Análisis y diseño de aplicaciones I

UT5 – Patrones de Estructura

Agenda

UCU

- Patrones Estructurales
  - Definición
  - Adapter
  - Bridge
  - Decorator
  - Facade
  - Proxy
  - DAO
  - Repository

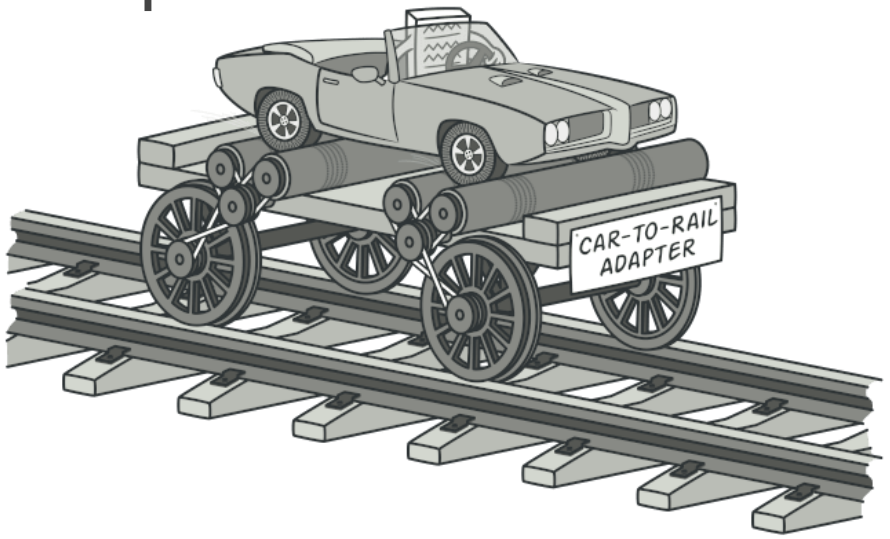
Patrones Estructurales

UCU

- Los patrones de diseño estructural son patrones que facilitan el diseño al identificar una forma sencilla de realizar relaciones entre entidades
- 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.

Adapter

- Es un patrón de diseño estructural que permite la **colaboración entre objetos con interfaces incompatibles**.
- A grandes rasgos, se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

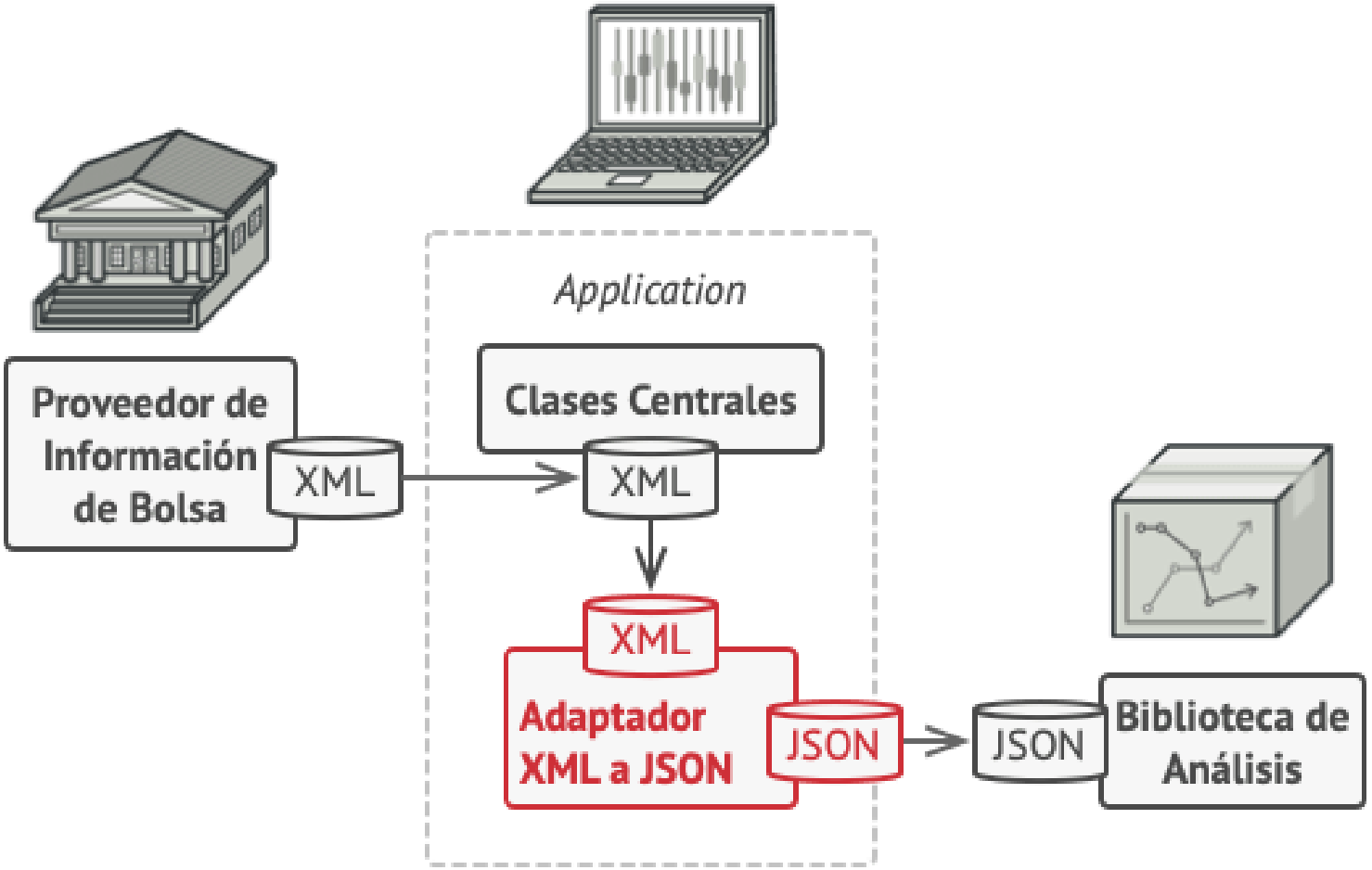


Adapter

UCU

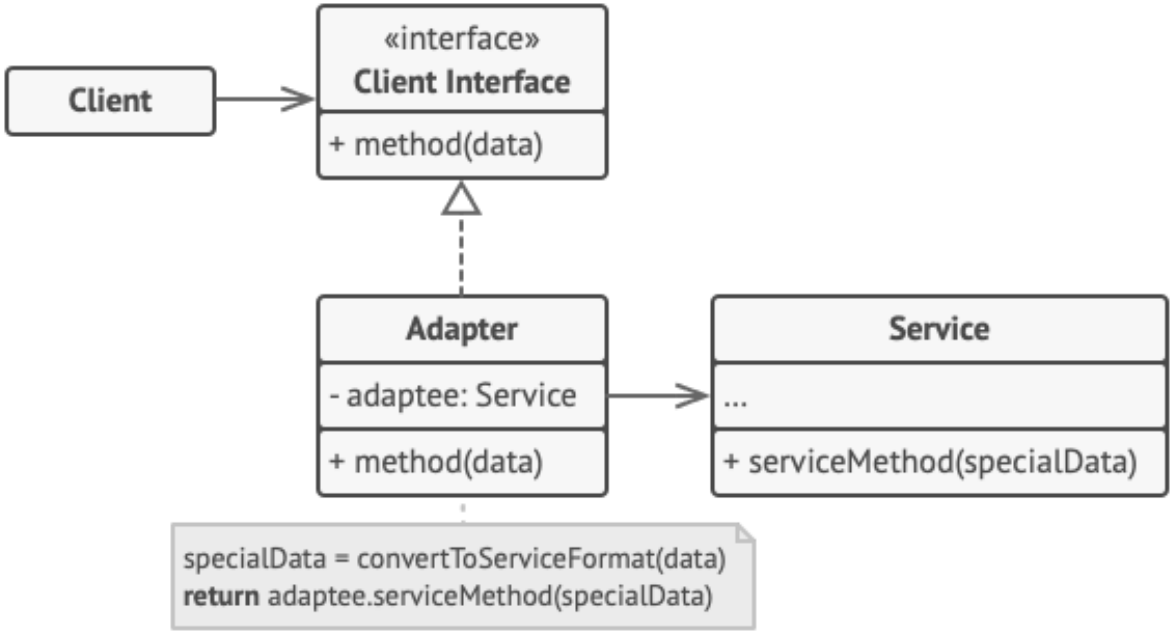
- Los adaptadores ayudan a objetos con distintas interfaces a colaborar entre sí.
  1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
  2. Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.
  3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

Adapter


Application
Proveedor de Información de Bolsa
XML
Clases Centrales
XML
XML
Adaptador XML a JSON
JSON
JSON
Biblioteca de Análisis

Adapter

La clase Adaptadora es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio. La clase adaptadora recibe llamadas del cliente a través de la interfaz adaptadora y las traduce en llamadas al objeto envuelto de la clase de servicio, pero en un formato que pueda comprender.



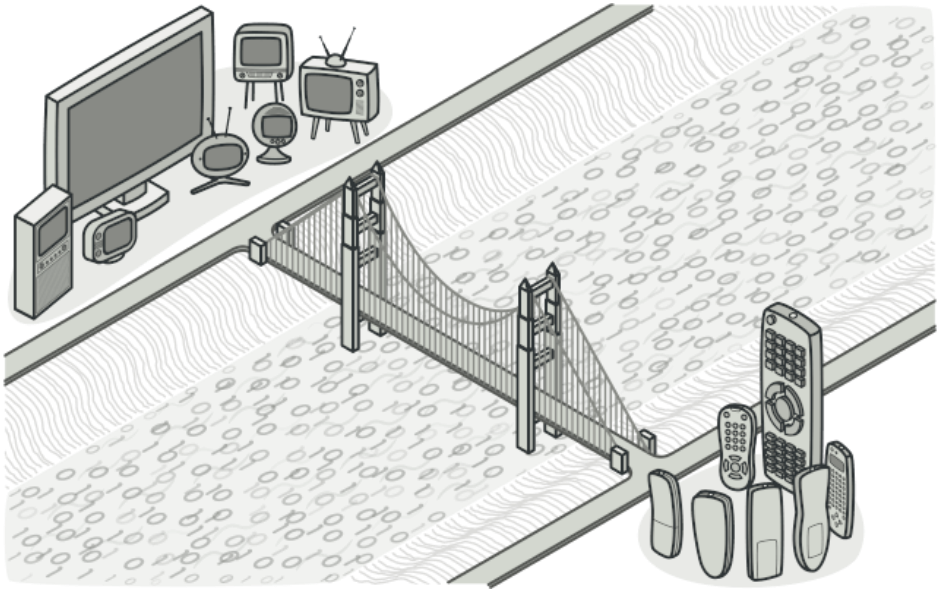
Adapter

UCU

- Utiliza este patrón cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con otra parte del código.
- Pros:
  - *Principio de responsabilidad única (*SRP*)*. Puedes separar la interfaz o el código de conversión, de la lógica de negocio.
  - *Principio de abierto/cerrado(*OCP*)*. Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente.
- Contra:
  - La complejidad general del código aumenta

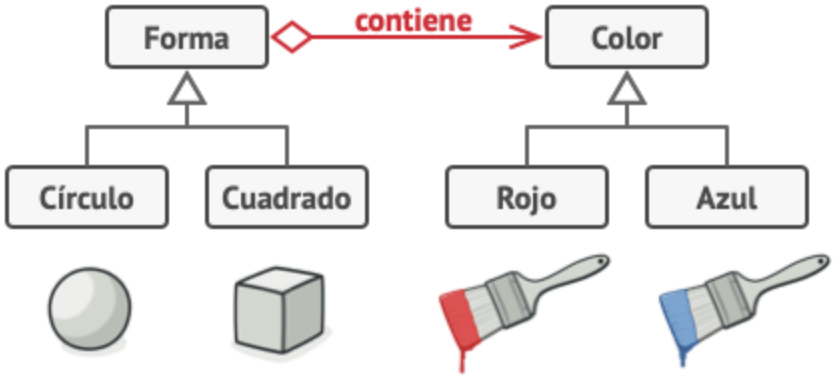
Bridge

- Es un patrón de diseño estructural que **permite dividir una clase, o un grupo de clases relacionadas, en dos jerarquías separadas** (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



Bridge

- El patrón Bridge pasa de la herencia a la composición del objeto. Esto quiere decir que se extrae una de las dimensiones a una jerarquía de clases separada, de modo que las clases originales referencian un objeto de la nueva jerarquía, en lugar de tener todo su estado y sus funcionalidades dentro de una clase.



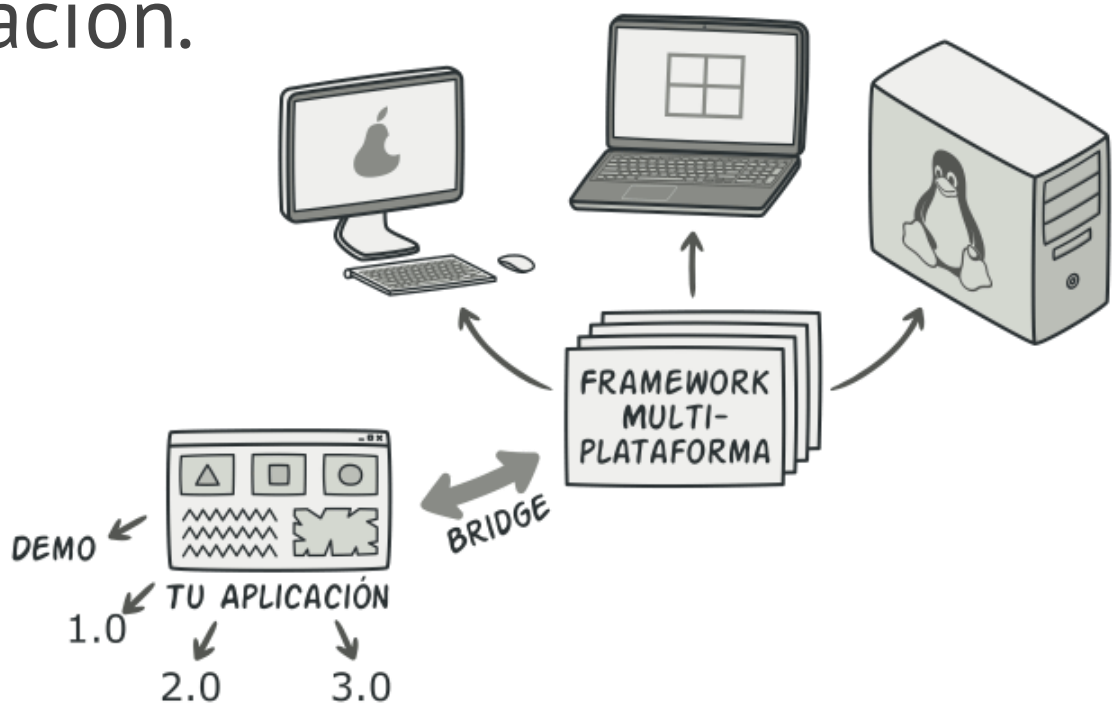
Bridge

UCU

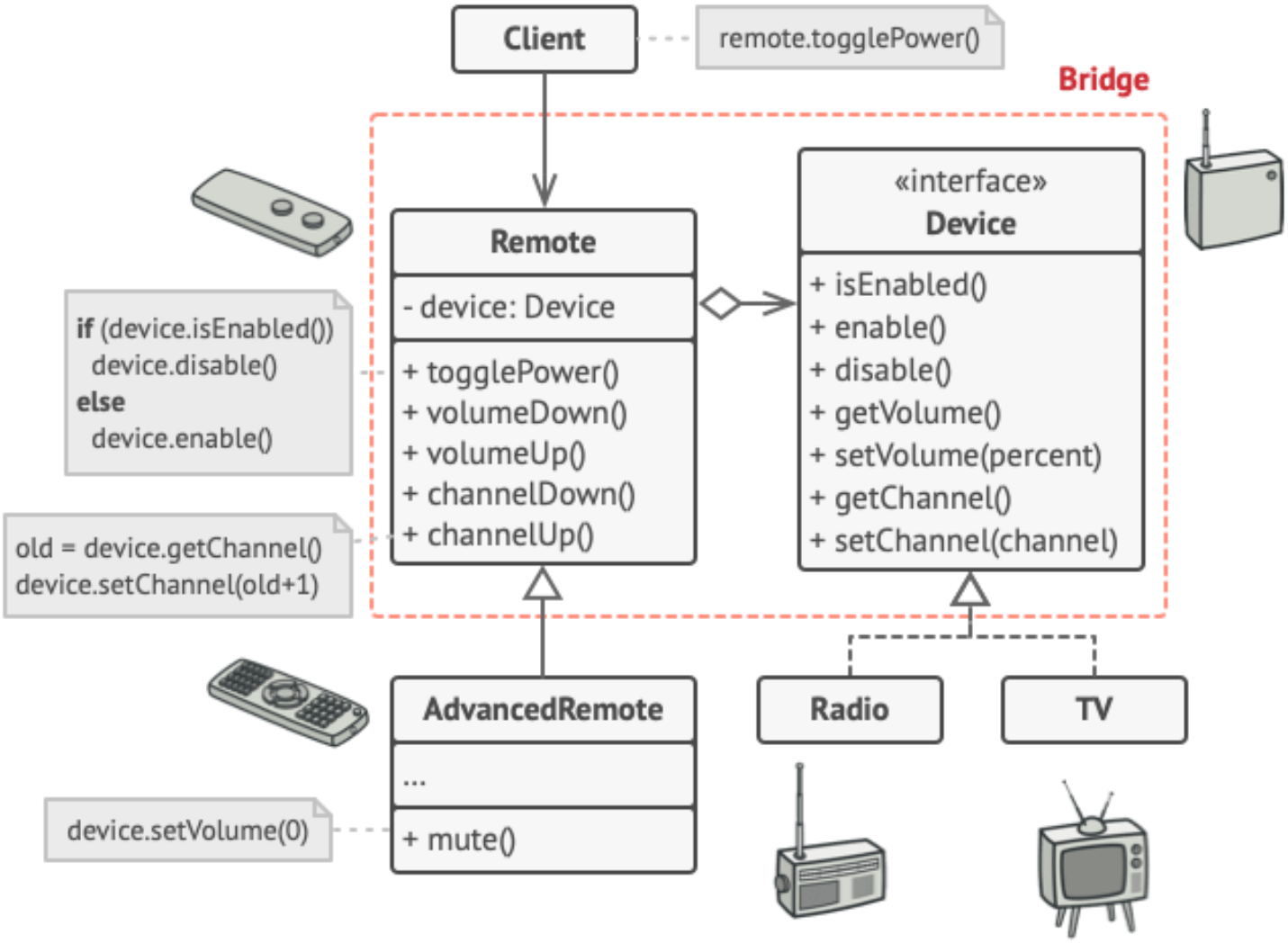
- La *Abstracción* (o *interfaz*) es una capa de control de alto nivel para una entidad. Esta capa no tiene que hacer ningún trabajo real por su cuenta, sino que debe delegar el trabajo a la capa de *implementación* (o *plataforma*).
- La *Abstracción* puede representarse por una interfaz gráfica (GUI), y la *implementación* puede ser el código del sistema operativo subyacente (API) a la que la capa GUI llama en respuesta a las interacciones del usuario.

Bridge

- Puedes cambiar las clases de la GUI sin tocar las clases relacionadas con la API. Además, añadir soporte para otro sistema operativo sólo requiere crear una subclase en la jerarquía de implementación.



Bridge


Client
remote.togglePower()
Bridge
Remote
- device: Device
+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()
«interface»
Device
+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)
if (device.isEnabled())
device.disable()
else
device.enable()
old = device.getChannel()
device.setChannel(old+1)
AdvancedRemote
...
+ mute()
device.setVolume(0)
Radio
TV

Bridge

UCU

- La clase base de control remoto declara un campo de referencia que la vincula con un objeto de dispositivo. Todos los controles remotos funcionan con los dispositivos a través de la interfaz general de dispositivos, que permite al mismo remoto soportar varios tipos de dispositivos.
- El código cliente vincula el tipo deseado de control remoto con un objeto específico de dispositivo a través del constructor del control remoto.

Bridge

UCU

- Utiliza el patrón Bridge cuando quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos).
- Pros:
  - *Principio de abierto/cerrado* (OCP): Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.
  - *Principio de responsabilidad única (*SRP*)*. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.
- Contra:
  - Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.

Adapter vs Bridge

UCU

- Propósito:
  - **Adapter**: Su propósito principal es permitir que dos interfaces incompatibles trabajen juntas. Se utiliza para adaptar la interfaz de una clase existente a la que un cliente espera, sin modificar la clase original.
  - **Bridge**: Su objetivo es desacoplar una abstracción de su implementación para que puedan variar de forma independiente. Se utiliza para dividir una clase grande o un conjunto de clases estrechamente relacionadas en dos jerarquías separadas: abstracción e implementación.

Adapter vs Bridge

UCU

- Intención:
  - **Adapter**: Resolver problemas de incompatibilidad entre interfaces que ya existen. Es más sobre hacer que el código antiguo funcione con código nuevo o de terceros.
  - **Bridge**: Se planifica de antemano. Es más sobre la organización del código de tal manera que la abstracción y la implementación puedan ser extendidas independientemente.

Adapter vs Bridge

UCU

- Estructura:
  - **Adapter**: En general, el patrón Adapter envuelve la clase adaptada y expone una interfaz que el cliente puede usar. No hay una separación clara entre abstracción e implementación.
  - **Bridge**: Tiene una estructura más compleja con una clara separación entre la jerarquía de abstracción y la jerarquía de implementación.

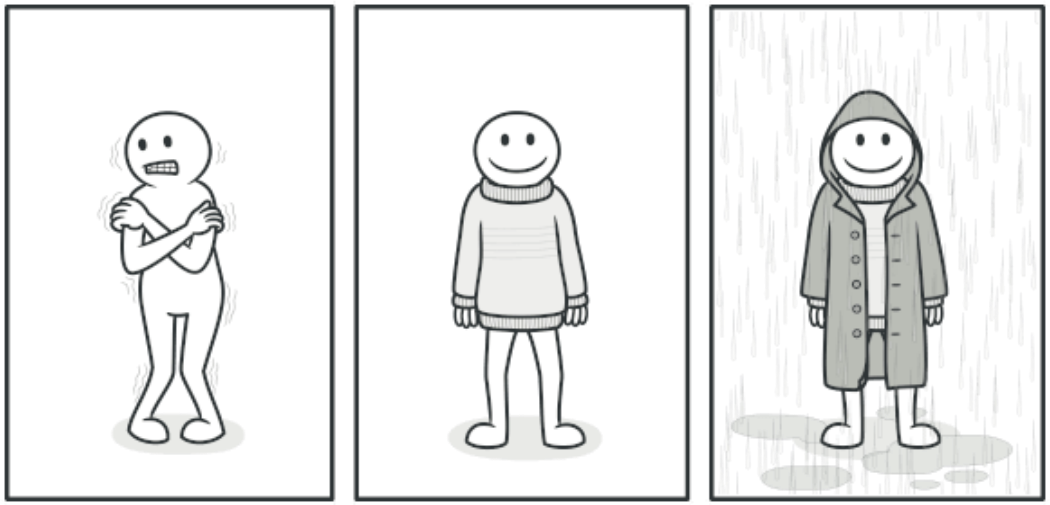
  En resumen, usa el patrón Adapter cuando necesitas que clases con interfaces incompatibles trabajen juntas. Usa el patrón Bridge cuando quieras separar y organizar una clase (o un conjunto de clases) en dos jerarquías separadas (abstracción e implementación) para que puedan ser extendidas independientemente.

Decorator

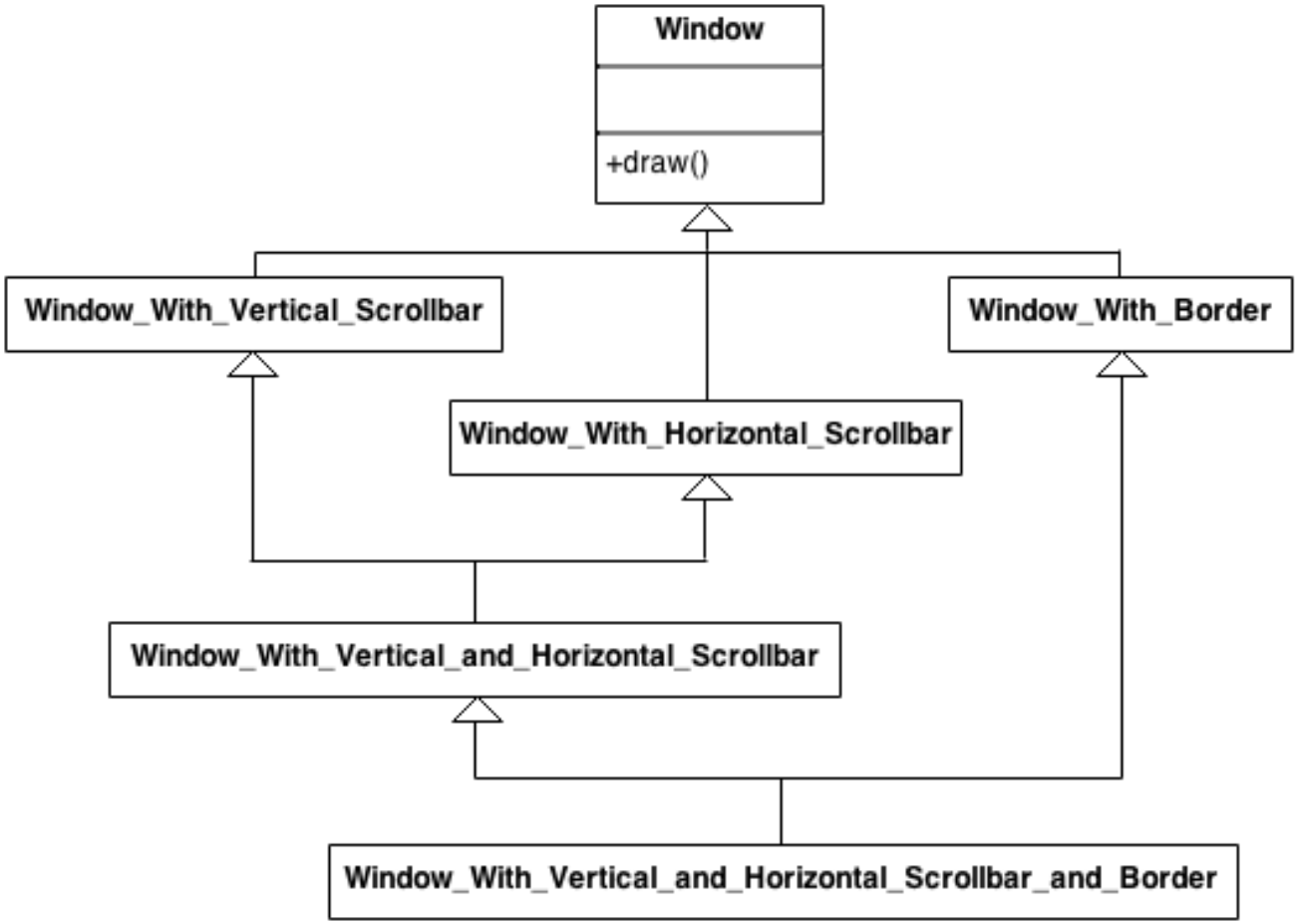
- Es un patrón de diseño estructural que te **permite añadir funcionalidades a objetos**. Los decoradores proporcionan una alternativa flexible a la subclasificación para ampliar la funcionalidad.

Decorator

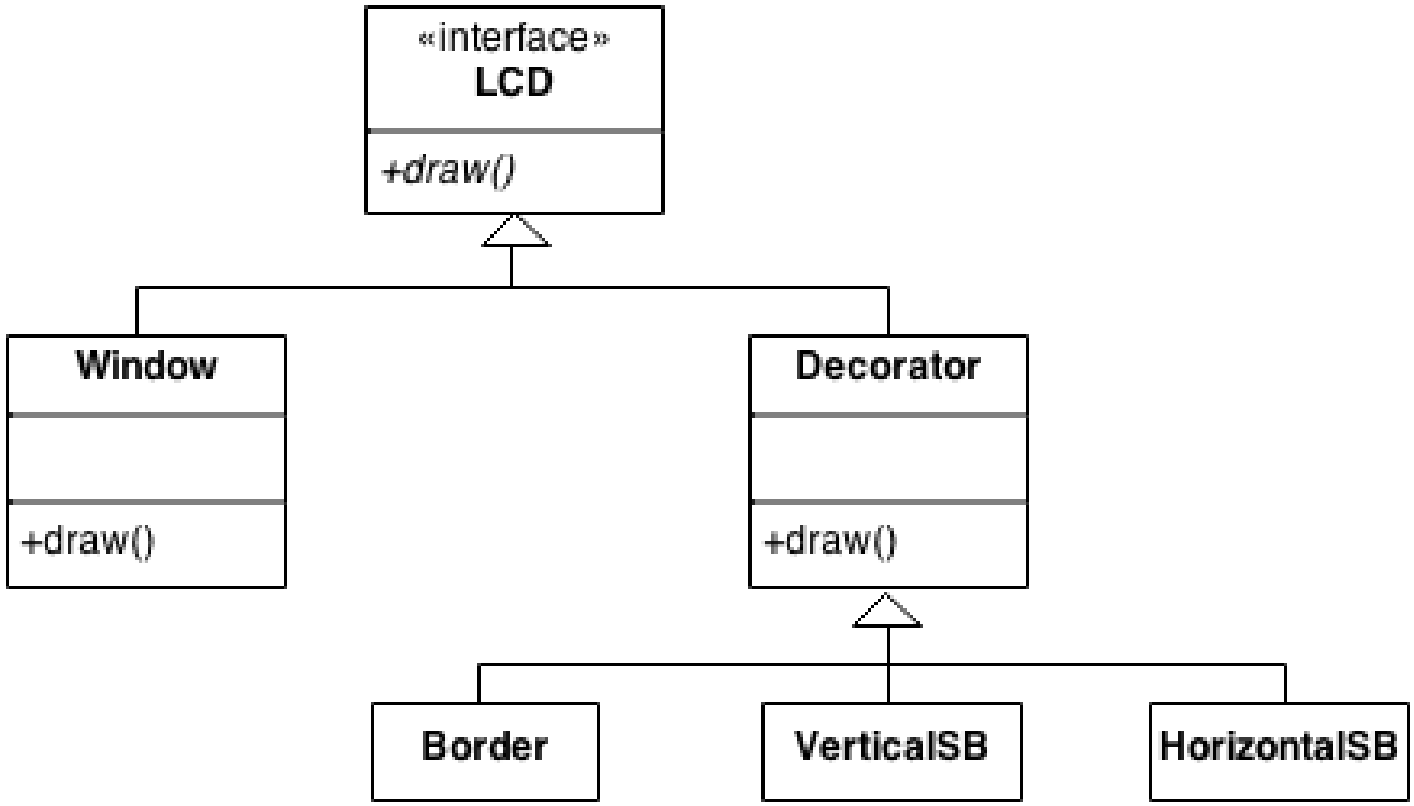
- La solución consiste en encapsular el objeto original dentro de una interfaz contenedora abstracta. Tanto los objetos decoradores como el objeto central heredan de esta interfaz abstracta. La interfaz utiliza una composición recursiva para permitir que se agregue un número ilimitado de "capas" de decoradores a cada objeto principal.



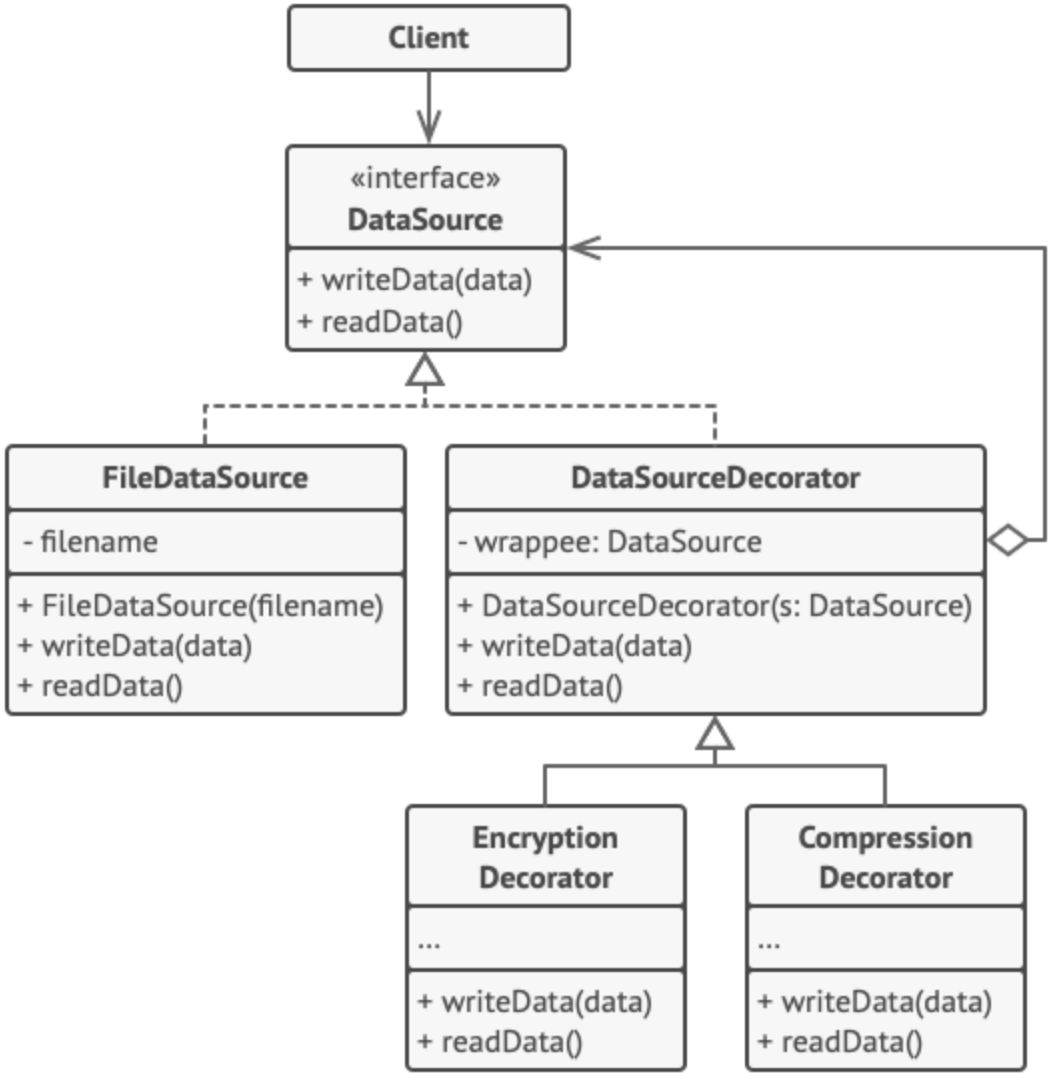
Decorator


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Horizontal_Scrollbar
Window_With_Border
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

Decorator


«interface»
LCD
+draw()
Window
+draw()
Decorator
+draw()
Border
VerticalSB
HorizontalSB

Decorator


Client
«interface»
DataSource
+ writeData(data)
+ readData()
FileDataSource
- filename
+ FileDataSource(filename)
+ writeData(data)
+ readData()
DataSourceDecorator
- wrappee: DataSource
+ DataSourceDecorator(s: DataSource)
+ writeData(data)
+ readData()
Encryption Decorator
...
+ writeData(data)
+ readData()
Compression Decorator
...
+ writeData(data)
+ readData()

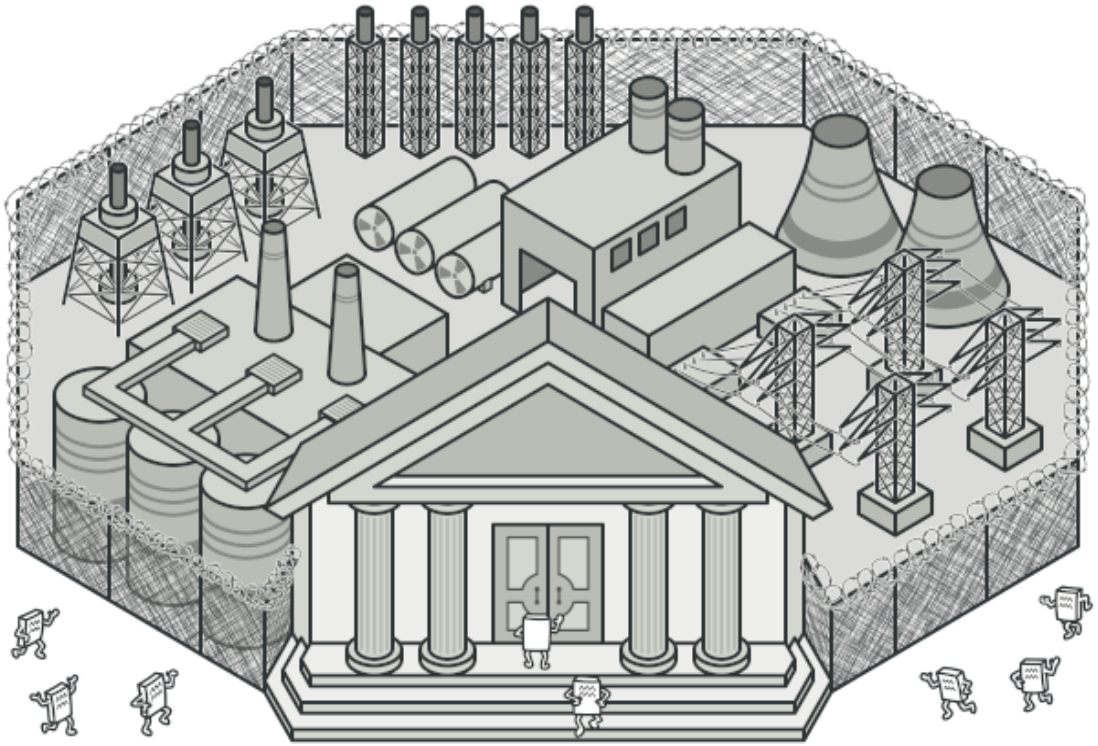
Decorator

UCU

- Utiliza este patrón cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.
- Utiliza el patrón cuando no sea posible extender el comportamiento de un objeto utilizando la herencia.
- Pros:
  - Principio de responsabilidad única (SRP). Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.
- Contras:
  - Resulta difícil eliminar un wrapper específico de la pila de wrappers.

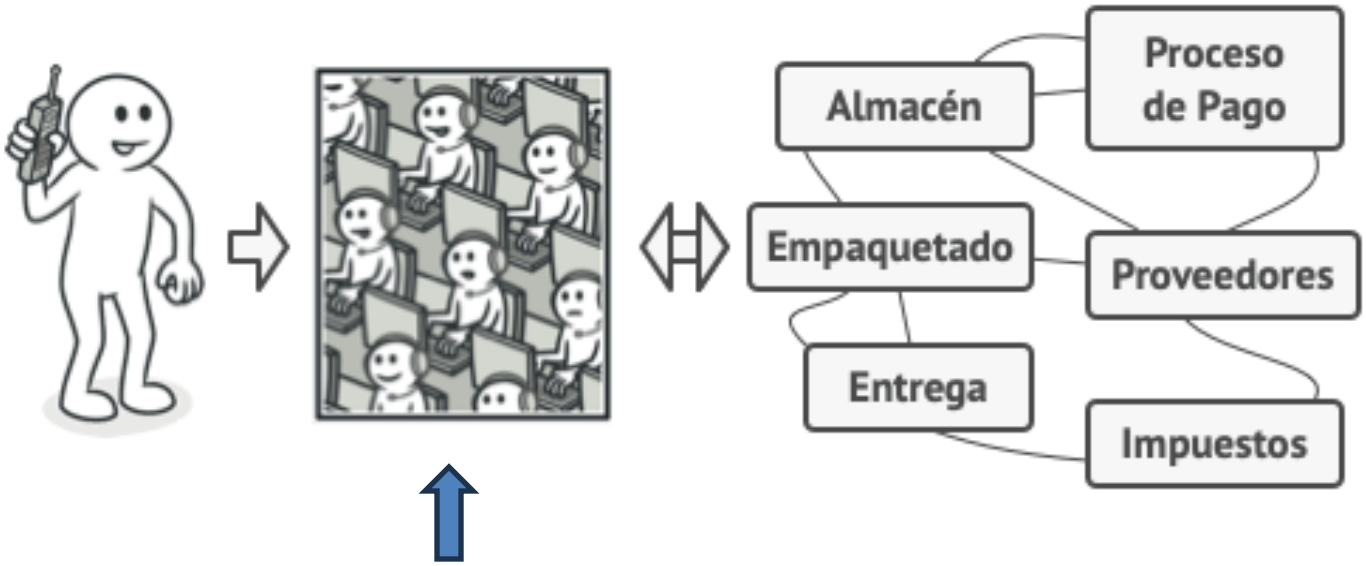
Facade

- Proporcionar una **interfaz unificada a un conjunto de interfaces en un subsistema**. Facade define una interfaz de nivel superior que facilita el uso del subsistema.
- Envuelve un subsistema complicado con una interfaz más simple.

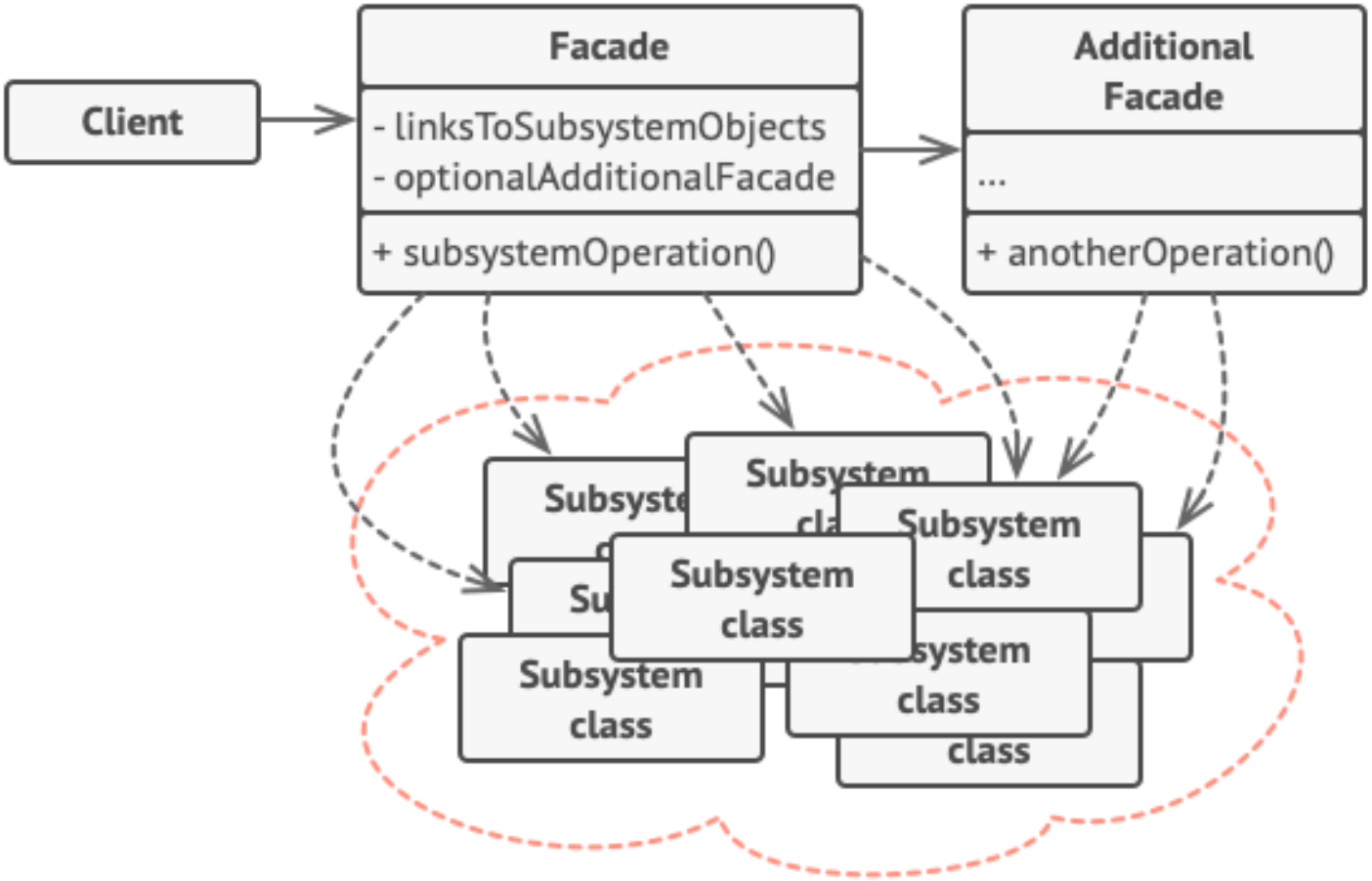


Facade

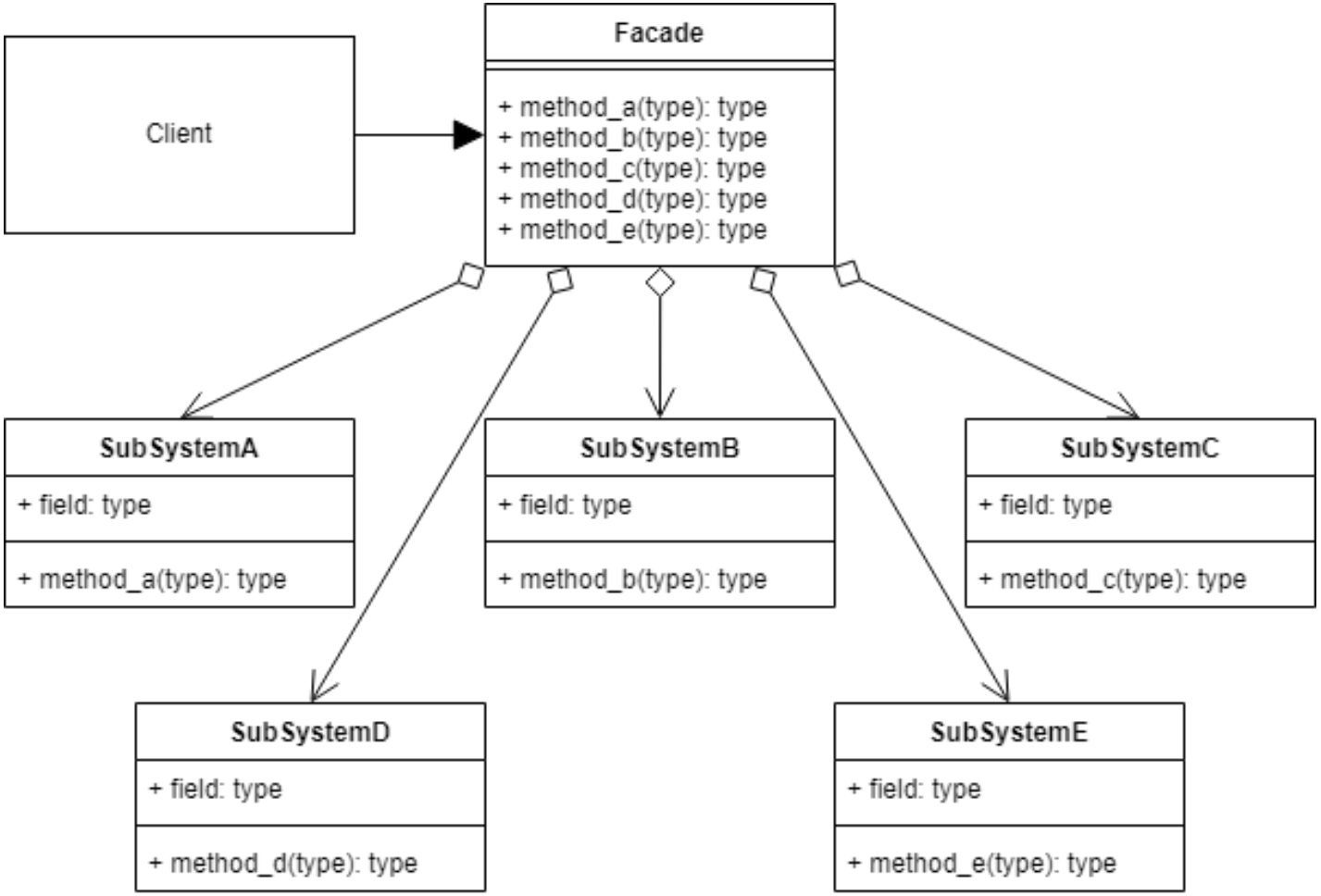
- Tener una fachada resulta útil cuando tienes que integrar tu aplicación con una biblioteca con decenas de funciones, de la cual sólo necesitas una pequeña parte.



Facade


Client
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional Facade
...
+ anotherOperation()
Subsystem class
Subsystem class
Subsystem class
Subsystem class
Subsystem class

Facade


Client
Facade
+ method_a(type): type
+ method_b(type): type
+ method_c(type): type
+ method_d(type): type
+ method_e(type): type
SubSystemA
+ field: type
+ method_a(type): type
SubSystemB
+ field: type
+ method_b(type): type
SubSystemC
+ field: type
+ method_c(type): type
SubSystemD
+ field: type
+ method_d(type): type
SubSystemE
+ field: type
+ method_e(type): type

Facade

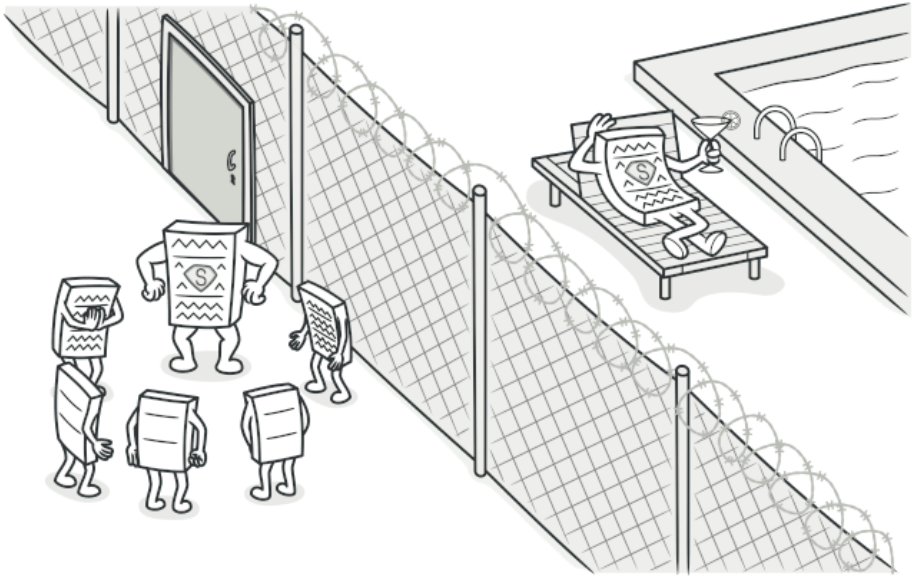
UCU

- Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.
- También cuando quieras estructurar un subsistema en capas. Permite reducir el acoplamiento.

- Pros:
  - Puedes aislar tu código de la complejidad de un subsistema.
  - Reducción del acoplamiento.
- Contras:
  - Una fachada puede convertirse en un objeto todopoderoso acoplado a todas las clases de una aplicación.

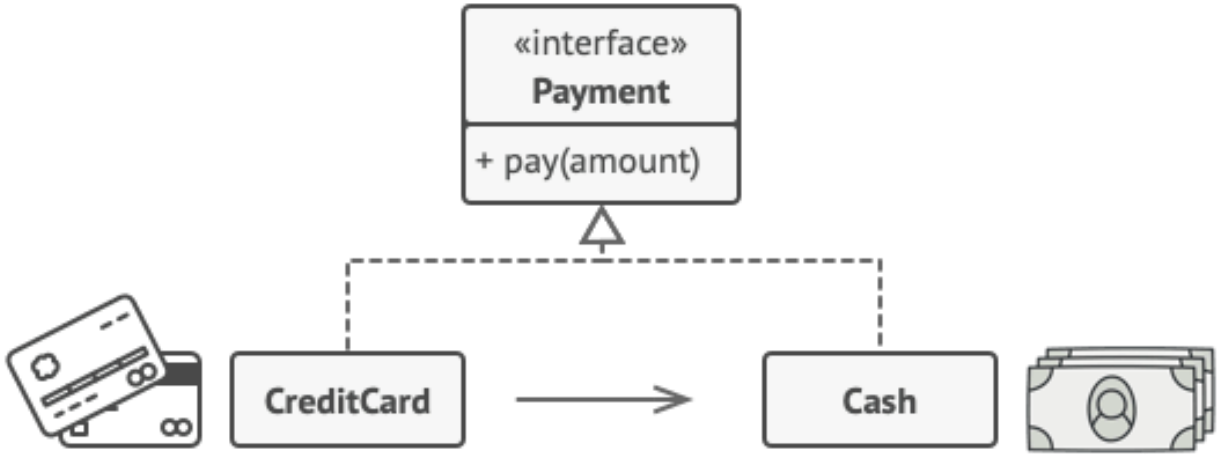
Proxy

- Proporcione un sustituto o marcador de posición para otro objeto para **controlar el acceso a él**.
- Agregue un contenedor y una delegación para proteger el componente real de una complejidad indebida.
- Un proxy con**trola el acceso al objeto original, permitiéndote hacer algo antes o después** de que la solicitud llegue al objeto original.

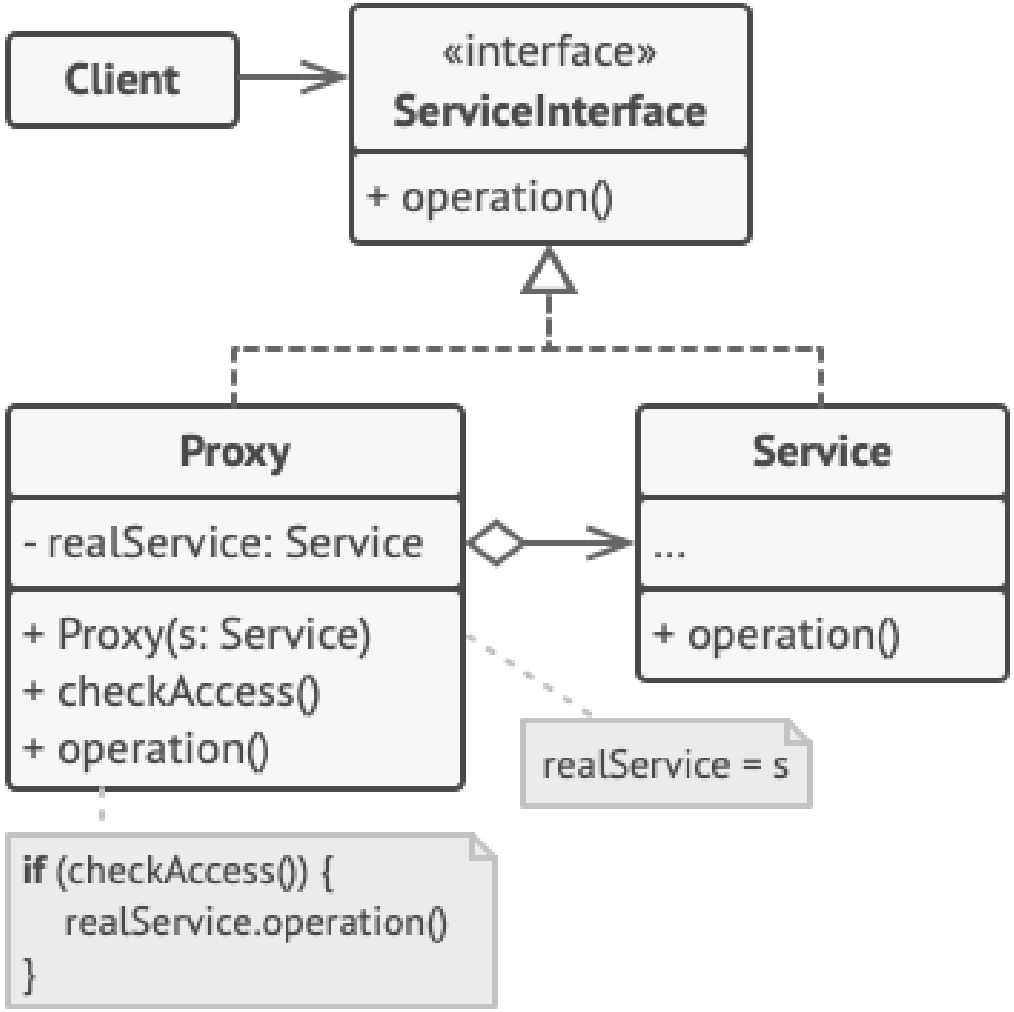


Proxy

- Es utilizado para optimizar recursos.
- El patrón Proxy sugiere que crees una **nueva clase proxy con la misma interfaz que un objeto de servicio original**. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.



Proxy


Client
«interface»
ServiceInterface
+ operation()
Proxy
- realService: Service
+ Proxy(s: Service)
+ checkAccess()
+ operation()
Service
...
+ operation()
realService = s
if (checkAccess()) {
realService.operation()
}

Proxy

- Utilizarlo con **Inicialización diferida** (proxy virtual). Es cuando tienes un objeto de servicio muy pesado que utiliza muchos recursos del sistema al estar siempre funcionando, aunque solo lo necesites de vez en cuando.
- **Control de acceso** (proxy de protección). Es cuando quieres que únicamente clientes específicos sean capaces de utilizar el objeto de servicio, por ejemplo, cuando tus objetos son partes fundamentales de un sistema operativo y los clientes son varias aplicaciones lanzadas (incluyendo maliciosas).
- **Solicitudes de registro** (proxy de registro). Es cuando quieres mantener un historial de solicitudes al objeto de servicio.
- **Referencia inteligente**. Es cuando debes ser capaz de desechar un objeto pesado una vez que no haya clientes que lo utilicen.

Proxy

UCU

- Pros:
  - Principio de abierto/cerrado (OCP). Puedes introducir nuevos proxies sin cambiar el servicio o los clientes.
  - Puedes gestionar el ciclo de vida del objeto de servicio cuando a los clientes no les importa.
- Contras:
  - La respuesta del servicio puede retrasarse

Facade vs Proxy vs Decorator

UCU

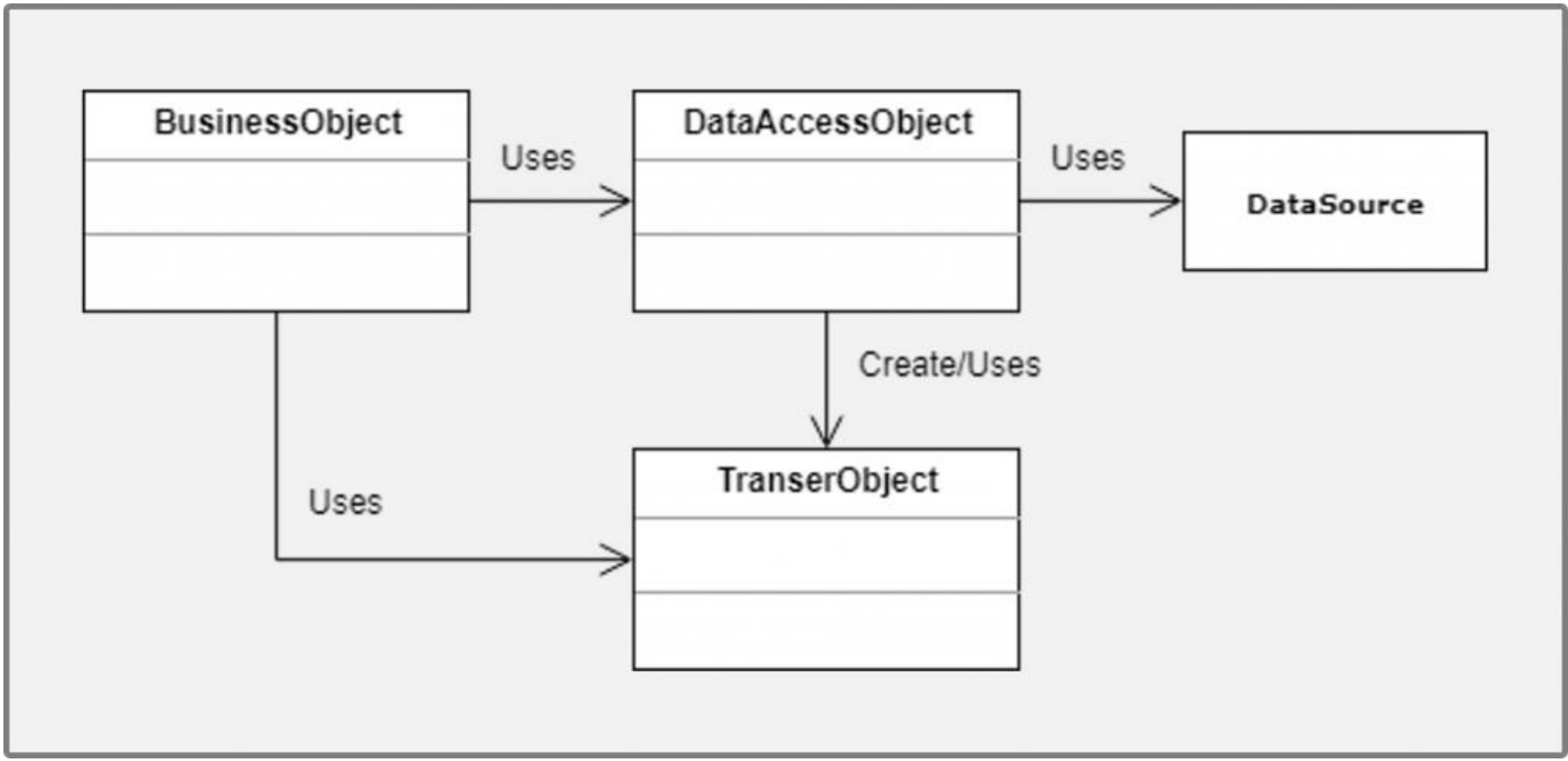
- **Facade**: es como una puerta de entrada simple a un conjunto complicado de funciones. Haces una caja negra para que tus clientes se preocupen menos, es decir, simplificas las interfaces.
- **Proxy**: proporciona la misma interfaz que la clase proxy y puede realizar tareas internas por su cuenta. (Entonces, en lugar de hacer múltiples copias de un objeto pesado “X”, hace copias de un proxy liviano “P” que a su vez administra “X” y traduce sus llamadas según sea necesario). Está resolviendo el problema del cliente de tener que administrar un objeto pesado y/o objeto complejo.
- **Decorator**: se usa para agregar más características a los objetos (generalmente decora objetos dinámicamente en tiempo de ejecución). No oculta ni perjudica las interfaces existentes del objeto, sino que las amplía en tiempo de ejecución.

DAO (Data Access Object)

- Es una abstracción de la persistencia de datos y se considera más cercano al repositorio de almacenamiento, que a menudo está centrado en tablas (base de datos).
- En muchos casos, nuestros DAO coinciden con las tablas de la base de datos, lo que permite una forma más sencilla de enviar/recuperar datos del almacenamiento, ocultando las consultas desagradables.

```
public class UserDaoImpl implements UserDao {
        private final EntityManager entityManager;
        @Override public void create(User user) { entityManager.persist(user); }
        @Override public User read(long id) { return entityManager.find(User.class, id); }
        // ...
}
```

DAO (Data Access Object)



Repository

- Es un mecanismo para encapsular el comportamiento de almacenamiento, recuperación y búsqueda, que emula una colección de objetos.
- En otras palabras, un repositorio también trata con datos y oculta consultas similares a DAO. Sin embargo, se encuentra en un nivel superior, más cerca de la lógica de negocio de una aplicación.
- En consecuencia, Repository puede utilizar DAO para la obtención o persistencia de datos en la base de datos.

```
public class UserRepositoryImpl implements UserRepository {
        private UserDaoImpl userDaoImpl;
        @Override public User get(Long id) { User user =
userDaoImpl.read(id); return user; }
        @Override public void add(User user) { userDaoImpl.create(user); }
        // ...
}
```

Repository vs DAO

UCU

- DAO es una abstracción de la persistencia de datos. Repository es una abstracción de una colección de objetos.
- DAO es un concepto de nivel inferior, más cercano a los sistemas de almacenamiento. Repository es un concepto de nivel superior, más cercano a los objetos de Negocio.
- DAO funciona como una capa de acceso/mapeo de datos, ocultando consultas desagradables. Repository es una capa entre Negocio y capas de acceso a datos, lo que oculta la complejidad de recopilar datos y preparar un objeto de negocio.
- Repository puede usar un DAO para acceder al almacenamiento de datos.

Tarea de Aplicación 4

Patrones de Estructura

Bibliografía

- https://refactoring.guru/design-patterns/structural-patterns
- https://sourcemaking.com/design_patterns/structural_patterns
- https://medium.com/@madhuri.pednekar/importance-of-repository-design-pattern-f26a71f3e26d
- https://www.baeldung.com/java-dao-vs-repository

“Tu actitud, no tu aptitud, determinará tu altitud”