

Design Patterns

Grupo #5

Factory

```
1  abstract class Creator {
2      public abstract Product factoryMethod();
3  }
4
5  class ConcreteCreator extends Creator {
6      @Override
7      public Product factoryMethod() {
8          return new ConcreteProduct();
9      }
10 }
11
12 interface Product {}
13
14 class ConcreteProduct implements Product {}
15
```

- **Intención:** Proporcionar una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que serán creados.
- Conocido como: Factory Method, Método Fábrica.
- **Motivo:** Permite a una clase delegar en sus subclases la creación de objetos, lo que facilita la extensión de la funcionalidad de la aplicación modificando únicamente las subclases involucradas.

Adapter-(Wrapper)

- **Intención:** Convertir la interfaz de una clase en otra interfaz que esperan los clientes. Adapter permite que clases con interfaces incompatibles trabajen juntas.
- **Motivo:** Permite la colaboración entre clases que de otro modo no podrían trabajar juntas debido a sus interfaces incompatibles.
- **Aplicaciones:** Cuando se quiere utilizar una clase existente, y su interfaz no se corresponde con la que necesitamos.

```
1 // Interfaz objetivo
2 interface Target {
3     void request();
4 }
5
6 // Clase a adaptar
7 class Adaptee {
8     void specificRequest() {
9         // Implementación específica
10    }
11 }
12
13 // Adaptador
14 class Adapter implements Target {
15     private Adaptee adaptee = new Adaptee();
16
17     @Override
18     public void request() {
19         adaptee.specificRequest();
20     }
21 }
22
```

Strategy-(Policy)

- **Intención:** Definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Strategy permite que el algoritmo varíe independientemente de los clientes que lo utilizan.
- **Motivo:** Permite seleccionar el algoritmo de comportamiento en tiempo de ejecución, proporcionando una alternativa flexible a la herencia para seleccionar comportamientos.
- **Aplicaciones:** Cuando hay varios algoritmos para realizar una tarea y se desea seleccionar dinámicamente cuál utilizar.

```
1 interface Strategy {
2     void algorithmInterface();
3 }
4
5 class ConcreteStrategyA implements Strategy {
6     public void algorithmInterface() {
7         // Implementación del algoritmo
8     }
9 }
10
11 class Context {
12     private Strategy strategy;
13
14     // Método para cambiar la estrategia en tiempo de ejecución
15     public void setStrategy(Strategy strategy) {
16         this.strategy = strategy;
17     }
18
19     public void executeStrategy() {
20         strategy.algorithmInterface();
21     }
22 }
23
```