UNIVERSIDAD NACIONAL GUILLERMO BROWN

Licenciatura en

CIENCIA DE DATOS

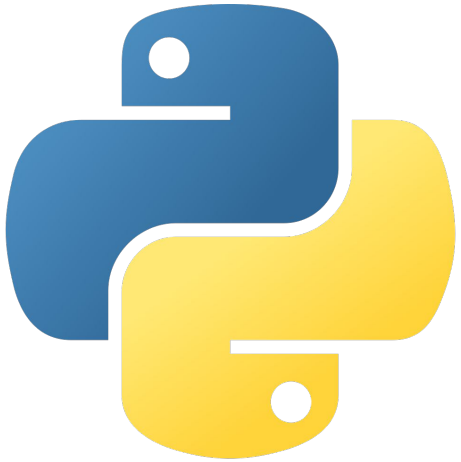
Tecnicatura en

PROGRAMACIÓN

PATRONES DE DISEÑO

. Modelos abstractos que describe cómo resolver comunes problemas que surgen en el desarrollo de software.
. Proporcionan un “lenguaje común” entre desarrolladores.
. Soluciones generalizadas a problemas frecuentes.
. Simplifican la arquitectura, facilitar su mantenimiento y mejorar su escalabilidad.
. Los patrones de diseño pueden aplicarse en la interfaz de usuario, la lógica de negocio y el acceso a datos.
. Son “soluciones probadas”
. Son muchos y algunos no tienen aplicación DIRECTA en Python.

QUE TENEMOS QUE SABER ANTES DE VER PD



python™

. **Python** no soporta *nativamente* **clases abstractas** (no se pueden instanciar)

. **Python** no soporta *nativamente* **Interfaces.** Clases cuyos métodos no tienen una implementación concreta (pass)

Son como un ***contrato*** => voy a recibir dos números enteros y voy a devolver un numero entero)

. **Python** usa "mixins" Clases con métodos no implementados que se pueden heredar y que permiten centralizar las operaciones comunes.

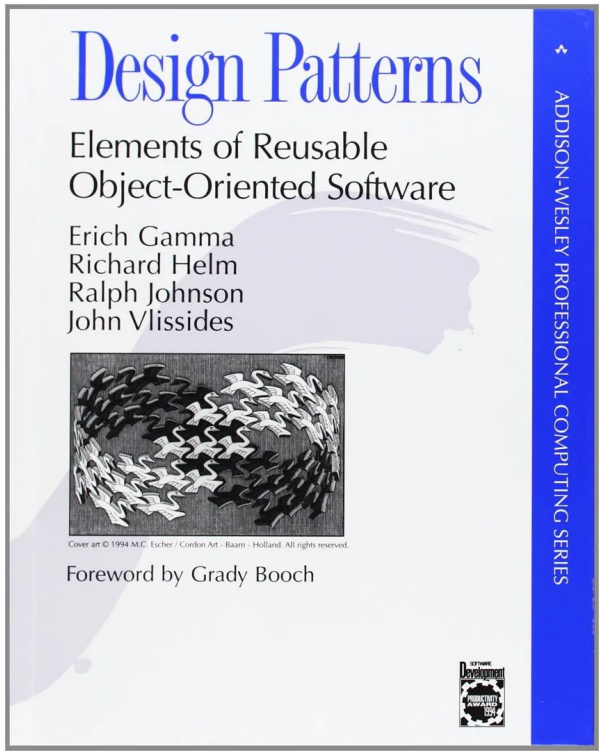
CUANDO USARLOS

. Se necesita resolver comunes problemas que surgen en el desarrollo de software.
. Se desea mejorar la estructura y organización de la aplicación.
. Se necesita reducir la duplicación de código en la aplicación.
. Se desea separar responsabilidades claras en la aplicación.
. Se necesita agregar funcionalidad a una clase sin modificarla directamente.
. Se desea crear un patrón específico para un problema recurrente en la aplicación.

En definitiva, el uso de patrones de diseño se recomienda cuando se necesita mejorar la estructura y organización de la aplicación, reducir la duplicación de código, separar responsabilidades claras, agregar funcionalidad a una clase sin modificarla directamente y resolver problemas específicos en la aplicación. Además en el desarrollo moderno se emplean "frameworks" que ya implementan los patrones de diseño adecuados para la situación que que debemos adaptar de ese entorno.

CUANTOS SON

Originalmente se recopilaron 22 patrones de diseño en el libro de “La pandilla de los 4” *Design Patterns*
A esos se los denomina patrones clásicos.
Pero con el paso del tiempo y con la evolución natural del diseño de software se han adicionado algunos y se han eliminado o agrupado otros.
Según algunos autores ya hay 33 aunque hay quienes piensan que de esos habría que eliminar varios porque son considerados más dañinos que utiles (anti-patrones)
Vamos a ver algunos pero se puede encontrar una muy buena referencia y casos de uso de los 22 clasicos en la pagina
https://refactoring.guru/es/design-patterns/catalog
Que está en castellano y tiene ejemplos en python.



TRES GRUPOS

Creacionales: Para la creación de objetos.

Estructurales: Se refieren a las operaciones entre objetos

Comportamiento: Gestionan el funcionamiento de los objetos

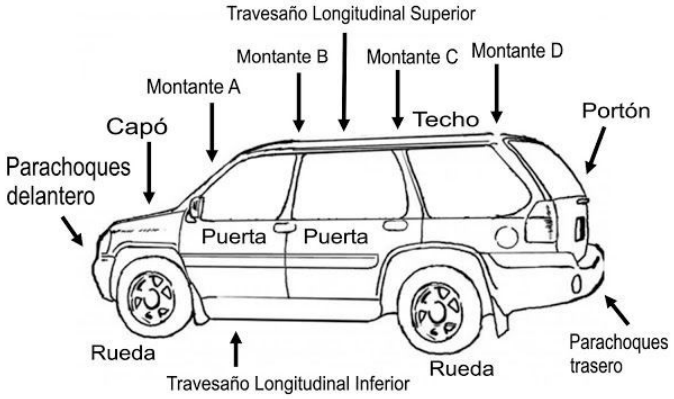
CREACIONALES - BUILDER

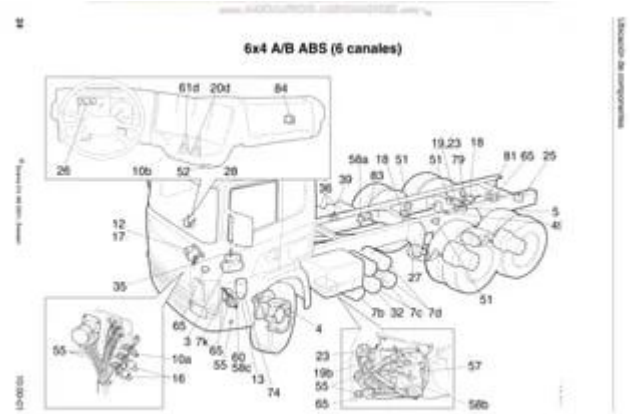
Este patrón nos sirve para "desmembrar" la construcción de objetos muy complejos en sub-clases mas simples.

A veces se necesitan enormes cantidades de argumentos para el __*init*__ de una clase. Y en ciertos casos no es necesario asignarles valor a todos.

Ejemplo clase *Vehiculo*. (karting- auto base- tope de gama)

Ademas, cada subclase miembro de la clase con la que trabajamos puede ser asignada a un encargado diferente. Con lo cual se reduce la tarea de los desarrolladores.





CREACIONALES - FACTORY

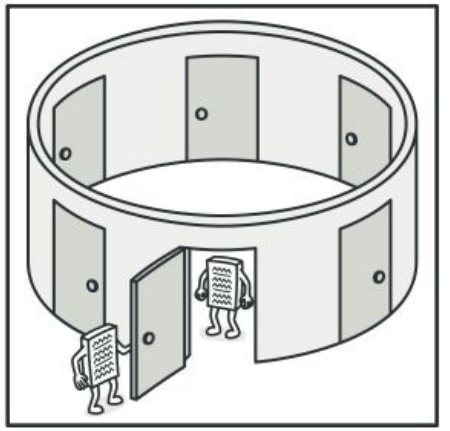
El patrón **factory** consiste en tener una clase (generalmente abstracta) que sirve como molde genérico para crear otras clases, similares o relacionadas con la clase *factoría*.

El ejemplo clásico es el caso de una pastelería que nos crea tortas a pedido del cliente.

Siguen siendo tortas, producidas por la misma factoría pero tienen características distintas (sabor, decoracion, forma, etc)

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.
La clase singleton cuando se instancia verifica que no haya un objeto ya creado. Si no existe, crea uno nuevo. Pero si ya existe lo devuelve como resultado.
Se usa mucho en conexiones a bases de datos u otras fuentes de datos, sobre todo en las que cobran por cada conexión.



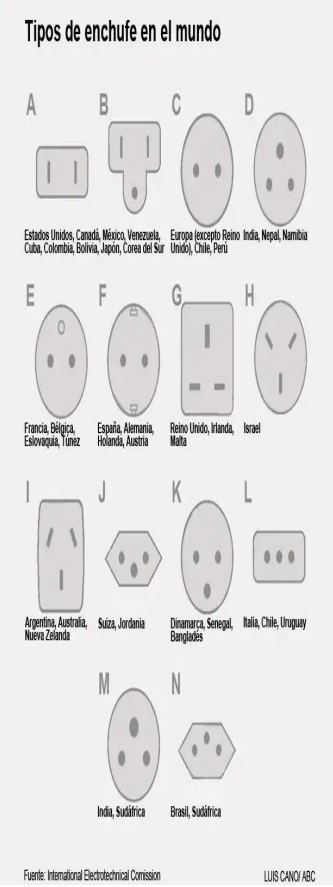
INFRAESTRUCTURA - DECORADOR

Este patrón de diseño ya lo hemos aprendido y empleado.
Se basa en emplear la funcionalidad ya existente en los métodos de una clase pero envolviendolos con operaciones que se realizan antes y después de la ejecución del método.
Este *wrapper* agrega funcionalidades alo existente o incluso, puede evitar la ejecucion del codigo envuelto.

INFRAESTRUCTURA - FLYWEIGHT

Si tenemos varias clases distintas que comparten partes de código similares, podemos extraer todos esos métodos y generalizarse en otra clase.
A través de atributos específicos en las clases originales podremos cambiar, permitir o evitar la ejecución de los métodos que separamos en la clase **flyweight** (peso mosca)
Pensemos en un juego donde tengamos muchos objetos representando jugadores, todas las características comunes pueden ser extraídas y unificadas en una clase. De esta manera ahorraremos memoria y ganaremos en velocidad de ejecución

INFRAESTRUCTURA - ADAPTER



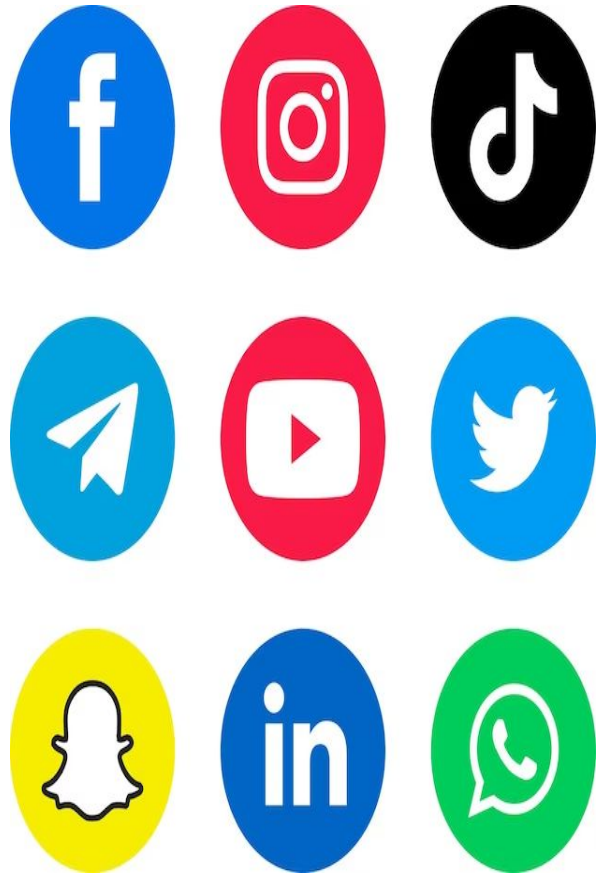
Si viajamos y queremos usar el cargador del celular tenemos que tener en cuenta que el tipo de enchufe que usamos en Argentina es bastante “particular” y que deberemos usar adaptadores para otros tipo de enchufe de pared.

El patrón **Adapter** en programación es exactamente eso: un adaptador que permite que dos clases con interfaces incompatibles trabajen juntas. Actúa como un puente, traduciendo las solicitudes de un objeto a un formato que el otro objeto pueda entender.

Sirve mucho cuando tenemos que:
. Integrar código existente: Cuando tenemos una biblioteca o clase antigua con una interfaz que no encaja en el nuevo diseño, pero no podemos (o no queremos) modificarla directamente.
. Usar bibliotecas de terceros: Si una biblioteca ofrece funcionalidades que necesitamos, pero su interfaz no es compatible con tu código, podemos usar un **Adapter** para "traducir" las llamadas.
. Ocultar la complejidad: En casos donde la lógica de adaptación es compleja, el patrón Adapter la encapsula, manteniendo el código principal más limpio y legible.

COMPORTAMIENTO - OBSERVER

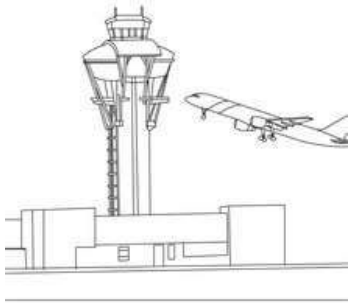
Cuando usamos una aplicación de redes sociales estamos usando el patrón **observer**.
Un objeto (la base de datos de la Red Social) recibe cambios continuamente. E informa de ciertos cambios a los suscriptores en función de a que estén suscriptos.



Por eso recibimos las notificaciones de las cuentas a las que seguimos y no TODAS las notificaciones que pueda haber.

En el patrón **observer** la clase que tiene las modificaciones es la que informa a sus suscriptores de los cambios recibidos.

COMPORTAMIENTO - MEDIATOR



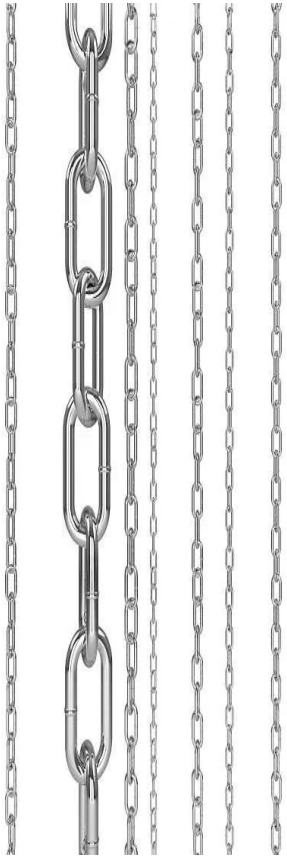
Este patrón lo vimos cuando leímos sobre Algoritmos de enjambre versión predador/presa.

También lo vemos en un asado entre amigos o en familia: Siempre tenemos un asador que gestiona la cocción de la carne (y otras comidas). Y que se encarga de recibir que corte quiere cada uno y cual es su punto.

Otro ejemplo del patrón **mediator/controller** es el de una torre de control de un aeropuerto. Si todos los aviones se tuvieran que organizar entre ellos para aterrizar o despegar no alcanzarian los canales de radio disponibles para mantener a todos los aviones organizados.

Por eso se centraliza el control de otros agentes independientes.

COMPORTAMIENTO - CADENA DE RESPONSABILIDAD



Cuando realizamos una compra en un e-commerce nuestro objeto Pedido es procesado por una serie de controles.
Primero se verifica que haya stock, después que se haya incorporado al carrito.
A continuación se verifica que la dirección de entrega sea válida, se elige el método de pago y, según cual sea este, si hay saldo, si se puede imputar a una tarjeta o banco, etc
Por último ese pedido (junto a otros) se procesa por otras clases que organizan la cadena del envío.
Así es como se emplea este patrón de **cadena de responsabilidades**.
Y su uso va de algo de tanto nivel como una venta hasta al procesamiento de un paquete TCP/IP en una red.

IMPLEMENTACIONES

Hay patrones de diseño tan útiles que los empleamos sin saber que son patrones.
El caso más emblemático es el del **Decorador**, pero también cuando usamos type para crear clases a pedido, estamos de una forma u otra empleando el patrón **Factory** sin saberlo.
Eso mismo nos va a pasar cuando trabajemos en grandes proyectos o utilicemos *frameworks* de desarrollo.
Todos esos proyectos se basan en varios patrones de diseño que ya los tienen incorporados.
Por eso conocer patrones es una buena herramienta para el programador. Reconocer que estamos usando nos va a permitir comunicarnos mejor con otros programadores y planificar las mejoras, correcciones de bugs o reescrituras de código con un mejor conocimiento del funcionamiento base sobre el que estemos trabajando.