

Técnicas de Programación Avanzada (Java)

Curso 2021-2021

S_3 (Patrones de diseño) {

3.1 Concepto de Patrones de Diseño. (+SOLID)

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

}

Tema 1: Objetos y memoria.

1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

1.2 Sentencias de control. Secuencia, selección e iteración.

1.3 Abstracción. Clases, objetos, métodos y atributos.

1.4 Sobrecarga de métodos y encapsulamiento.

Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos

2.1 Herencia. Interfaces y clases abstractas. Agregación.

2.2 Polimorfismo.

2.3 Gestión de Excepciones.

2.4 Genericidad y plantillas.

2.5 Utilidades. Entrada y Salida.

2.6 Anotaciones.

Tema 3. Patrones de Diseño.

3.1 Concepto de Patrones de Diseño.

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

Tema 4. Programación de Interfaces.

4.1 Interfaces Gráficas de Usuario.

4.2 Gestión de eventos.

Tema 5. Temas Avanzados.

5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

Attributions



Diapositivas redactadas con
texto e imágenes de:



V2021-1.7. Alexander Shvets.

<https://refactoring.guru/es/design-patterns/book>



3.0 Malas prácticas de programación - Síntomas



- **Fragilidad:** al realizar un simple cambio, se rompe el código de otras muchas partes, incluso aunque no está relacionado en modo alguno. Incrementa el coste de cambiar el código, porque se pueden generar muchos problemas inesperados.
- **Rigidez:** código difícil de cambiar, o extender. Requiere la creación de otros sistemas externos.
- **Inmovilidad:** falta de flexibilidad, es difícil separar el Sistema en componentes para reusar el código en otras partes. Lleva a la duplicidad de Código.
- **Viscosidad:** los problemas presentes en el código llevan a seguir empleando métodos malos, pues es más conveniente y fácil que aplicar buenas estructuras. Ciclo vicioso. Parches.



Code smells




| Family | Name | Description of the family |
|-----------------------------------|--|---|
| Bloaters | Long method, Large class, Primitive obsession, long parameter list, data clumps | Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually, these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them). |
| Object-Orientation Abusers | Alternative Classes with Different Interfaces, Refused Bequest, Switch Statements, Temporary Field | All these smells are incomplete or incorrect application of object-oriented programming principles. |
| Change preventers | Divergent Change, Parallel Inheritance Hierarchies, Shotgun Surgery | These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result. |
| Dispensables | Comments, Duplicate Code, Data Class, Dead Code, Lazy Class, Speculative Generality | A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand. |
| Couplers | Feature Envy, Inappropriate Intimacy, Incomplete Library Class, Message Chains, Middle Man | All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation. |

3.0 Buenas prácticas de programación

- **Abstracción:** La abstracción es un proceso de interpretación y diseño que implica reconocer y enfocarse en las características importantes de una situación u objeto, y filtrar o ignorar todas las particularidades no esenciales. Dejar a un lado los detalles de un objeto y definir las características específicas de éste, aquellas que lo distinguen de los demás tipos de objetos. Hay que centrarse en lo que es y lo que hace un objeto, antes de decidir cómo debería ser implementado. (¿Qué?)
- **Encapsulación:** Es la propiedad que permite asegurar que la información de un objeto está oculta del mundo exterior. Consiste en agrupar en una Clase las características (atributos) con un acceso privado y los comportamientos (métodos) con un acceso público → Acceder o modificar los miembros de una clase a través de sus métodos.
- **Modularidad:** La modularidad es la propiedad que permite dividir una aplicación en partes más pequeñas, cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- **Herencia/Generalización:** permite la reusabilidad de Código extendiendo clases y añadiendo nuevos métodos o atributos. Evita la repetición de código similar (redundancia).

BUENAS PRACTICAS EN PROGRAMACIÓN

- **Diseño modular, encapsulación a nivel de método:**
 - Código reutilizable (modular, con clases sencillas)
 - Extensible (mejor con clases abstractas e interfaces)
 - Núcleo del código protegido (**abstracción y encapsulamiento**) (*Abstraction & Encapsulation*)
- **Programar a una interfaz no a una implementación**
 - En lugar de hacer dependencias entre clases, introducir **interfaz** intermedia (*Generalization - Interface*)
- **Favorecer la composición frente a la herencia (*Decomposition*)**
 - Asociación (baja dependencia)
 - Agregación
 - Composición (alta dependencia)  → Generalización
- **Principios SOLID**



PATRONES DE DISEÑO

Principios SOLID

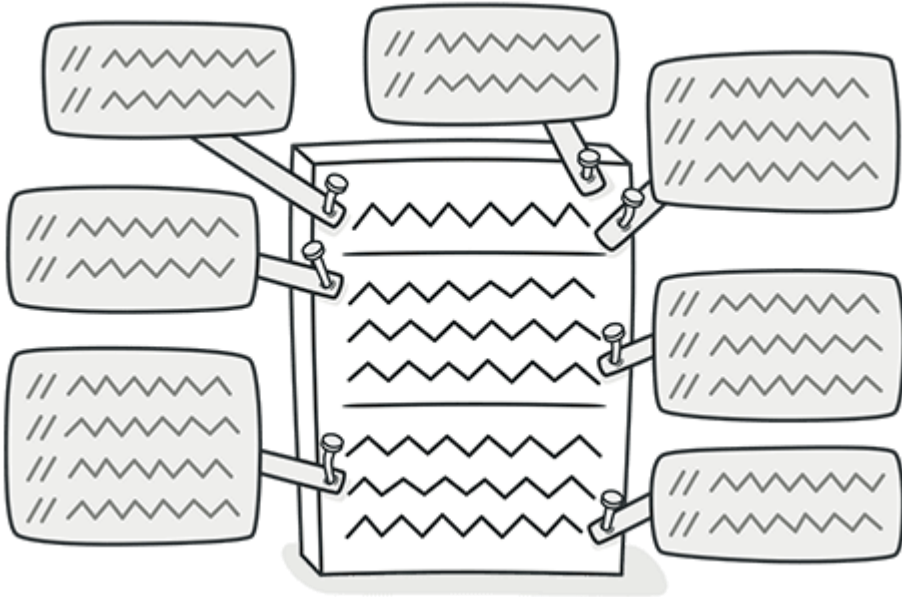
| Inicial | Acrónimo | Concepto |
|---------|----------|---|
| S | SRP | Principio de responsabilidad única (<i>Single responsibility principle</i>) la noción de que un objeto solo debería tener una única responsabilidad. |
| O | OCP | Principio de abierto/cerrado (<i>Open/closed principle</i>) la noción de que las “entidades de software ... deben estar abiertas para su extensión, pero cerradas para su modificación”. |
| L | LSP | Principio de sustitución de Liskov (<i>Liskov substitution principle</i>) la noción de que los “objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa”. Véase también diseño por contrato . |
| I | ISP | Principio de segregación de la interfaz (<i>Interface segregation principle</i>) la noción de que “muchas interfaces cliente específicas son mejores que una interfaz de propósito general”. ⁴ |
| D | DIP | Principio de inversión de la dependencia (<i>Dependency inversion principle</i>) la noción de que se debe “depender de abstracciones, no depender de implementaciones”. ⁴ La Inyección de Dependencias es uno de los métodos/patrones que siguen este principio |

<https://es.wikipedia.org/wiki/SOLID>

Cosas a evitar y recomendaciones

| to DO – Keep it simple and easy | NOT to do |
|---|---|
| - Lanzar programa desde una clase creada específicamente para ello | - Tener todo mi programa en una misma clase |
| - Llamar a métodos desde dentro de los switch | - Poner dentro de los switch muchas líneas de código |
| - Separar acciones en métodos | - Tener métodos que realizan diferentes funciones |
| - Usar polimorfismo en vez de condicionales en las clases | - Heredar de una clase y repetir el mismo código en las clases hijas |
| - Usar comentarios cuando sea necesario para explicar por qué se implementa así el código o similar | - Abusar de comentarios y emplearlos como sustitutos de funciones/métodos |
| - Si usamos Código de otra persona, debemos dar atribuciones y referenciarlo. | - Usar variables con nombres genéricos, letras, |
| - Seguir las convenciones para nombres de clases, paquetes, módulos, métodos, variables, constantes... (mayúsculas/minúsculas, camelCase, snake_case) | - Mezclar estilos. Check: https://en.wikipedia.org/wiki/Naming_convention_(programming) |

Comentarios: no abusar



- No deben sustituir a una funcionalidad. A veces se pueden sustituir por un método
- Si el código es claro y el nombre de las variables y funciones está bien elegido, no son necesarios (GOAL)
- Si el Código cambia, hay que actualizar los comentarios relacionados. Un exceso de ellos aumenta el tiempo requerido para esta tarea

@annotations & JavaDocs NO SON COMENTARIOS!
USAR SIEMPRE



```
void printProperties(List users) {  
    for (int i = 0; i < users.size(); i++) {  
        String result = "";  
        result += users.get(i).getName();  
        result += " ";  
        result += users.get(i).getAge();  
        System.out.println(result);  
  
        // ...  
    }  
}
```



```
void printProperties(List users) {  
    for (User user : users) {  
        System.out.println(getProperties(user));  
  
        // ...  
    }  
}  
  
String getProperties(User user) {  
    return user.getName() + " " + user.getAge();  
}
```

3.1 Concepto: Patrones de Diseño

3.1 Concepto de Patrones de Diseño

Introducción

Los **patrones de diseño** son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. No son plantillas, son consejos.

- Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
- Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
- Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

3.1 Concepto de Patrones de Diseño.

Tipos

| Familias | Se dedican a... | Ejemplos de patrones |
|-------------------|--|---|
| Creacionales | Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente. | <ul style="list-style-type: none">- Factory Method- Abstract Factory- Builder- Prototype- Singleton |
| Estructurales | Explican cómo ensamblar objetos y clases en estructuras más grandes, mientras se mantiene la flexibilidad y eficiencia de la estructura. | <ul style="list-style-type: none">- Adapter- Bridge- Composite- Decorator- Facade- Flyweight- Proxy |
| De comportamiento | Tratan con algoritmos y la asignación de responsabilidades entre objetos. | <ul style="list-style-type: none">- Chain of responsibility / - Command- Iterator / - Mediator- Memento / - Observer- State / - Strategy- Template method / - Visitor |

Inyección de dependencias

Es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos.

```
public class Vehiculo {  
  
    private Motor motor = new Motor();  
  
    public Double enAceleracionDePedal(int presionDePedal) {  
        motor.setPresionDePedal(presionDePedal);  
        int torque = motor.getTorque();  
        Double velocidad = ...  
        return velocidad;  
    }  
}
```

```
public class Vehiculo {  
  
    private Motor motor = null;  
  
    public void setMotor(Motor motor){  
        this.motor = motor;  
    }  
  
    public Double enAceleracionDePedal(int presionDePedal) {  
        Double velocidad = null;  
        if (null != motor){  
            motor.setPresionDePedal(presionDePedal);  
            int torque = motor.getTorque();  
            velocidad = ... //realiza el cálculo  
        }  
        return velocidad;  
    }  
}
```

```
public class VehiculoFactory {  
  
    public Vehiculo construyeVehiculo() {  
        Vehiculo vehiculo = new Vehiculo();  
        Motor motor = new Motor();  
        vehiculo.setMotor(motor);  
        return vehiculo;  
    }  
}
```

3.2 Patrones Creacionales

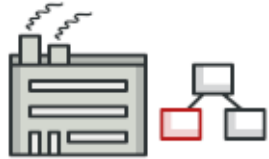
3.1 Concepto de Patrones de Diseño.

Patrones Creacionales

| Patrón | Finalidad | |
|------------------|--|--|
| Factory Method | Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán. | |
| Abstract Factory | Nos permite producir familias de objetos relacionados sin especificar sus clases concretas. | |
| Builder | Nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción. | |
| Prototype | Nos permite copiar objetos existentes sin que el código dependa de sus clases | |
| Singleton | Nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia. | |

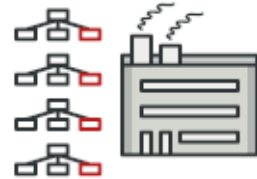
3.2 Patrones de creación

Introducción



Factory Method

Proporciona una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



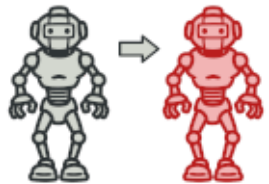
Abstract Factory

Permite producir familias de objetos relacionados sin especificar sus clases concretas.



Builder

Permite construir objetos complejos paso a paso. Este patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.



Prototype

Permite copiar objetos existentes sin que el código dependa de sus clases.

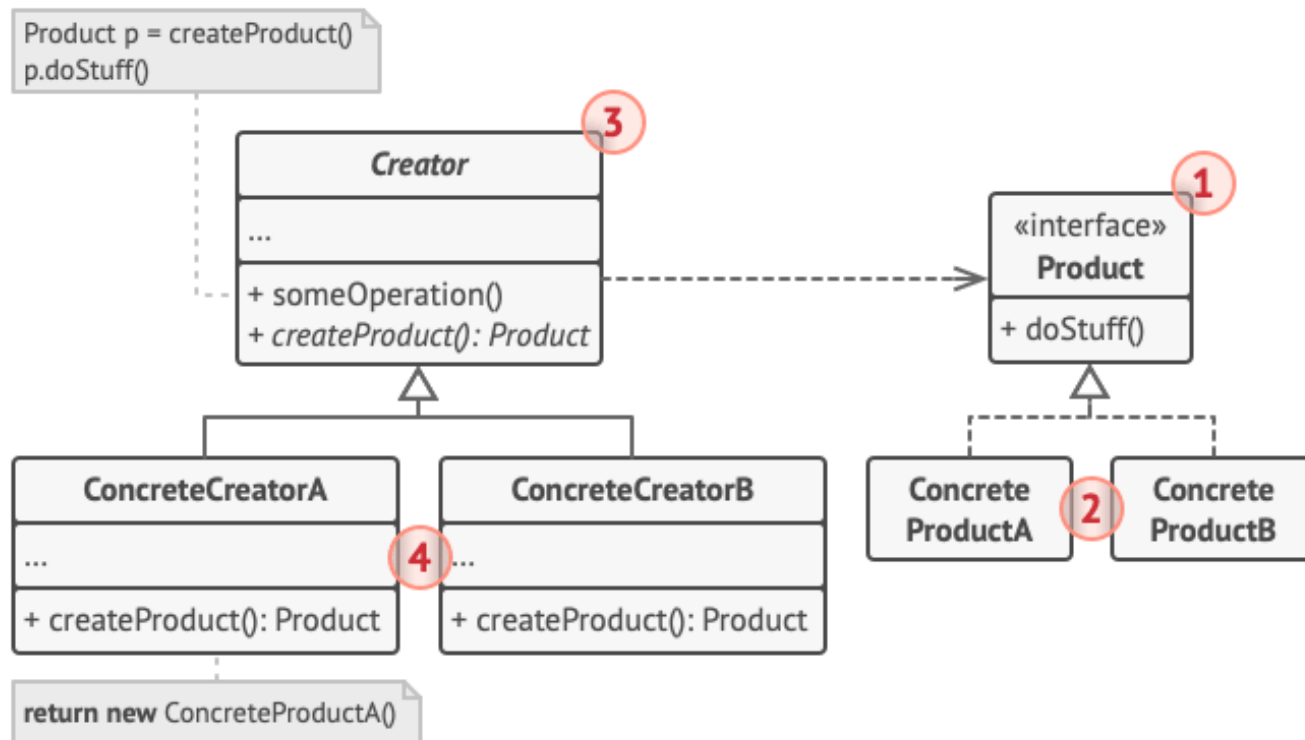


Singleton

Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

3.2 Patrones Creacionales

FACTORY METHOD



En lugar de llamar al operador **new** para construir objetos directamente, se invoque a un método *fábrica* especial.

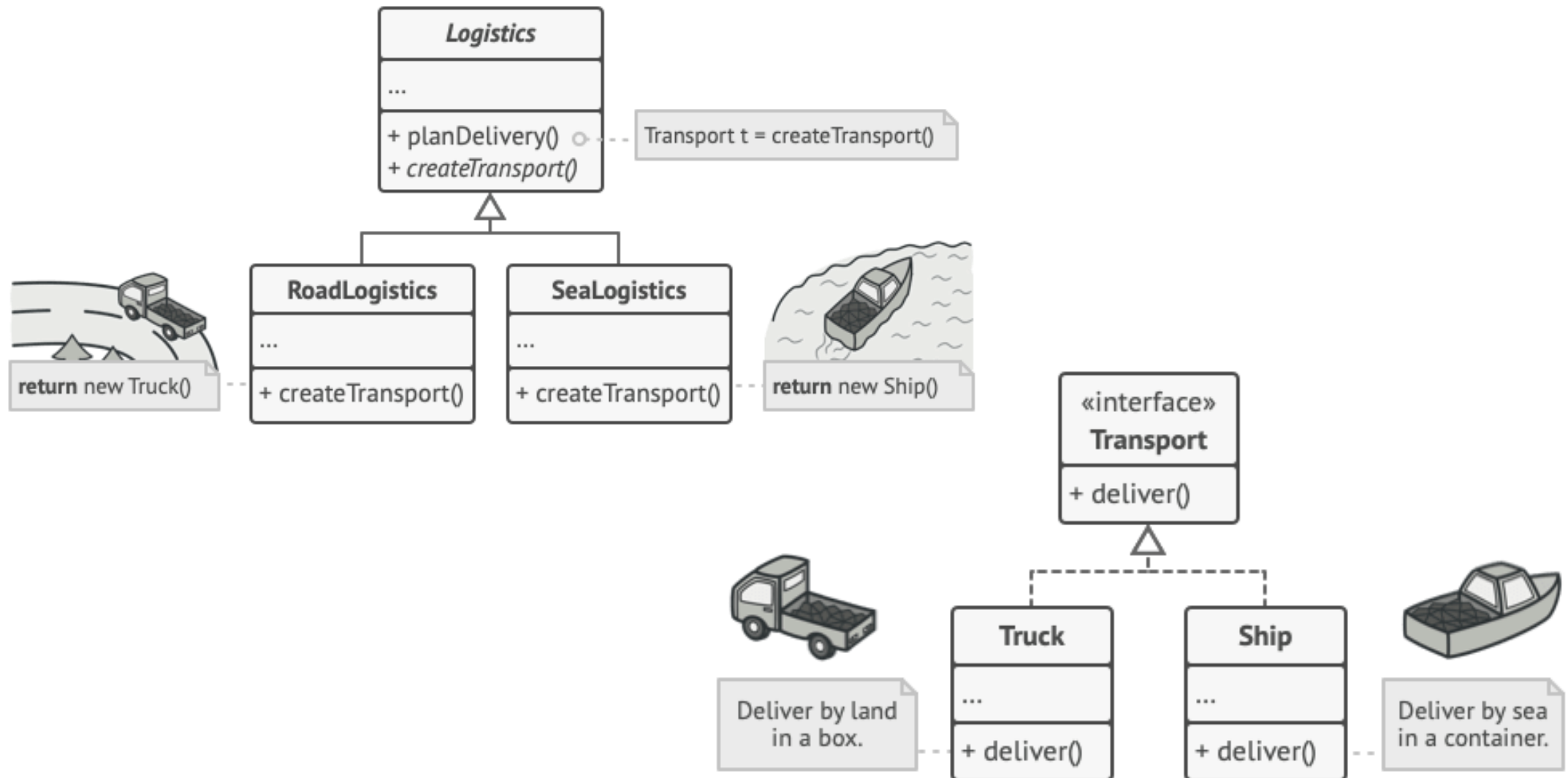
Ahora se puede sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

Las subclases sólo pueden devolver productos de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.

Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

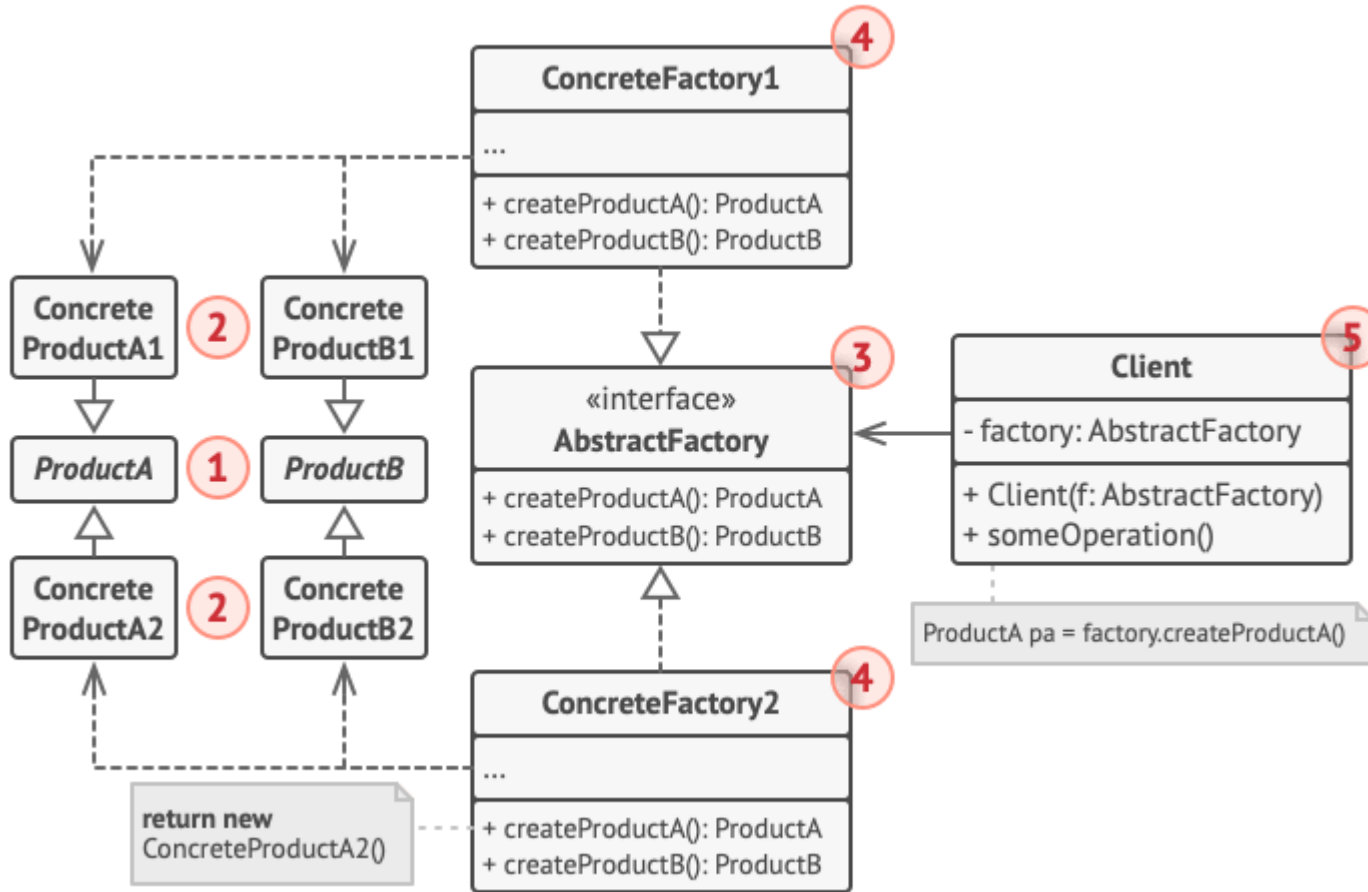
3.2 Patrones Creacionales

FACTORY METHOD



3.2 Patrones Creacionales

ABSTRACT METHOD



Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

Declaramos de forma explícita interfaces para cada producto diferente de la familia de productos

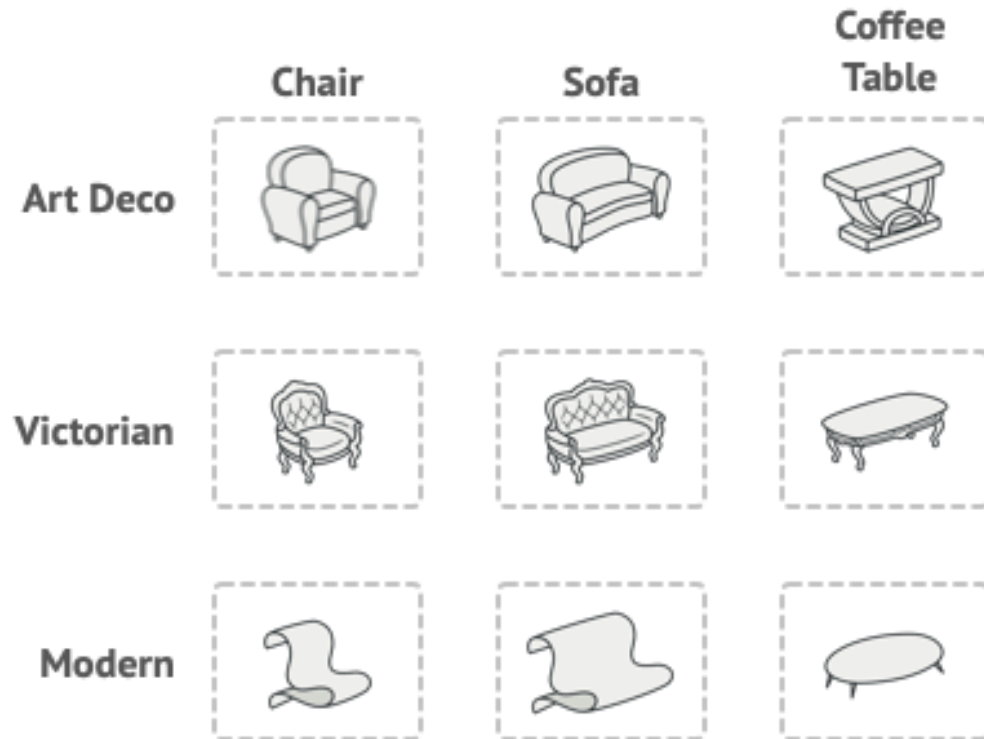
Después podemos hacer que todas las variantes de los productos sigan esas interfaces.

El siguiente paso consiste en declarar la *Fábrica abstracta*: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos. Estos métodos deben devolver productos **abstractos** representados por las interfaces que extrajimos previamente.

Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz **FábricaAbstracta**

3.2 Patrones Creacionales

ABSTRACT METHOD

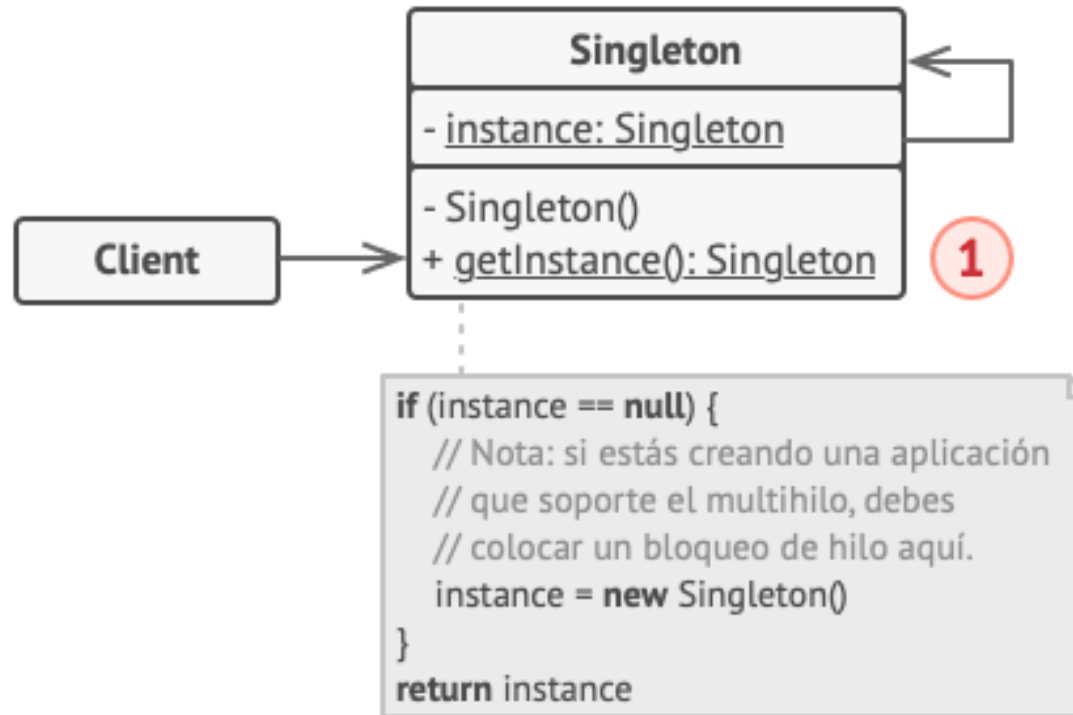


Ejemplo:

- JComponents para diferentes sistemas operativos (diseño)

3.2 Patrones Creacionales

SINGLETON



Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.

Crear un método de creación estático que actúe como constructor(este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático). Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si el código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

3.3 Patrones Estructurales

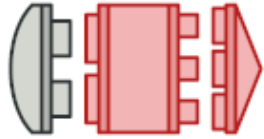
3.1 Concepto de Patrones de Diseño.

Patrones Creacionales

| Patrón | Finalidad |
|------------------|--|
| Adapter | Permite la colaboración entre objetos con interfaces incompatibles. |
| Bridge | Permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra. |
| Composite | Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales. |
| Decorator | Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades. |
| Facade | Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases. |
| Flyweight | Permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto. |
| Proxy | Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original. |

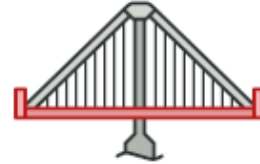
3.3 Patrones estructurales

Introducción



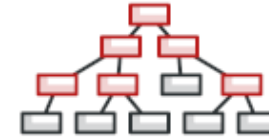
Adapter

Permite la colaboración entre objetos con interfaces incompatibles.



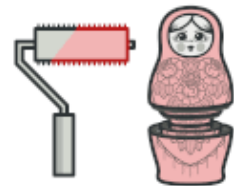
Bridge

Permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



Composite

Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.



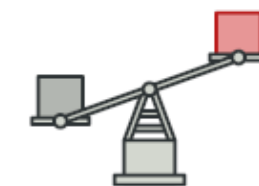
Decorator

Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



Facade

Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

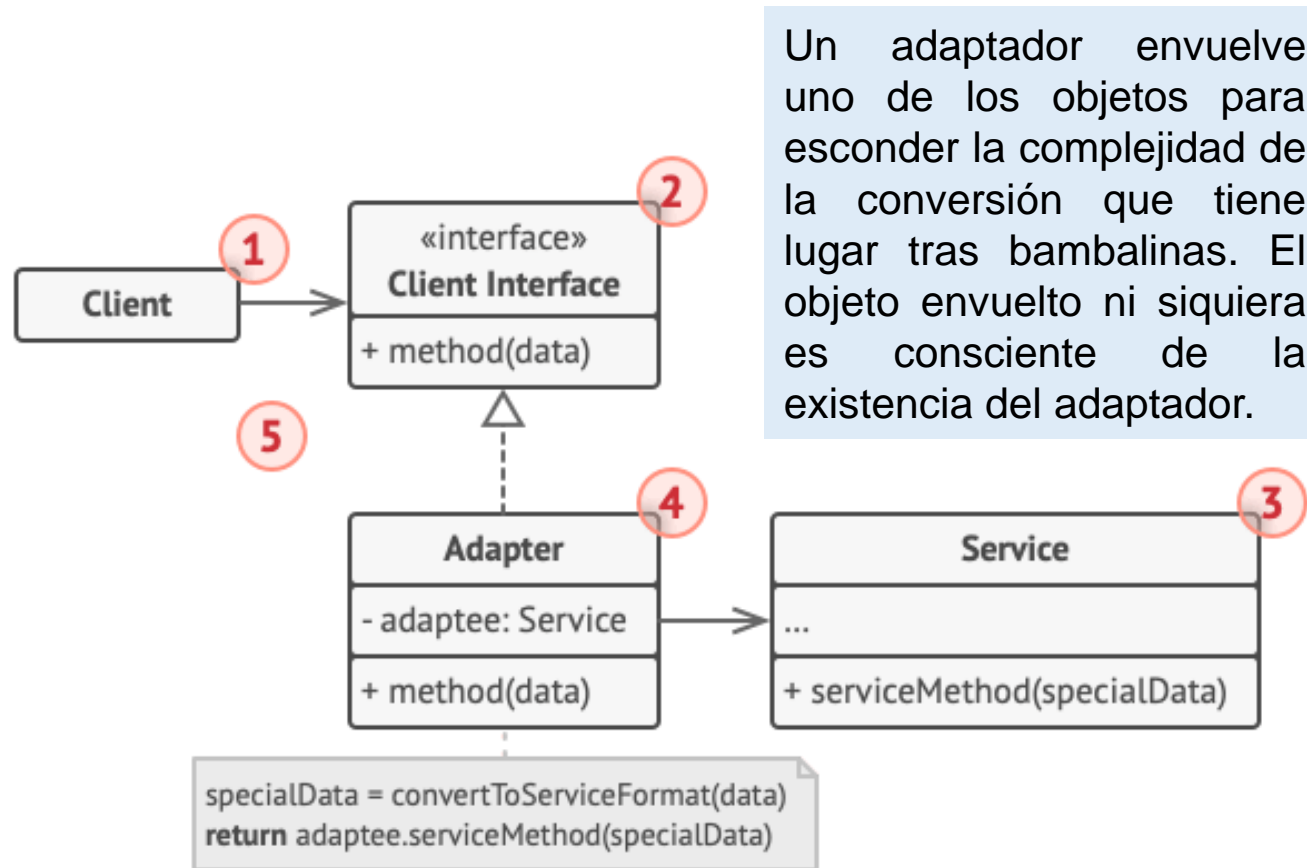


Flyweight

Permite mantener más objetos dentro de la cantidad disponible de memoria RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

3.3 Patrones Estructurales

ADAPTER



Un adaptador envuelve uno de los objetos para esconder la complejidad de la conversión que tiene lugar tras bambalinas. El objeto envuelto ni siquiera es consciente de la existencia del adaptador.

Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar. Funciona así:

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.

2. Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.

3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

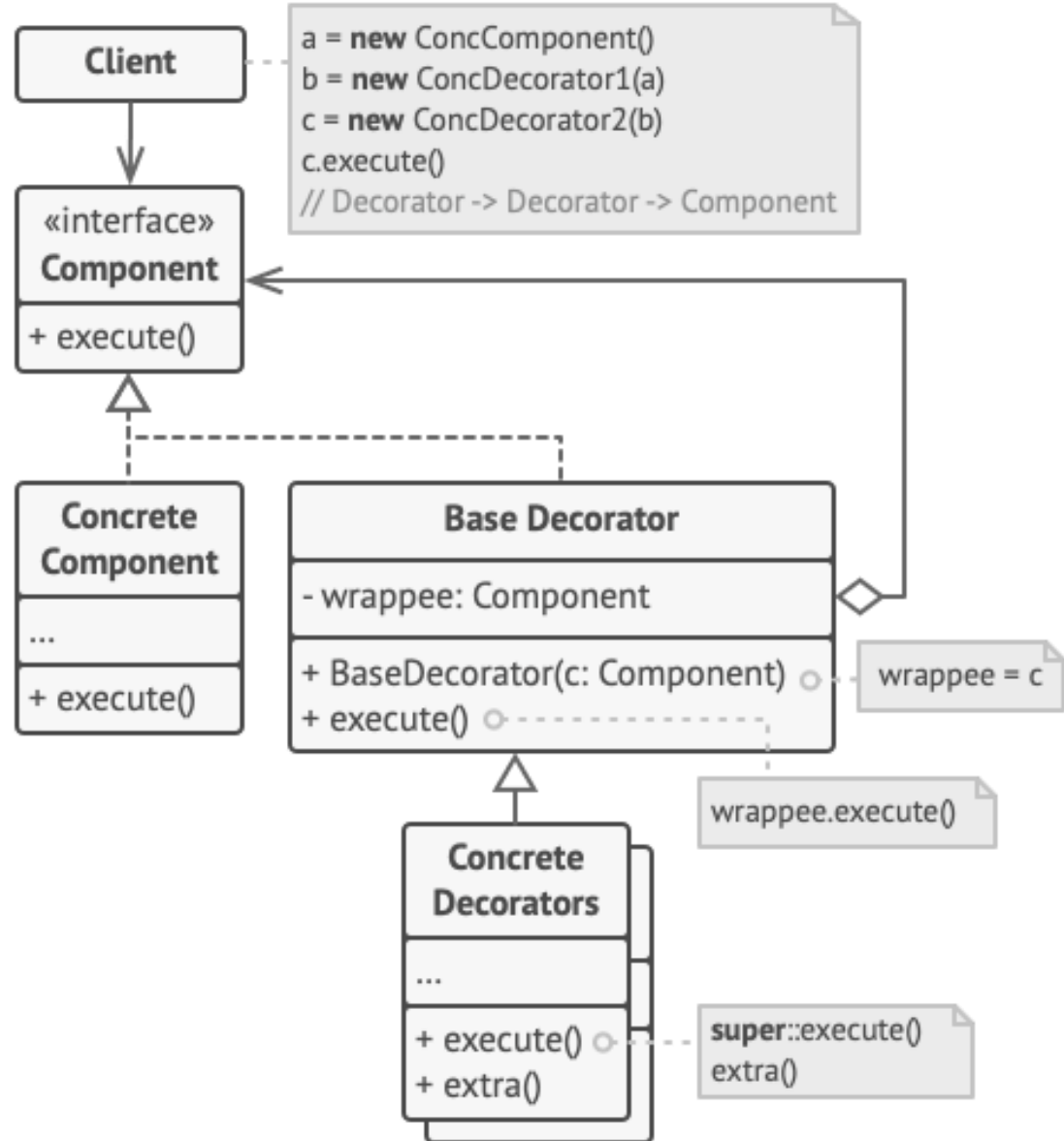
Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

En ocasiones se puede incluso crear un adaptador de dos direcciones que pueda convertir las llamadas en ambos sentidos.

3.3 Patrones Estructurales

DECORATOR

Decorator es un patrón de diseño estructural que te permite añadir funcionalidades sin necesidad de usar herencia (apila comportamientos)



1. El componente declara la interfaz común tanto para contenedores como para objetos envueltos.

2. Concrete Component es una clase de objetos que son “envueltos”. Define el comportamiento básico, que puede ser alterado por decoradores.

3. La clase Base Decorator tiene un campo para hacer referencia a un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto componentes concretos como decoradores. El decorador base delega todas las operaciones al objeto envuelto.

4. Los decoradores concretos definen comportamientos adicionales que se pueden agregar a los componentes de manera dinámica. Los decoradores concretos anulan los métodos del decorador base y ejecutan su comportamiento antes o después de llamar al método principal.

5. El Cliente puede envolver componentes en múltiples capas de decoradores, siempre que funcione con todos los objetos a través de la interfaz de componentes.

3.4 Patrones de Comportamiento

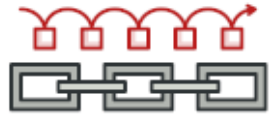
3.1 Concepto de Patrones de Diseño.

Patrones de Comportamiento (1/2)

| Patrón | Finalidad |
|-------------------------|--|
| Chain of Responsibility | Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena. |
| Command | Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar. |
| Iterator | Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.). |
| Mediator | Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador. |
| Memento | Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación. |
| Observer | Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando. |

3.1 Concepto de Patrones de Diseño.

Patrones de Comportamiento (1/2)



Chain of Responsibility

Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.



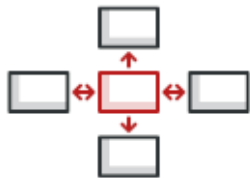
Command

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.



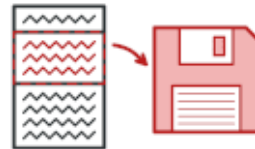
Iterator

Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).



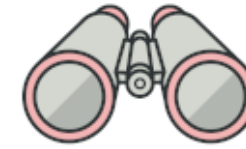
Mediator

Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.



Memento

Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.



Observer

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

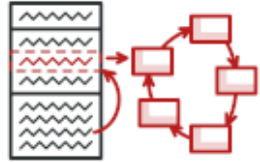
3.1 Concepto de Patrones de Diseño.

Patrones de comportamiento (2/2)

| Patrón | Finalidad |
|-----------------|--|
| State | Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase |
| Strategy | Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables. |
| Template Method | Define el esqueleto de un algoritmo en la superclase pero permite que las subclasses sobrescriban pasos del algoritmo sin cambiar su estructura. |
| Visitor | Permite separar algoritmos de los objetos sobre los que operan. |

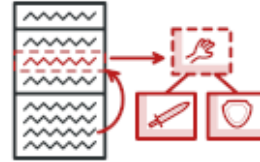
3.1 Concepto de Patrones de Diseño.

Patrones de comportamiento (2/2)



State

Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.



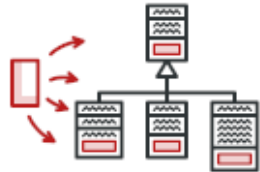
Strategy

Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.



Template Method

Define el esqueleto de un algoritmo en la superclase pero permite que las subclasses sobrescriban pasos del algoritmo sin cambiar su estructura.

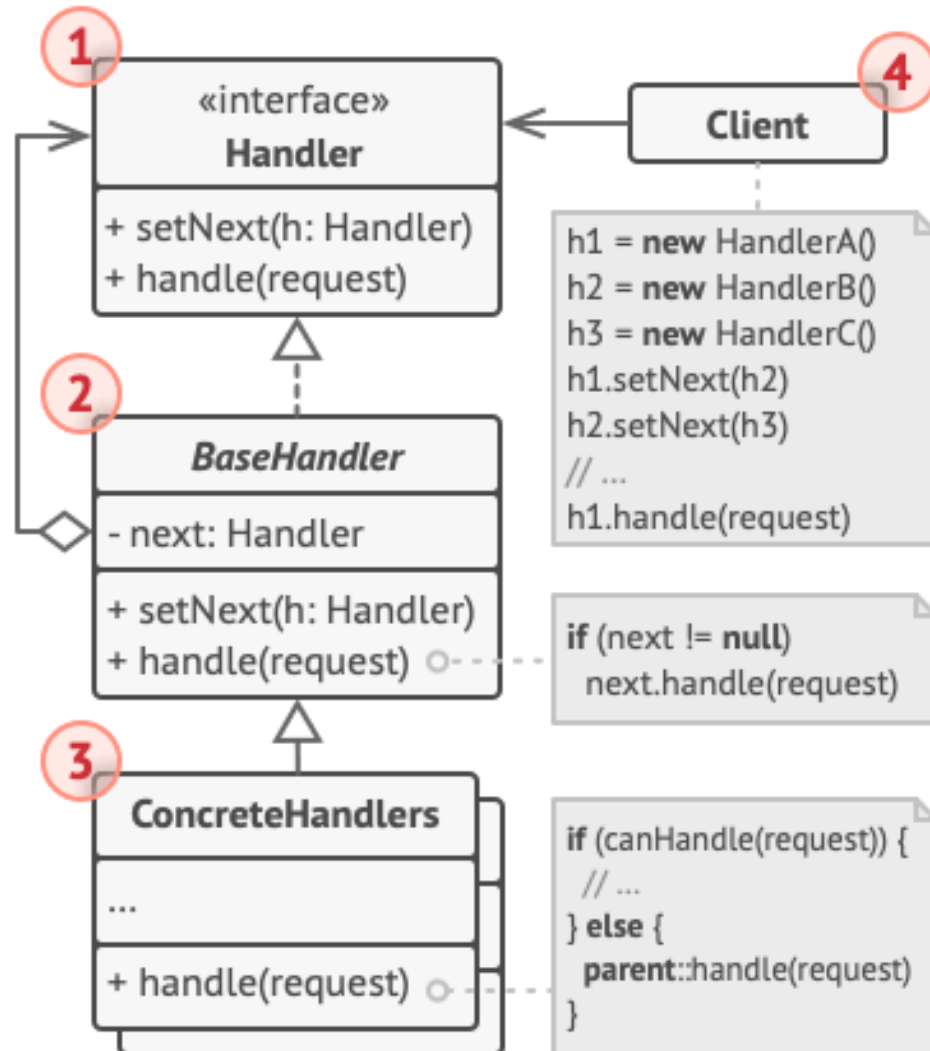


Visitor

Permite separar algoritmos de los objetos sobre los que operan.

3.4 Patrones de Comportamiento

Chain of Responsibility



1. El Handler declara la interfaz, común para todos los manipuladores concretos. Por lo general, contiene un solo método para manejar solicitudes, pero a veces también puede tener otro método para configurar el siguiente controlador de la cadena.

2. El controlador base es una clase opcional en la que puede colocar el código repetitivo que es común a todas las clases de controladores.

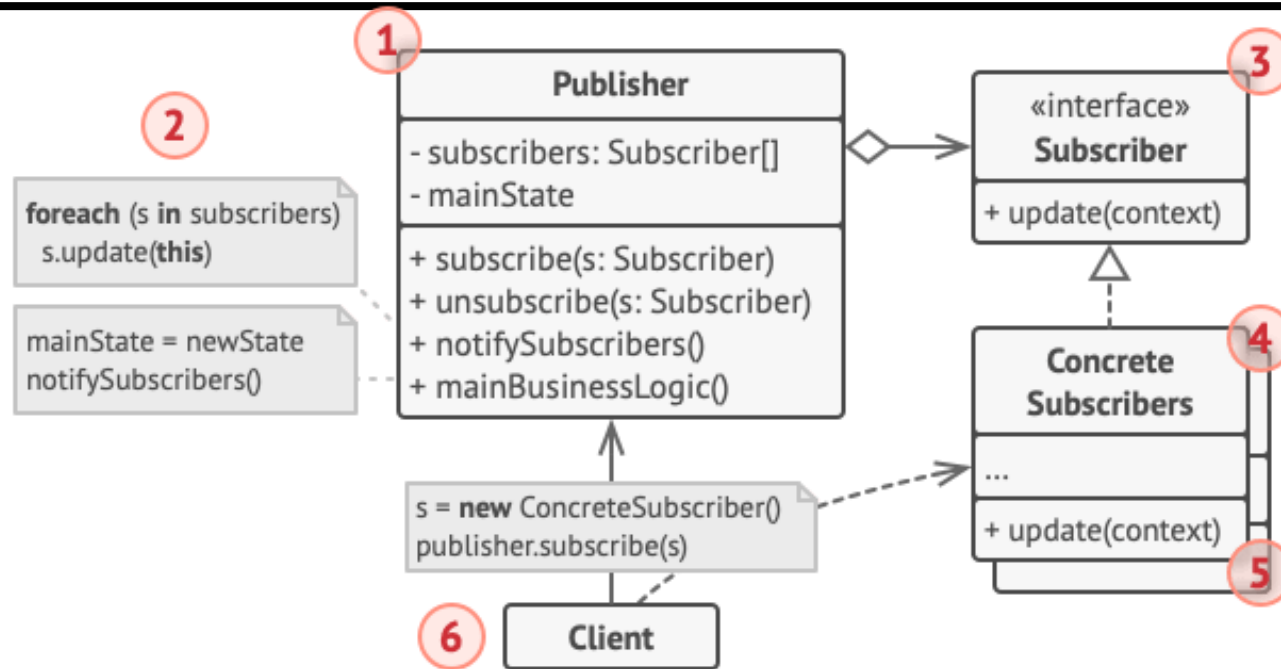
Por lo general, esta clase define un campo para almacenar una referencia al siguiente controlador. Los clientes pueden construir una cadena pasando un manejador al constructor o definidor del manejador anterior. La clase también puede implementar el comportamiento de manejo predeterminado: puede pasar la ejecución al siguiente controlador después de verificar su existencia.

3. Los manipuladores concretos contienen el código real para procesar las solicitudes. Al recibir una solicitud, cada manejador debe decidir si la procesa y, además, si la pasa a lo largo de la cadena. Los controladores suelen ser autónomos e inmutables, y aceptan todos los datos necesarios solo una vez a través del constructor.

4. El Cliente puede componer cadenas solo una vez o componerlas dinámicamente, dependiendo de la lógica de la aplicación. Tenga en cuenta que se puede enviar una solicitud a cualquier controlador de la cadena; no es necesario que sea el primero.

3.4 Patrones de Comportamiento

Observer



1. El editor emite eventos de interés para otros objetos. Estos eventos ocurren cuando el editor cambia su estado o ejecuta algunos comportamientos. Los editores contienen una infraestructura de suscripción que permite que se unan nuevos suscriptores y que los suscriptores actuales abandonen la lista.

2. Cuando ocurre un nuevo evento, el editor revisa la lista de suscripción y llama al método de notificación declarado en la interfaz del suscriptor en cada objeto de suscriptor.

3. La interfaz del suscriptor declara la interfaz de notificación. En la mayoría de los casos, consta de un único método de actualización. El método puede tener varios parámetros que permiten al editor pasar algunos detalles del evento junto con la actualización.

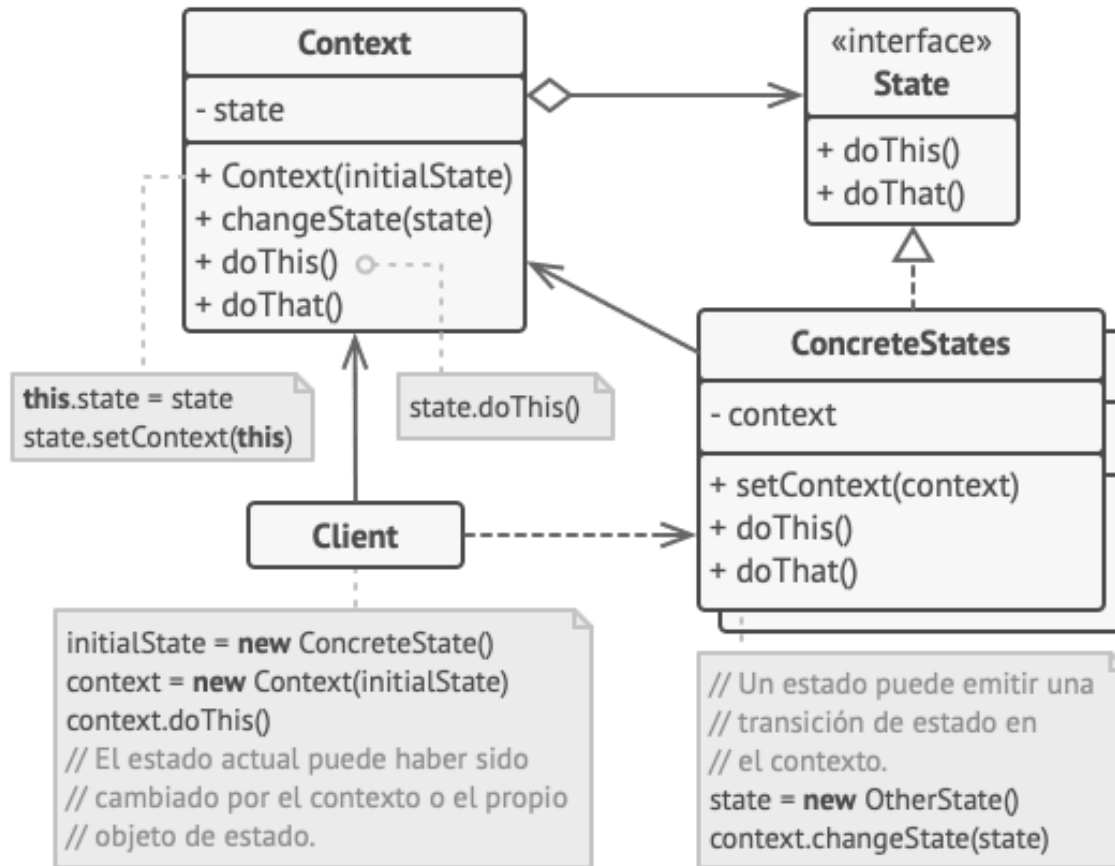
4. Los suscriptores concretos realizan algunas acciones en respuesta a las notificaciones emitidas por el editor. Todas estas clases deben implementar la misma interfaz para que el editor no esté asociado a clases concretas.

5. Por lo general, los suscriptores necesitan información contextual para manejar la actualización correctamente. Por esta razón, los editores suelen pasar algunos datos de contexto como argumentos del método de notificación. El editor puede hacerse pasar por un argumento, lo que permite que el suscriptor obtenga los datos necesarios directamente.

6. El Cliente crea objetos de publicador y de suscriptor por separado y luego registra a los suscriptores para las actualizaciones.

3.4 Patrones de Comportamiento

State



1. La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.

2. La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.

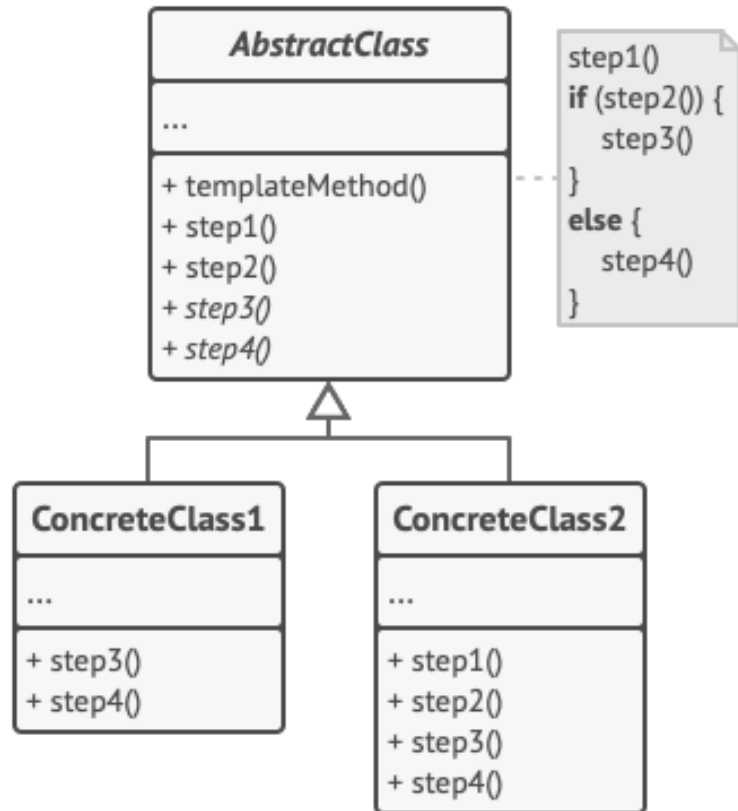
3. Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

4. Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

3.4 Patrones de Comportamiento

Template

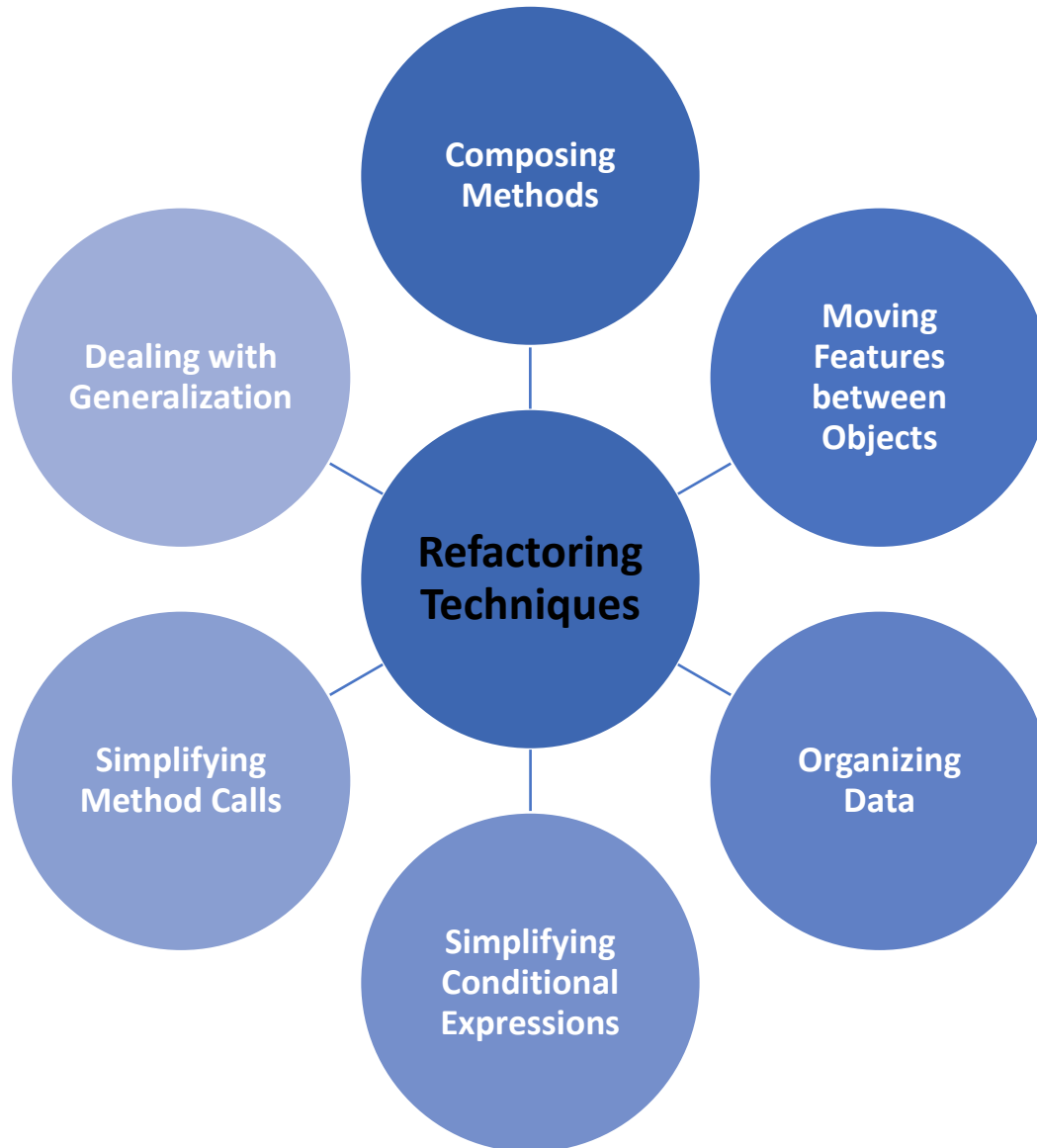


1. Clase Abstracta declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse abstractos o contar con una implementación por defecto.

2. Las **Clases Concretas** pueden sobrescribir todos los pasos, pero no el propio método plantilla.

3.5* Refactorización

Refactoring techniques



- Extract Method
- Inline Method
- Extract Variable
- Inline Temp
- Replace Temp with Query
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm
- Move Method
- Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension
- [...]

<https://refactoring.com/catalog/>

<https://refactoring.guru/refactoring/techniques>

Bibliografía

- ❖ Gamma, E., Johnson, R., Helm, R., Johnson, R. E., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

❖ **Sumérgete en los patrones de diseño.** V2021-1.7. Alexander Shvets.

<https://refactoring.guru/es/design-patterns/book>

Versión online: <https://refactoring.guru/es/design-patterns/catalog>

- ❖ Freeman, E., Robson, E., Bates, B., & Sierra, K. (2008). *Head first design patterns*. " O'Reilly Media, Inc.".
- ❖ Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, 1(34), 597.

Técnicas de Programación Avanzada (Java)

Curso 2021-2021

```
exit(); //Gracias!
```