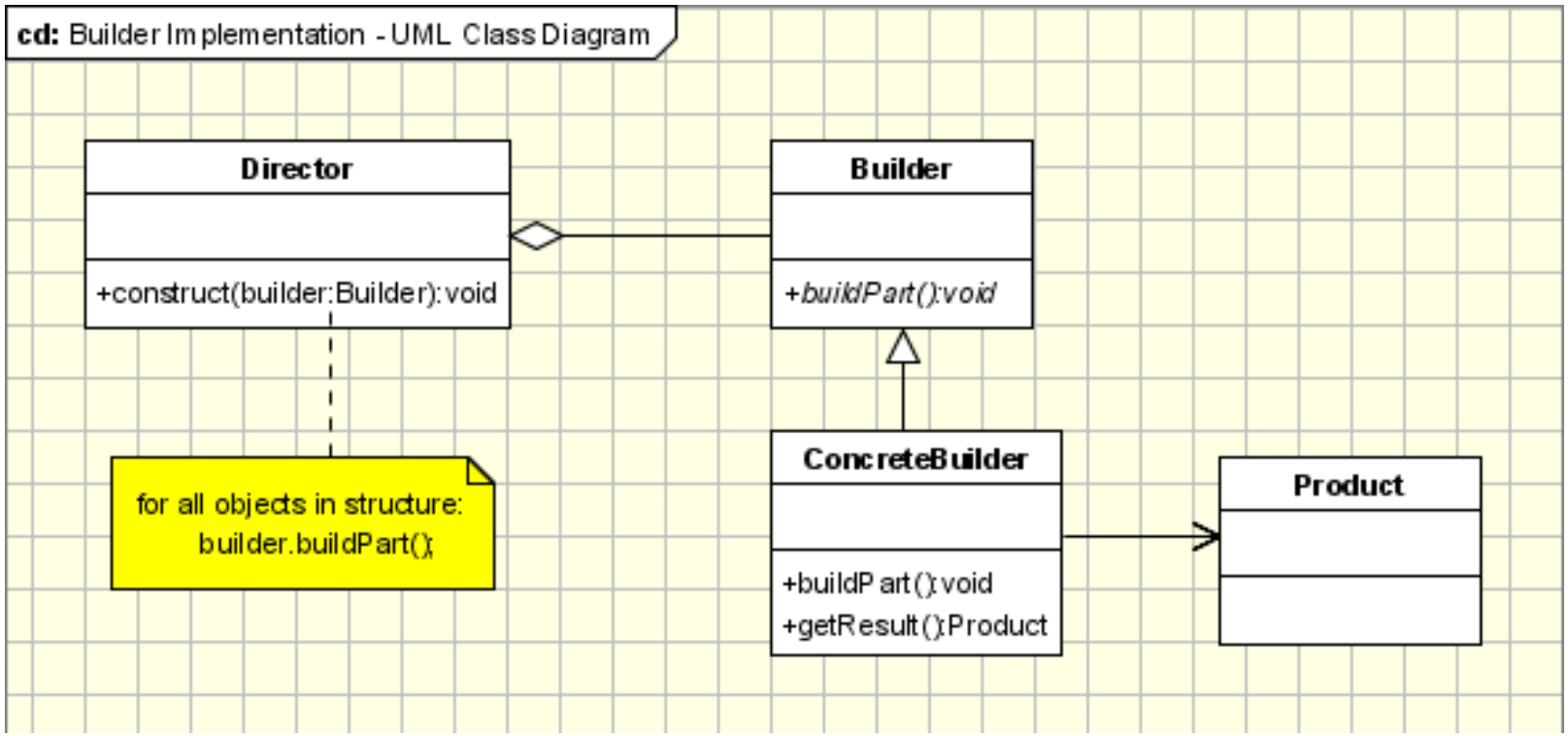
Patrón: Builder

- Problema:
  - Cuanto más complejo es una aplicación, la complejidad de las clases y los objetos que utiliza aumenta.
- Contexto:
  - Una aplicación necesitar un mecanismo para la creación de objetos complejos que es independiente de los que componen el objeto.
- Solución:
  - Define una instancia para la creación de un objeto, dejando que las subclases decidan qué clase instanciar

Participantes del patrón: Builder

- **Builder**
  - especifica una interfaz abstracta para crear partes de un objeto de Product.
- **ConcreteBuilder**
  - construye y pone juntas las partes del producto mediante la implementación de la interfaz Builder.
- **Director**
  - construye el objeto complejo mediante la interfaz Builder.
- **Product**
  - representa el objeto complejo que se está construyendo.

Participantes del patrón: Builder



Ejemplo en Javascript

```
function Director() {
        this.construct = function(builder) {
                builder.step1();
                builder.step2();
                return builder.get();
        }
}

# Abstract class
function Builder(){
        this.step1 = null;
        this.step2 = null;
        this.getResult = null;
}

function CarBuilder(){
        this.car = null;
        this.step1 = function(){
                this.car = new Car();
        }
        this.step2 = function(){
                this.car.addParts();
        }
        this.getResult = function(){
                return this.car;
        }
}
CarBuilder.prototype = Builder;
```

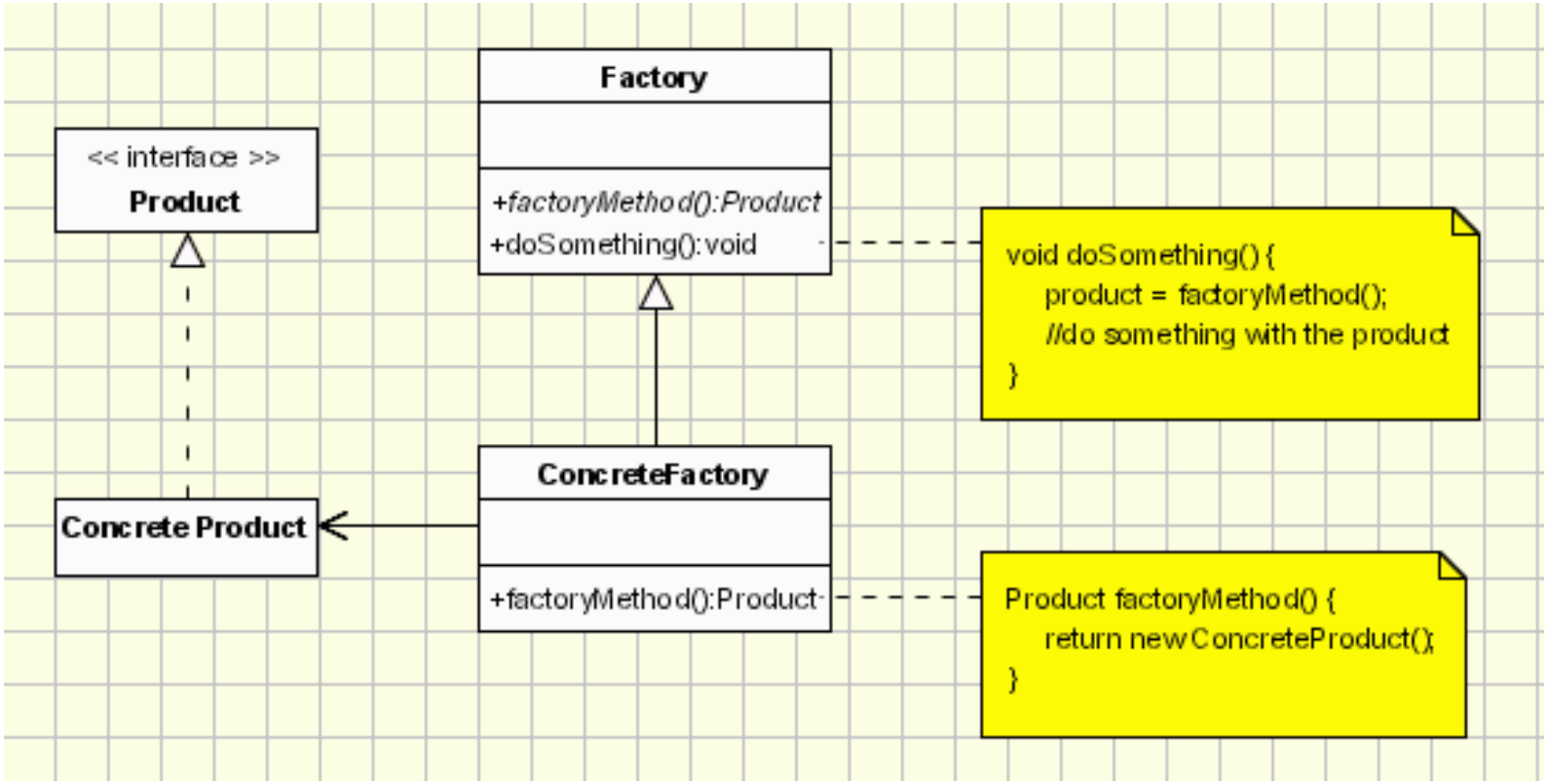
Patrón: FactoryMethod (a.k.a Virtual Constructor)

- Problema:
  - Una clase no puede anticipar la clase de objetos que debe crear.
  - Una clase quiere sus subclases especifiquen los objetos que crean.
- Contexto:
  - Los frameworks utilizan clases abstractas para definir y mantener las relaciones entre objetos. Una responsabilidad es crear tales objetos.
- Solución:
  - Definir una interfaz para crear un objeto, pero dejando la elección de su tipo a las subclases, la creación se aplaza hasta el tiempo de ejecución.

Participantes del patrón: FactoryMethod

- **Product**
  - define la interfaz para los objetos que FactoryMethod crea**.**
- **ConcreteProduct**
  - implementa la interfaz Product.
- **Creator(o Factory)**
  - declara el método FactoryMethod, que devuelve un objeto Producto. Puede llamar al método de generación para la creación de objetos Product
- **ConcreteCreator**
  - sobreescribe el método de generación para crear objetos ConcreteProduct

Participantes del patrón: Factory Method



Ejemplo del Patrón FactoryMethod

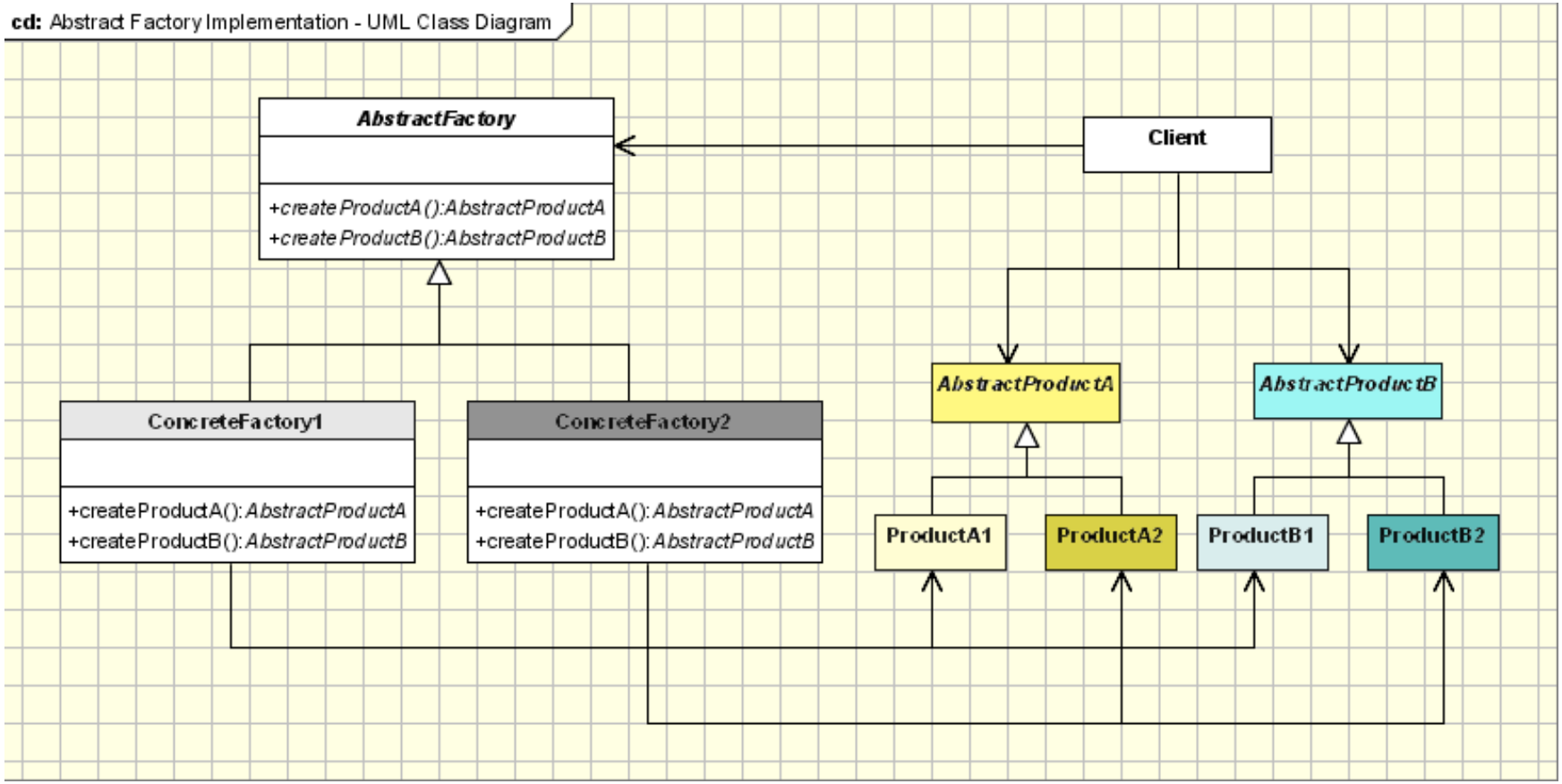
Patrón: AbstractFactory (a.k.a Kit)

- Problema:
  - Desea proporcionar una biblioteca de clases de productos
  - Desea revelar sólo sus interfaces, no sus implementaciones.
- Contexto:
  - Evitar añadir código a las clases existentes con el fin de hacer que encapsule información más general.
- Solución:
  - Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

Participantes del patrón: AbstractFactory

- **AbstractFactory**
  - declara una interfaz para las operaciones que crean **AbstractProduct**
- **ConcreteFactory**
  - implementa operaciones para crear **Product** concretos.
- **AbstractProduct**
  - declara una interfaz para un tipo de objetos **Product**.
- **Product**
  - define un producto a ser creado por el ConcreteFactory correspondiente; que implementa la interfaz AbstractProduct.
- **Client**
  - utiliza las interfaces declaradas por las clases **AbstractFactory** y **AbstractProduct**.

Participantes del patrón: AbstractFactory



Ejemplo del Patrón AbstractFactory

Algunas notas sobre Patrones de Creación

- Abstract Factory tiene el objeto factory produciendo objetos de varias clases.
- Builder tiene el objeto factory construyendo un producto incrementalmente usando una estructura compleja.
- Las clases AbstractFactory a menudo son implementadas usando FactoryMethods pero también pueden ser implementadas usando Prototype.
- Builder se enfoca en construir objetos complejos paso a paso.
- Abstract Factory hace énfasis en una familia de objetos de producto (ya sean simples o complejos).
- Builder regresa el producto como un paso final, pero con en cuanto a Abstract Factory, el producto se regresa inmediatamente.

Patrones de Estructura
- ¿Cómo pueden trabajar una clase y un objeto?

- Adapter
  Relaciona interfaces de diferentes clases
- Bridge
  Separa la interfaz de un objeto de su implementación
- Composite
  Una estructura de árbol de objetos simples y compuestos
- Decorator
  Añadir responsabilidades a los objetos dinámicamente
- Façade
  Una única clase que representa todo un subsistema
- Flyweight
  Una instancia usada para compartición eficiente
- Proxy
  Un objeto que representa a otro objeto

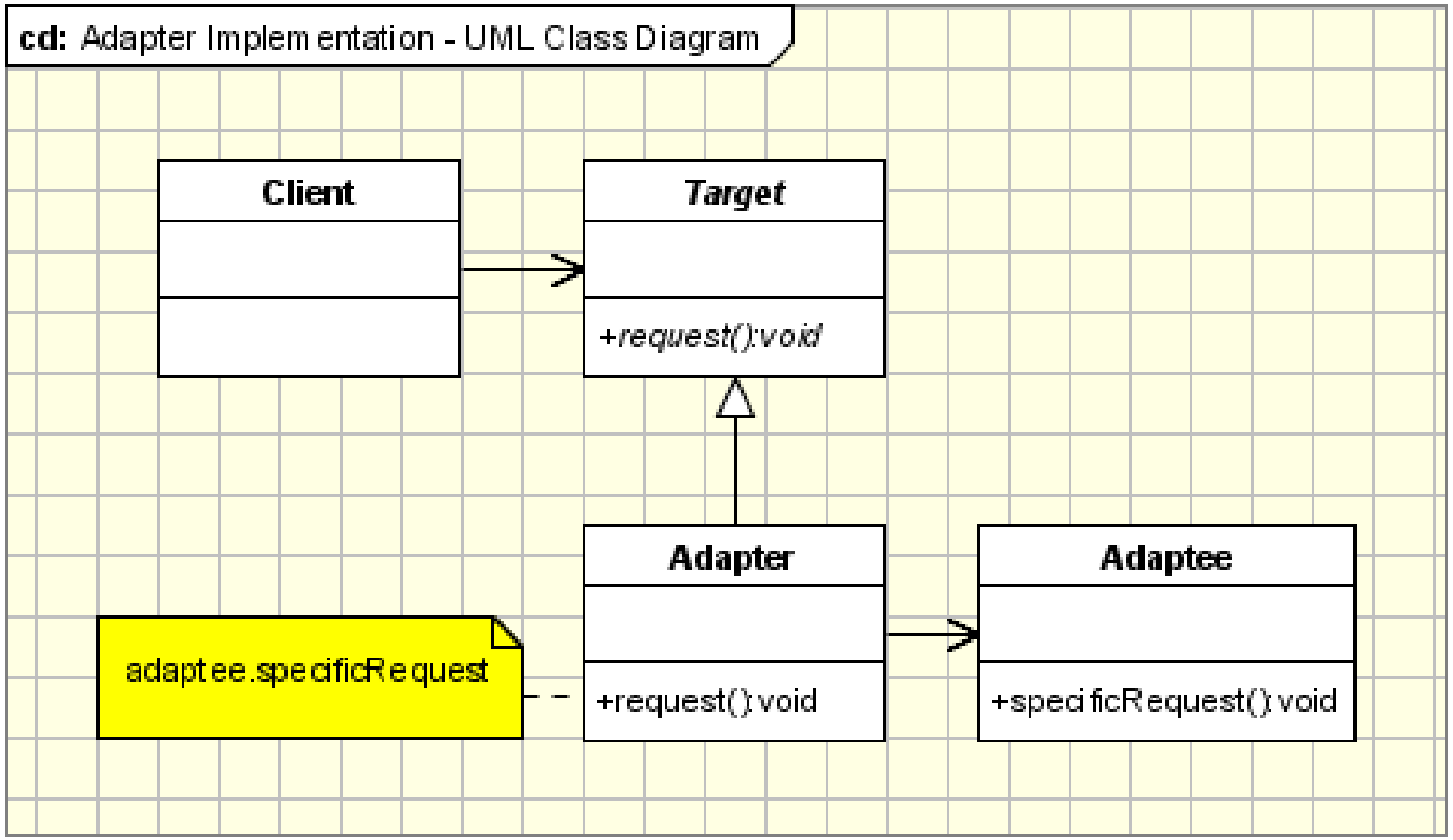
Patrón: Adapter(a.k.a Wrapper)

- Problema:
  - Desea utilizar una clase existente, y su interfaz no coincide con la que necesita
  - Desea crear clases reutilizables que cooperen con clases sin relación o imprevistas
- Contexto:
  - Relacionar dos componentes que no tienen una interfaz común
- Solución:
  - Convertir la interfaz de una clase en otra interfaz que el clientes espera.

Participantes del patrón: Adapter

- **Target**
  - define la interfaz de dominio específico que utiliza **Client**.
- **Adapter**
  - adapta la interfaz **Adaptee** para la interfaz de destino.
- **Adaptee**
  - define una interfaz existente que necesita adaptarse.
- **Client**
  - colabora con objetos de acuerdo con la interfaz **Target**.

Participantes del patrón: Adapter



Ejemplo del Patrón Adapter

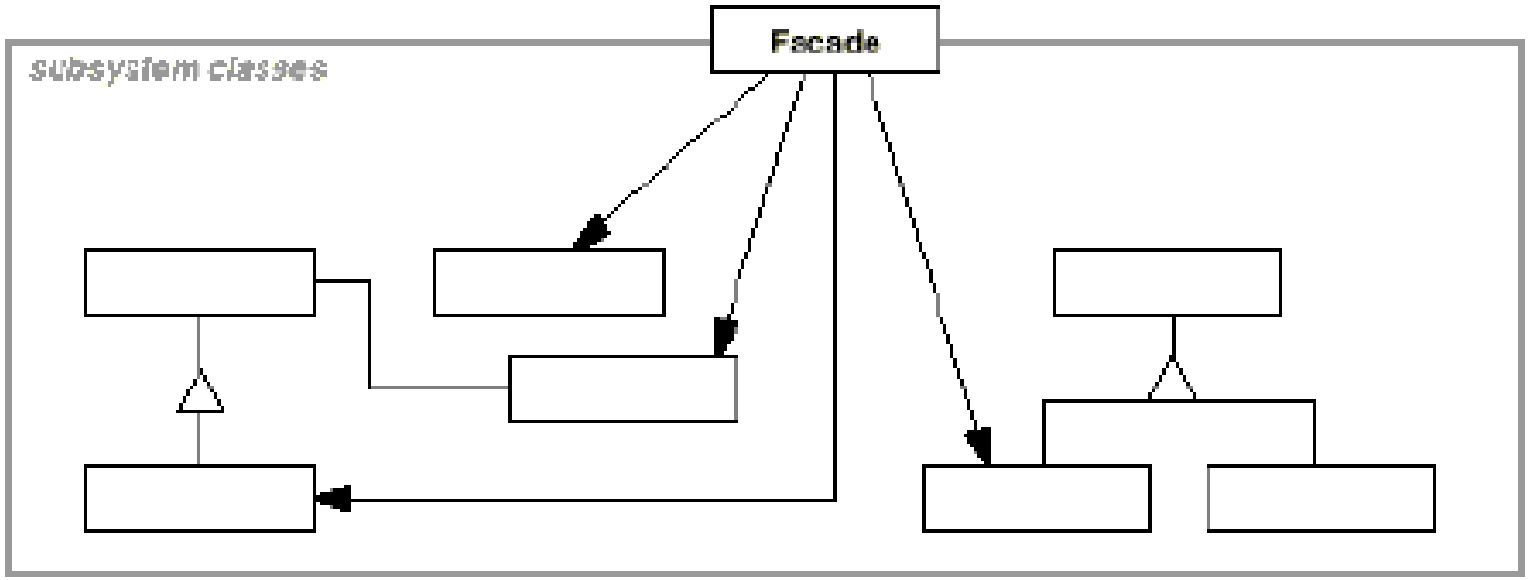
Patrón: Façade

- Problema:
  - Proveer una interfaz simple a un subsistema complejo
- Contexto:
  - Minimizar la comunicación y dependencias entre subsistemas
- Solución
  - Proporcionar una interfaz unificada para un conjunto de interfaces de un subsistema.

Participantes del patrón: Façade

- **Façade**
  - Conoce cuáles clases del subsistema son responsables por una petición.
  - Delegan peticiones del cliente a los objetos del subsistema apropiado
- **clases del subsistema**
  - Implementan una funcionalidad del subsistema
  - Llevan a cabo el trabajo asignado por el objeto **Façade**

Participantes del patrón: Façade



Ejemplo del Patrón Façade