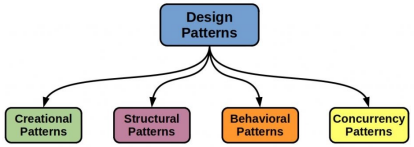
Ingeniería de Software II

Patrones de Diseño

Patrones Estructurales

Docente: Dr. Pedro E. Colla

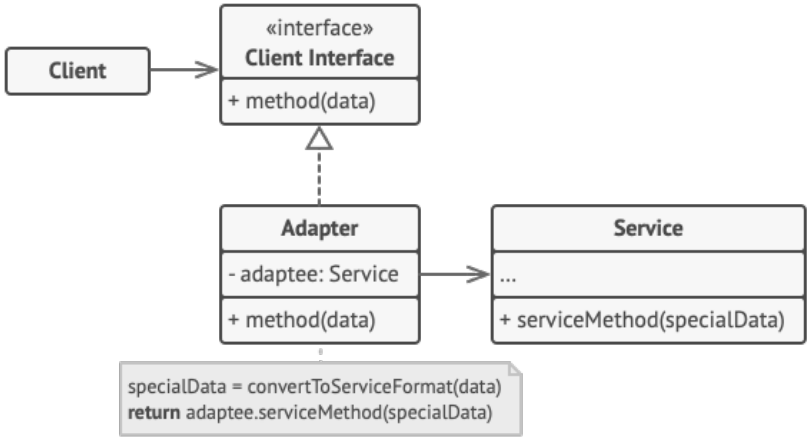
Tipos de patrones



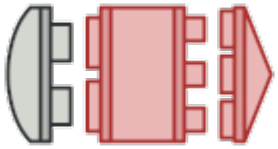
- **Patrones de creación**
  Los patrones de creación abstraen la forma en la que se crean los objetos, permitiendo tratar las clases a crear de forma genérica dejando para más tarde la decisión de qué clases crear o cómo crearlas.
- **Patrones estructurales**
  Tratan de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Lo fundamental son las relaciones de uso entre los objetos, y, éstas están determinadas por las interfaces que soportan los objetos.
- **Patrones de comportamiento**
  Los patrones de comportamiento estudian las relaciones entre llamadas entre los diferentes objetos, normalmente ligados con la dimensión temporal.

Creación	Estructurales	Comportamiento
Singleton Factory Builder Prototype	Adapter Bridge Composite Decorator Fachade Flyweight Proxy	Chain of responsibility Command Interpreter Iterator Mediator Memento Observer … et al.

Patrones de estructurales



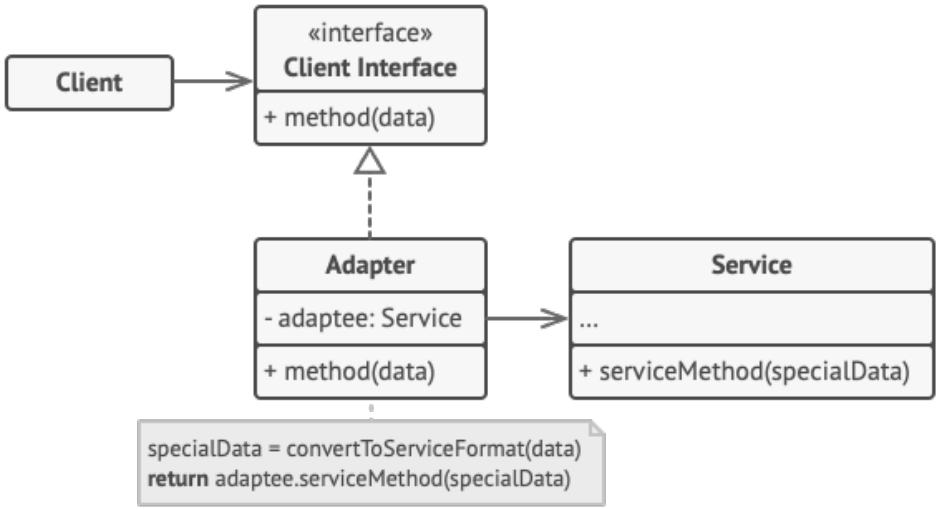
Adapter



Adapter

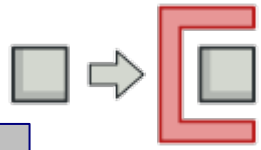
Es un patrón que facilita o permite que objetos con interfaces incompatibles cooperen. También puede usarse para proveer comportamientos consistentes a múltiples interfaces.

Propósito	Problema
• Separa la interfaz y/o gestión de datos de la lógica primaria. • Se puede introducir adaptadores adicionales sin cambiar la interfaz al cliente.	Mayor complejidad.



Python3.7 adapter.py

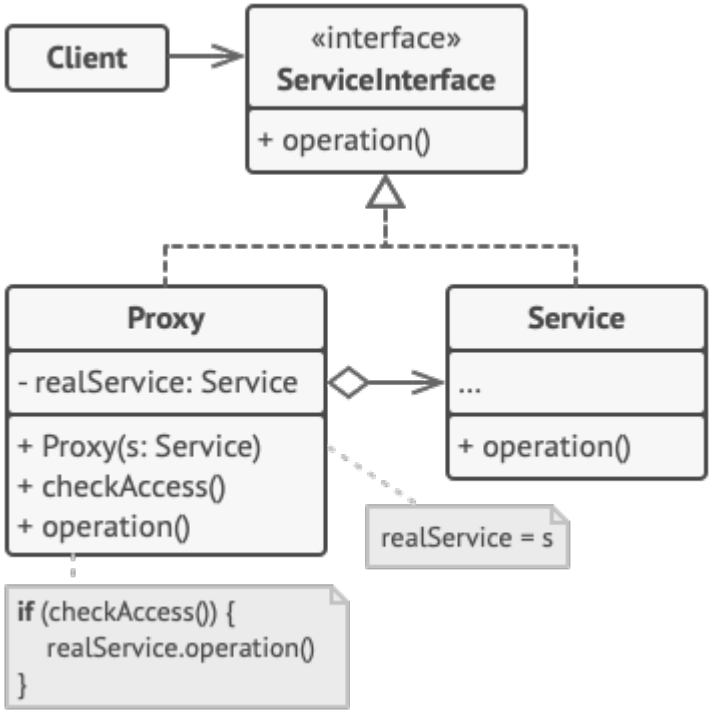
Proxy



Proxy

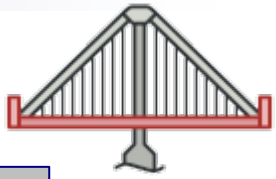
Es un patrón que permite ofrecer un substituto para otro objeto. El proxy controla el acceso para el objeto original permitiendo realizar acciones de preparación y terminación.

Propósito	Problema
• El control de comportamiento y ciclo de vida del objeto de servicio no son visibles al cliente. • Puede gestionar casos donde el servicio no está disponible (o no existe).	Código muy complicado pues deben duplicarse las clases del objeto de servicio. Comportamientos no documentados del objeto de servicios.



Python3.7 proxy.py

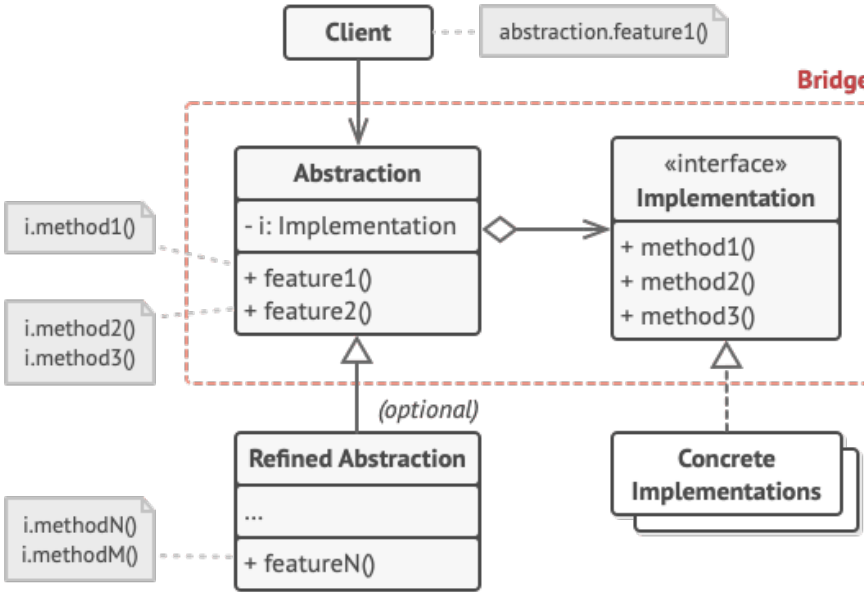
Bridge



Bridge

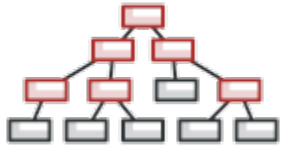
Es un patrón que permite dividir una clase muy grande en conjuntos de clases relacionadas entre si de manera que se utilicen jerarquías diferentes para la abstracción y la implementación.

Propósito	Problema
• Se pueden crear clases independientes de la plataforma. • Se divide el alto y bajo nivel de implementación. • Se pueden agregar abstracciones e implementaciones independientes unas de otras.	Mayor complejidad.



Python3.7 bridge.py

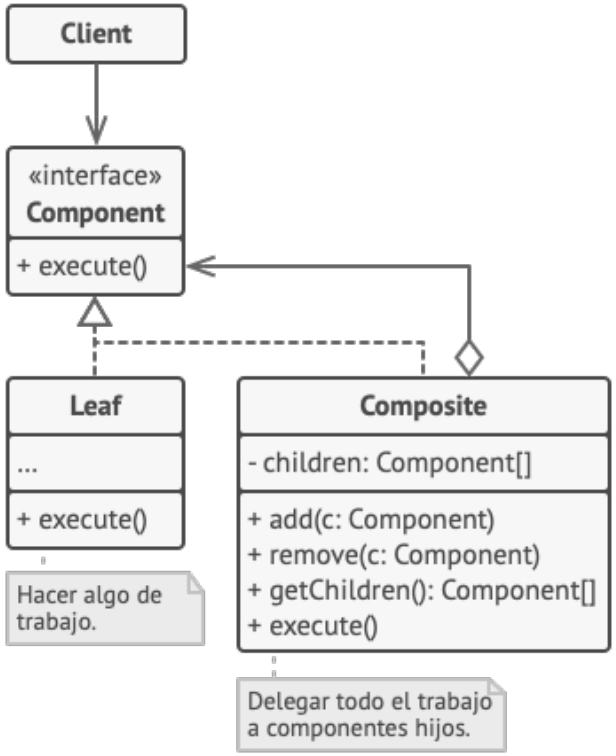
Composite



Composite

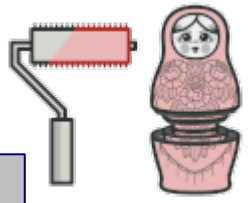
Es un patrón que permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

Propósito	Problema
• Se puede trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.	Resulta dificil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.



Python3.7 composite.py

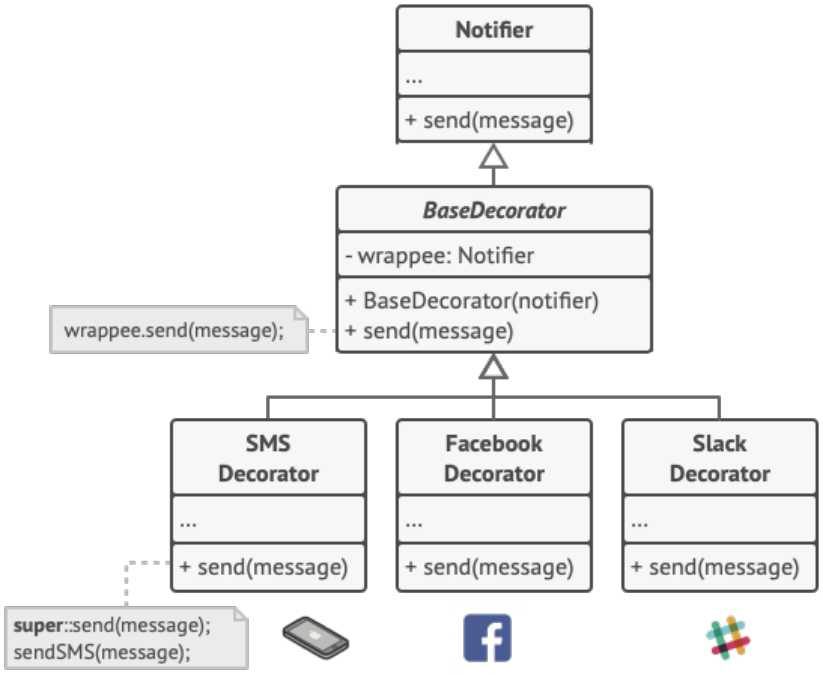
Decorator



Decorator

Es un patrón que permite adjuntar comportamientos nuevos a objetos mediante el recurso de colocarlos dentro de "envoltorios" (*wrapper*) que gestionan los comportamientos.

Propósito	Problema
• Extender el comportamiento sin necesidad de subclases especiales diferenciadas. • Agregar o quitar responsabilidades en run-time. • Se pueden combinar comportamientos diferentes en "wrappers"	Dependencias en como son llamados los diferentes wrappers (el orden tiene semántica).



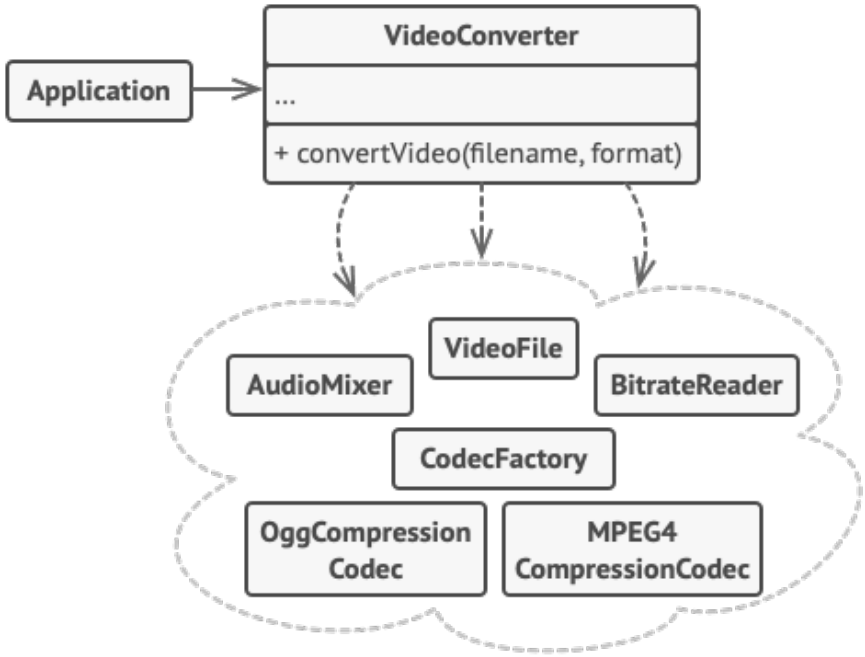
Python3.7 decorator.py

Facade (Façade)

Facade

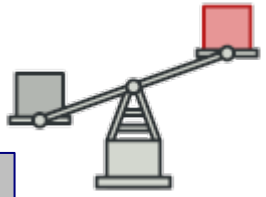
Es un patrón estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

Propósito	Problema
• Aislar la complejidad en el código de subsistemas. • Actúa como un coordinador o "*dispatcher*" de ser necesario.	Puede deteriorar métricas de acoplamiento.



Python3.7 facade.py

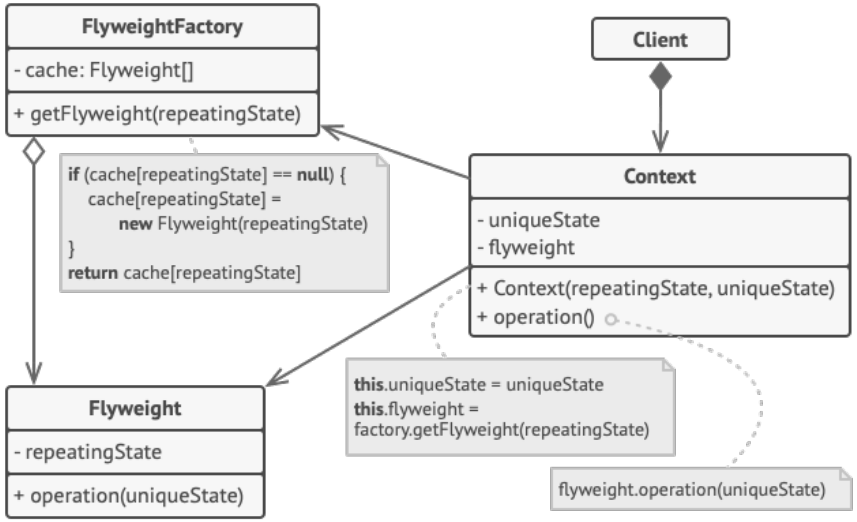
Flyweight



Flyweight

Es un patrón que permite reducir requerimientos de memoria colocando partes comunes de estado, inicialización o referencia entre múltiples objetos en uno solo.

Propósito	Problema
• Si hay muchos objetos similares en ejecución se protegen significativamente los recursos de procesamiento.	Puede deteriorar métricas de acoplamiento. Puede representar un equilibrio entre CPU y memoria (costo de instanciación y mantenimiento vs. costo de almacenamiento)



Python3.7 flyweight.py

Taller

- Desarrolle una clase que permita acceder a una vieja clase requiriendo dos parámetros con solo uno. Use como plantilla el patrón llamado "*adapter.py*". (IS2_taller_adfix.py)
- Desarrolle una clase que permita operar como proxy a una validación de seguridad agregándole una capa de encriptación adicional. Use como plantilla el patrón llamado "*proxy.py*". (IS2_taller_door.py)
- Desarrolle una clase que permita operar como bridge de forma que se invoque en run-time diferentes métodos de fabricación para un objeto cuyas dimensiones sean definidas en forma común a ambos. Use como plantilla el patrón llamado "*bridge.py*". (*¿cómo cambiaría el método de producción en "run-time"?*) (IS2_taller_cuboid.py)
- Generar una clase que implemente una organización en un ambiente de desarrollo con dos niveles de reporte. Use como plantilla el patrón llamado "*composite.py*". (*¿Agregar otra gerencia o removerla?)* (IS2_taller_organizacion.py)
- Genere un ejemplo de decorator para agregar en forma anidada atributos de presentación HTML a un texto dado. Use como plantilla el patrón llamado "decorator.py" (IS2_taller_fonts.py)

Referencias en https://refactoring.guru/es/design-patterns/structural-patterns

¿Preguntas?

Bibliografía

- Architecture Patterns with Python – Percival & Gregory – O'Reilly
- Design Patterns – Lasater
- *https://refactoring.guru/es/design-patterns/structural-patterns*