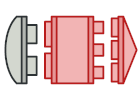
# PATRONES DE DISEÑO ESTRUCTURALES

### OBJETIVO

Aprender a programar patrones estructurales Adaptador y Decorador

### INTRODUCCIÓN

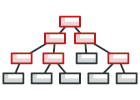
Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.

[](https://refactoring.guru/es/design-patterns/adapter)**[Adapter](https://refactoring.guru/es/design-patterns/adapter)**

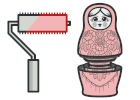
[Permite la colaboración entre objetos con interfaces incompatibles.](https://refactoring.guru/es/design-patterns/adapter)

[](https://refactoring.guru/es/design-patterns/bridge)**[Bridge](https://refactoring.guru/es/design-patterns/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.](https://refactoring.guru/es/design-patterns/bridge)

[](https://refactoring.guru/es/design-patterns/composite)**[Composite](https://refactoring.guru/es/design-patterns/composite)**

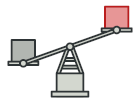
[Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.](https://refactoring.guru/es/design-patterns/composite)

[](https://refactoring.guru/es/design-patterns/decorator)**[Decorator](https://refactoring.guru/es/design-patterns/decorator)**

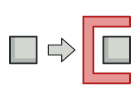
[Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.](https://refactoring.guru/es/design-patterns/decorator)

[](https://refactoring.guru/es/design-patterns/facade)**[Facade](https://refactoring.guru/es/design-patterns/facade)**

[Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.](https://refactoring.guru/es/design-patterns/facade)

[](https://refactoring.guru/es/design-patterns/flyweight)**[Flyweight](https://refactoring.guru/es/design-patterns/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.](https://refactoring.guru/es/design-patterns/flyweight)

[](https://refactoring.guru/es/design-patterns/proxy)**[Proxy](https://refactoring.guru/es/design-patterns/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.](https://refactoring.guru/es/design-patterns/proxy)

1. **El patrón Adaptador: patrón estructural**

El patrón Adaptador permite que objetos con interfaces incompatibles trabajen juntos. Actúa como un puente entre dos interfaces.

Características:

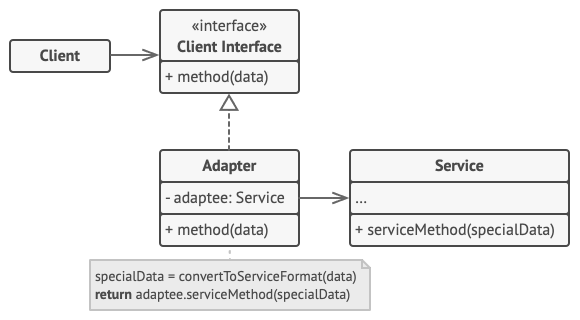
Convierte la interfaz de una clase en otra interfaz que el cliente espera.

Permite que clases con interfaces incompatibles cooperen.

Uso común:

Cuando se necesita adaptar una clase existente con una interfaz que no es compatible con el sistema existente.

Diagrama de clases



1. **Patrón Decorador: patrón estructural**

El patrón Decorador añade responsabilidades adicionales a un objeto dinámicamente. No altera la estructura original del objeto.

Características:

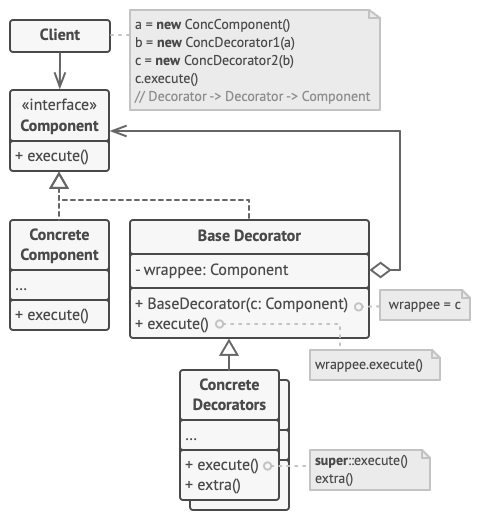
Añade funcionalidades a objetos sin cambiar su estructura.

Permite extender las capacidades de un objeto de manera flexible.

Uso común:

Cuando quieres añadir nuevas funcionalidades o comportamientos a objetos existentes de forma dinámica, sin modificar su código original.

Diagrama de clases



Taller para la entrega

1. Una de las operaciones en la clase **java.util.Collections** es **static Collection unmodifiableCollection(Collection c)**. Esta operación acepta una colección arbitraria como parámetro y devuelve una colección que tiene los mismos elementos pero que no puede ser modificados. Cualquier operación de la colección que intente modificarla lanzará una **UnsupportedOperationException**. La interface se muestra en la Figura 1.

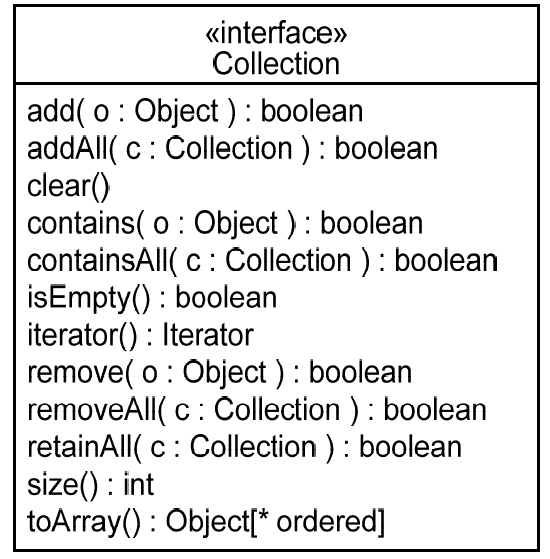


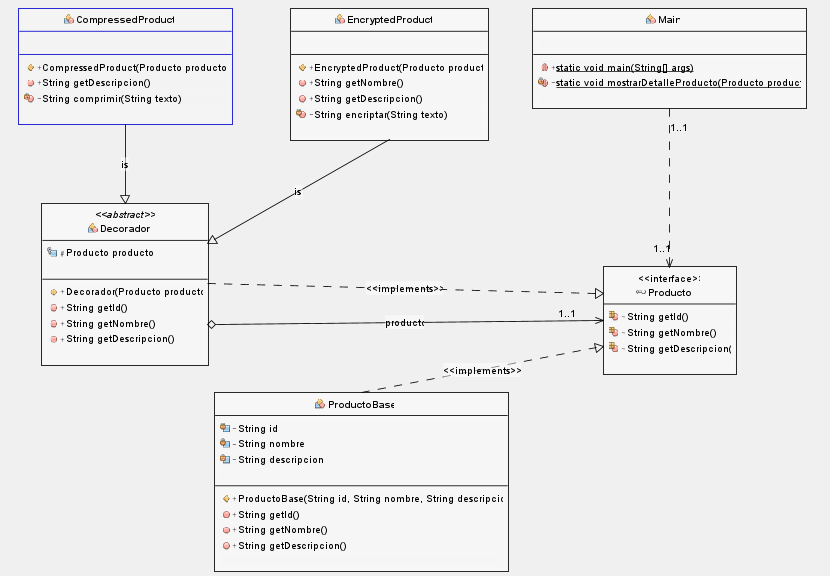
Figure 1. Interface Collection

* + - 1. ¿Cómo podría implementarse la operación **unmodifiableCollection()**?  
           
         Rta: La operación **unmodifiableCollection()** podría implementarse mediante un patrón de diseño decorador y adaptador. Estos patrones envuelven la colección original con un decorador o adaptador que intercepta y controla los intentos de modificación.
      2. Puedes usar un patrón Adapter para resolver este problema de diseño. ¿Cuál de los patrones Adapter es el más apropiado y por qué?  
           
         Rta: Si, se va a utilizar el adpater con base a objetos, dado que el comportamiento de estos varía dependiendo de las funcionalidades que estos deban realizar, es decir al aplicar el adapter por objetos, para resumir en cuanto al patrón Adapter más apropiado, el Adapter de objetos es la mejor opción, ya que permite adaptar las interfaces de las colecciones de forma dinámica según las funcionalidades requeridas, brindando mayor flexibilidad.
      3. Suponga que se cuenta con una operación para crear versiones no modificables de una clase (en lugar de una interfaz). ¿Qué patrón Adapter usarías en ese caso y por qué?  
           
         Rta: Si se tiene una operación para crear versiones no modificables de una clase concreta en lugar de una interfaz, el patrón Adapter más apropiado sería el Adapter de clases.  
         El Adapter de clases se utiliza cuando no es posible modificar el código fuente de una clase existente, ya sea porque está en una biblioteca externa o por alguna otra restricción. En este caso, como se trata de una clase concreta y no una interfaz, no se puede utilizar el Adapter de objetos, ya que este requiere la implementación de una interfaz para adaptarla.
      4. Realiza un diagrama de clases detallando tu diseño para implementar la operación **unmodifiableCollection()** utilizando el patrón Adapter que hayas elegido. Indica con estereotipos qué clase es el adaptador y cuál es la adaptada.  
         Rta:  
         Diagrama, Tabla

         Descripción generada automáticamente con confianza media
      5. EscribeEscriba el código en Java de acuerdo con su diseño. En particular, escribe código para la operación **unmodifiableCollection()**, el constructor de la clase adaptador y las operaciones ***isEmpty()*** y ***add()*** en la clase adaptador.
      6. Analize si esta solución cumple con el principio de substitución de liskov o lo viola.  
           
         Rta: En este caso, la solución viola el principio de substitución de Liskov. Esto se debe a que los objetos de la clase UnmodifiableCollectionAdapter no pueden ser completamente substituidos por objetos de la interfaz Collection, ya que no admiten todas las operaciones definidas por la interfaz.

1. Haciendo uso del patrón decorador dé la posibilidad de encriptar o comprimir o ambos, la información de los productos: id, nombre y descripción. Por ejemplo, suponga que la encripción aplica el carácter siguiente, si el producto es “Arroz chino” la encripción será “Bsspa dijop”. La compresión es más compleja, pero suponga que existe un algoritmo para comprimir y otro para descomprimir cuya implementación es por lo pronto irrelevante. Proponga un diagrama de clases e implemente decoradores a la clase producto de tal forma que se tenga EncryptedProduct, CompressedProducto, así mismo construya una implementación en java del objeto original y los decoradores.

Rta:



1. Investigue como trabaja en patrón fachada y aplíquelo agrupando varios servicios segregados en una única clase que los recibe, los procesa y los despacha a cada uno de los destinos.

Condiciones de la entrega

Se debe entregar la respuesta a las actividades I y II de acuerdo a lo establecido en las preguntas