Universidad Autónoma de Entre Ríos

FCyT FACULTAD DE CIENCIA Y TECNOLOGÍA UADER
SEDE CONCEPCIÓN DEL URUGUAY

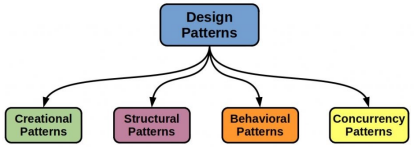
Ingeniería de Software II

Patrones de Diseño

Patrones de Comportamiento

Docente: Dr. Pedro E. Colla

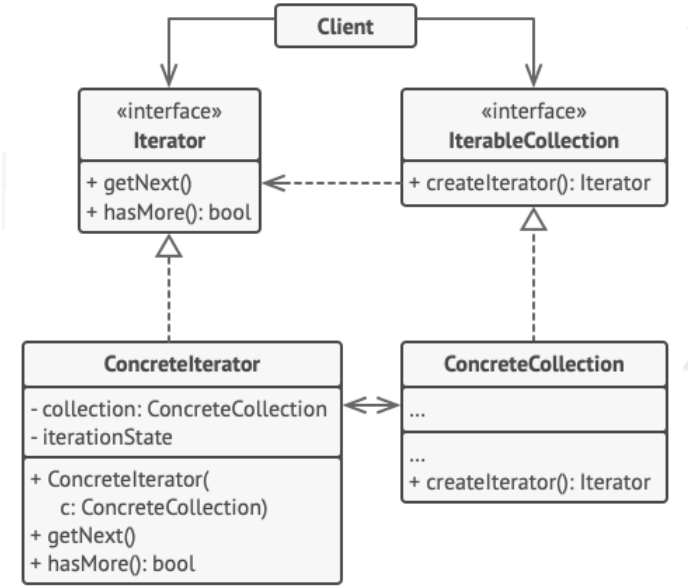
Tipos de patrones



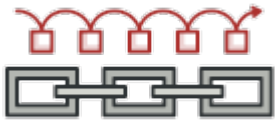
- **Patrones de creación**
  Los patrones de creación abstraen la forma en la que se crean los objetos, permitiendo tratar las clases a crear de forma genérica dejando para más tarde la decisión de qué clases crear o cómo crearlas.
- **Patrones estructurales**
  Tratan de conseguir que cambios en los 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 de las 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 comportamiento



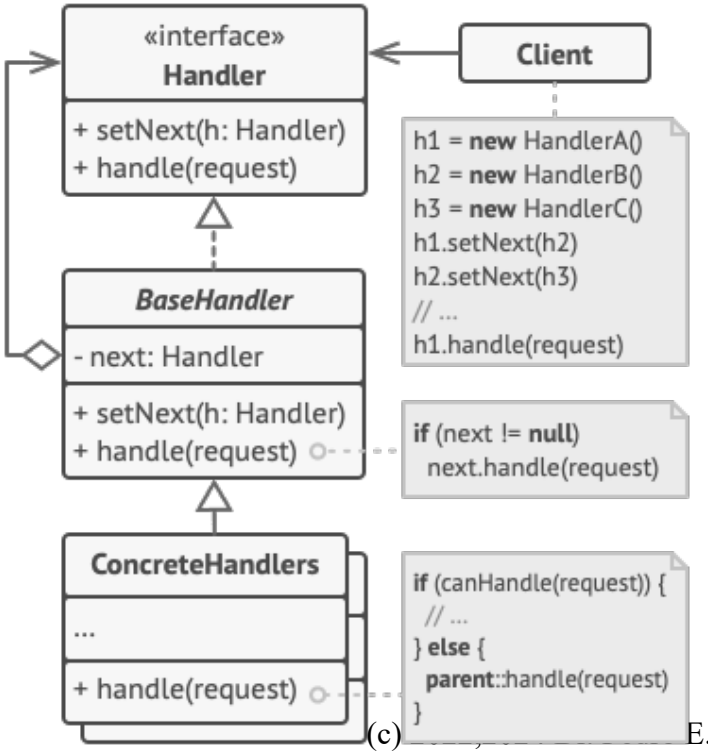
Chain of responsibility



Chain of responsibility

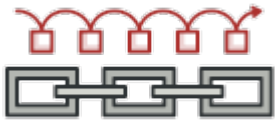
Es un patrón de comportamiento que permite pasar requerimientos entre una cadena de gestores. Cada uno al recibir el requerimiento decide si procesarlo, y consumirlo o dejarlo pasar al terminar.

Propósito	Problema
• Permite gestión ordenada de mensajes • Se puede introducir controladores adicionales sin cambiar la interfaz al cliente.	Mayor complejidad. Costo de construcción. Persistencia. Tiempo de procesamiento



Python3.7 chain.py

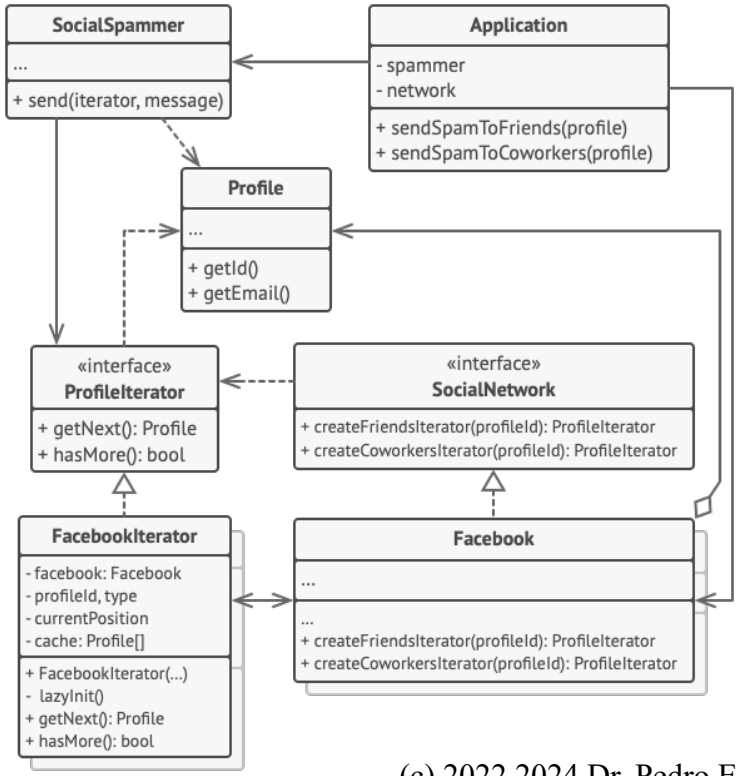
Iterator



Iterator

Es un patrón de comportamiento que permite recorrer una colección de objetos sin exponer su representación o implementación (lista, stack, árbol, matriz, etc.)

Propósito	Problema
• Permite procesar todos los componentes. • El código para iterar es genérico.	Mayor complejidad. Menos eficiente que un iterador especializado.



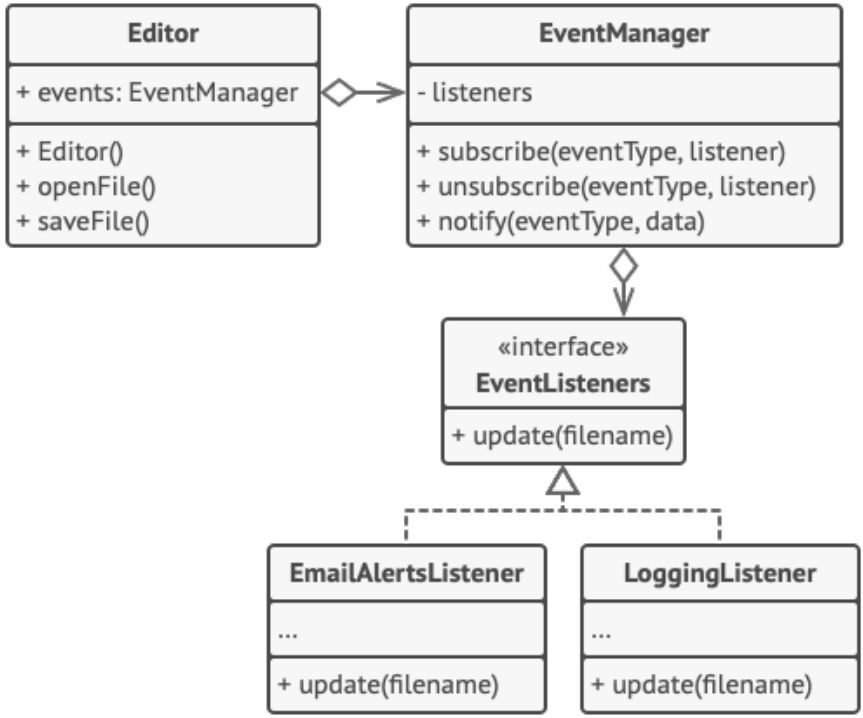
Python3.7 iterator.py

Observer

Observer

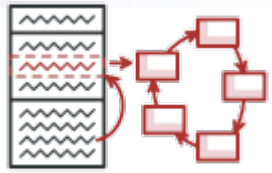
Es un patrón de comportamiento que permite definir un mecanismo de subscripciòn mediante el cual notificar a múltiples objetos sobre algún evento de interés.

Propósito	Problema
• Permite compartir eventos en forma flexible. • Implementa la transferencia transparente de información entre receptores no similares.	Mayor complejidad. Dificil establecer un orden de aviso a subscriptores



Python3.7 observer.py

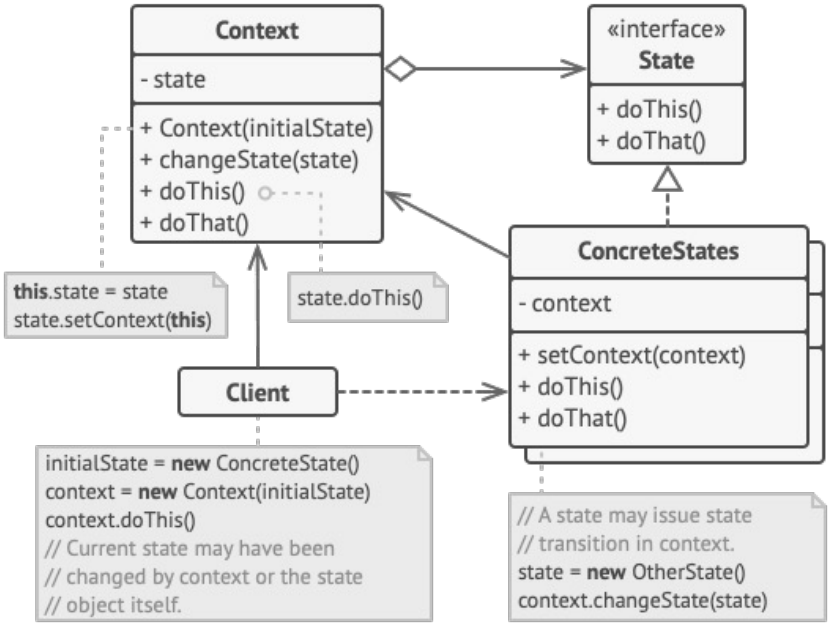
State



State

Es un patrón de comportamiento que permite que un objeto cambie su comportamiento de acuerdo a cierto estado interno, de tal manera que luzca como una clase distinta.

Propósito	Problema
• Reduce acoplamiento. • Alterar la red de estados de una FSM sin modificar los existentes. • Reducir condicionales (McCabe)	Mayor complejidad. Puede ser excesivo para máquinas de estado simples.



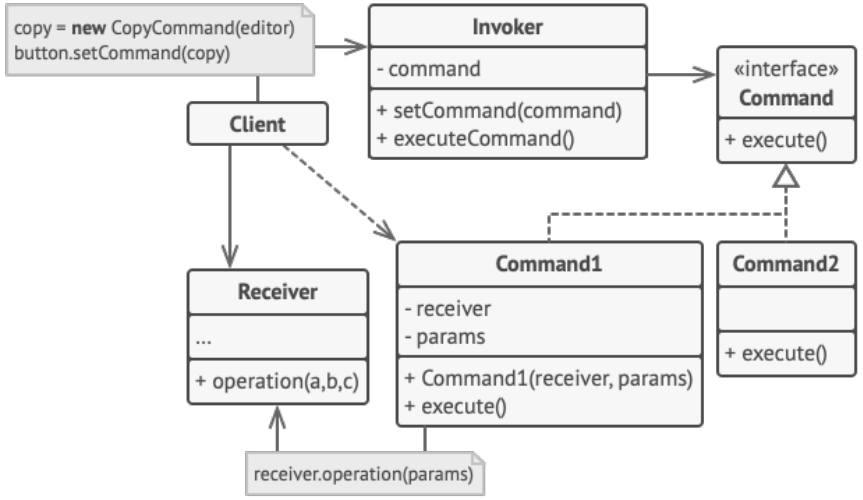
Python3.7 state.py

Command

Command

Es un patrón de comportamiento que permite convertir una solicitud en un objeto independiente que contiene toda la información sobre la solicitud.

Propósito	Problema
• Gestionar la aplicación, retraso o priorización del comando junto a su contexto.	Mayor complejidad. Debe salvarse no solo el request, sino su método de ejecución y su contexto.



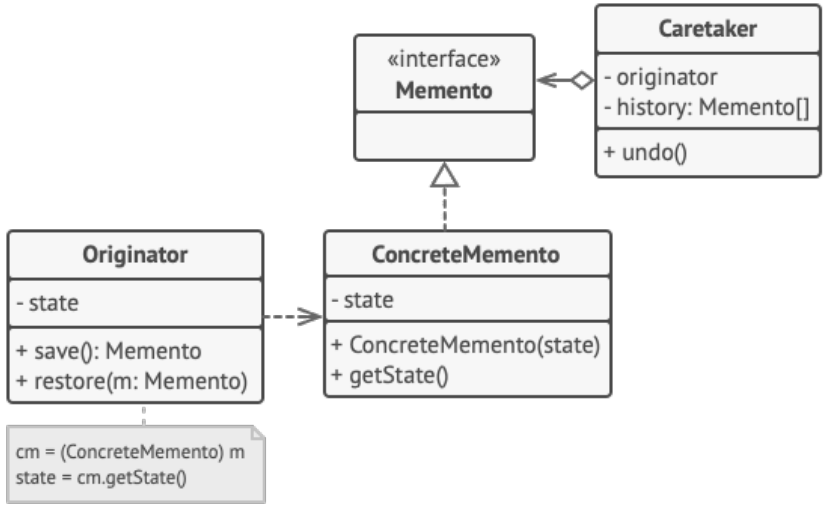
Python3.7 command.py

Memento

Memento

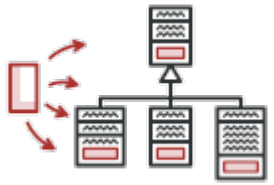
Es un patrón de comportamiento que permite almacenar información relevante sobre un objeto independientemente de su implementación y recuperar un estado anterior.

Propósito	Problema
• Permite guardar información sin revelar encapsulamiento o estructura (blob). • Gestionar historia de estados (snapshot, checkpoint).	Mayor complejidad. Garbage collection. Mayor consumo de recursos.



Python3.7 memento.py

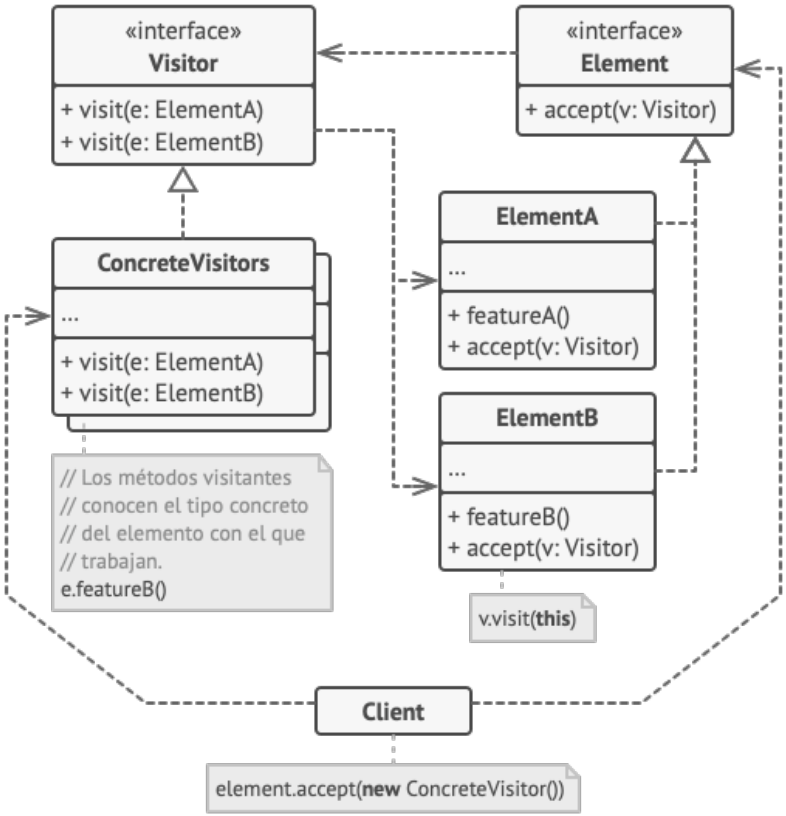
Visitor



Visitor

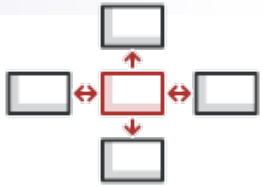
Es un patrón de comportamiento que permite separar algoritmos de los objetos sobre los que operan.

Propósito	Problema
• Un único observante puede conocer información única sobre una jerarquía de objetos.	Mayor complejidad.



Python3.7 visitor.py

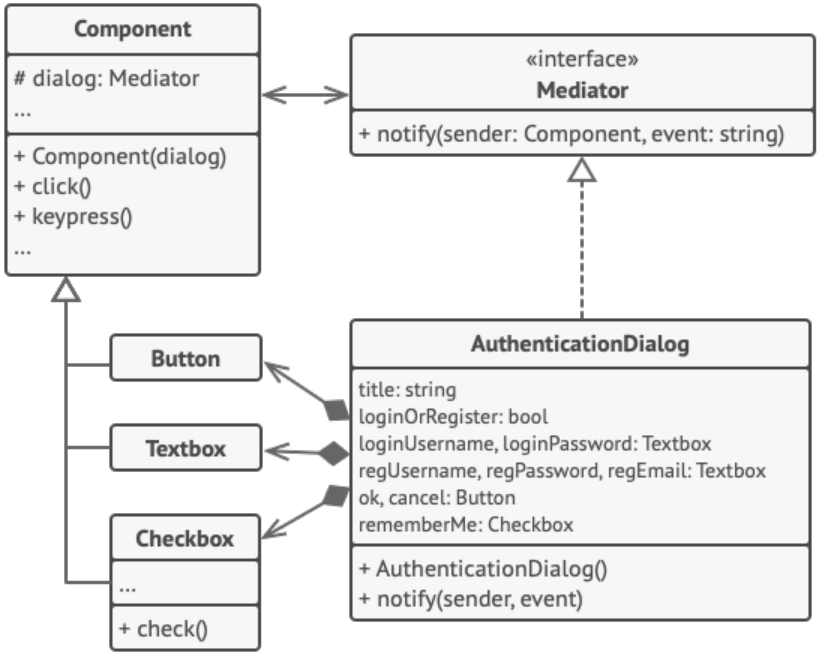
Mediator



Mediator

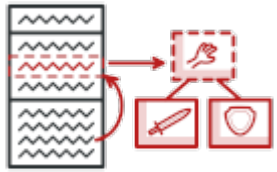
Es un patrón de comportamiento que gestionar la interacción entre objetos para impedir su intercambio directo y despachar requerimientos entre los mismos.

Propósito	Problema
• Reduce acoplamiento. • Permite diferir la definición de relaciones particulares entre objetos.	Mayor complejidad. Punto simple de falla. Gestión de estado y persistencia



Python3.7 mediator.py

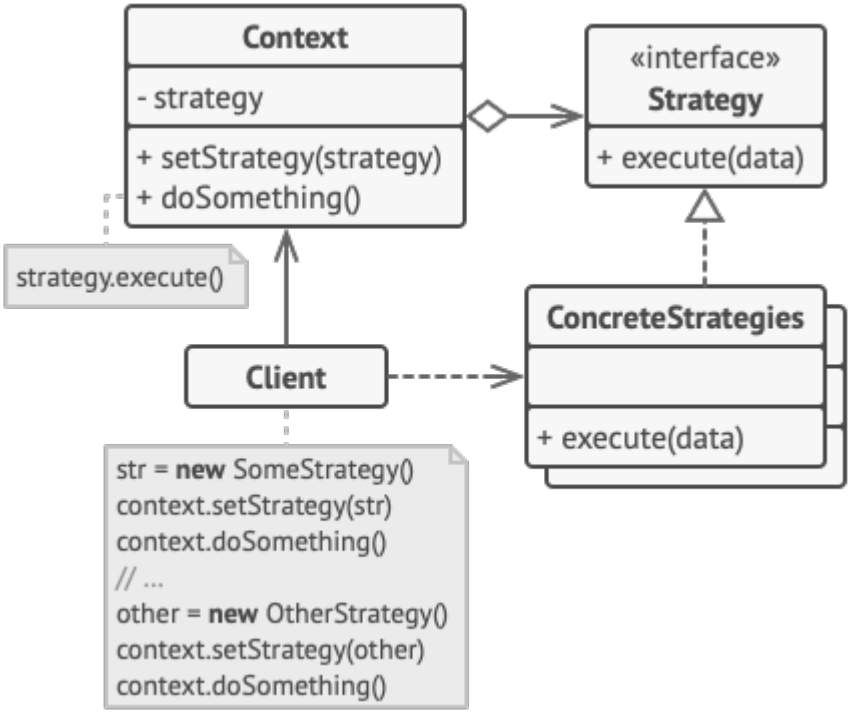
Strategy



Strategy

Es un patrón de comportamiento que permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

Propósito	Problema
• Selección dinámica de algoritmos en tiempo de ejecución. • Detalles del algoritmo diferentes de su utilización • Herencia por composición	Mayor complejidad. Determinar la forma mas eficiente de elegir la estrategia



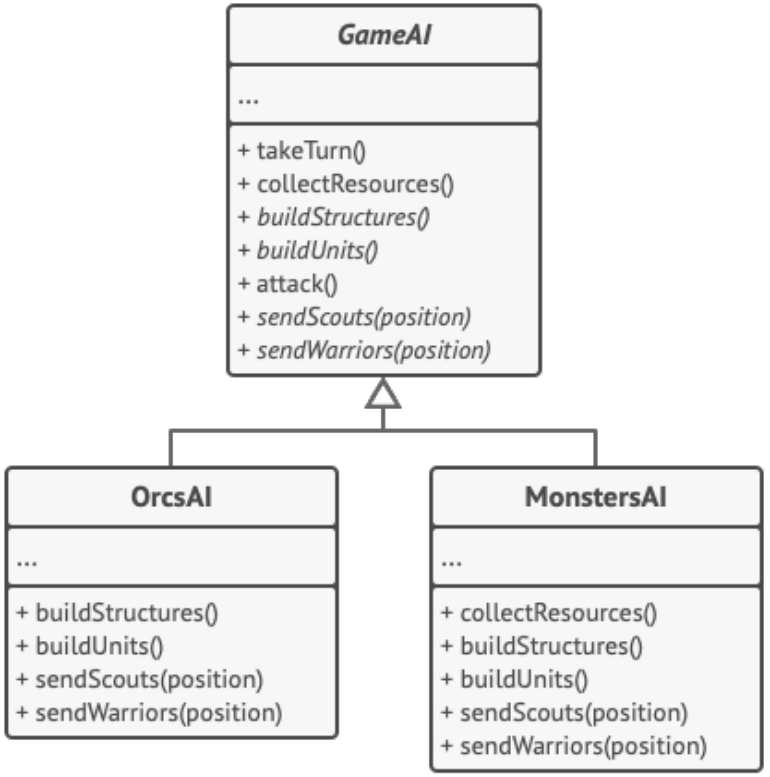
Python3.7 strategy.py

Template

Template

Es un patrón de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura..

Propósito	Problema
• Permitir a los clientes que sobrescriban tan solo ciertas partes de un algoritmo grande, para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo.	Mayor complejidad. Variante sofisticada de herencia.



Python3.7 template.py

Taller

- Desarrolle una clase que permita imprimir un archivo utilizando el handler de formato correcto (*pdf,txt,doc*). Use como plantilla el patrón "chain.py". (IS2_taller_formato.py)
- Genere una lista de ítems e implemente una clase para recorrerlos. Use como plantilla el patrón "iterator.py". (IS2_taller_patron.py)
- Genere una clase que establezca 3 clases observadoras, respectivamente formatos Hex, Decimal y Octal. Subscriba esas clases a una clase donde hay un valor sujeto a observación. Cambie los valores y observe el funcionamiento del mecanismo de subscripción. Use como plantilla el patrón "observer.py" (IS2_taller_visores.py)
- Genere una clase para gestionar la sintonía de una radio AM/FM según el modo en que se encuentre. Use como plantilla el patrón "state.py" (IS2_taller_scanner.py)
- Implemente un procesador de comandos según la plantilla "command.py" (IS2_taller_cmd.py)
- Implemente una función de "undo" según la plantilla "memento.py" (IS2_taller_memory.py)
- Implemente una función asignación de profesores y alumnos a cursos plantilla "visitor.py" (IS2_taller_curso.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/design-patterns/python*