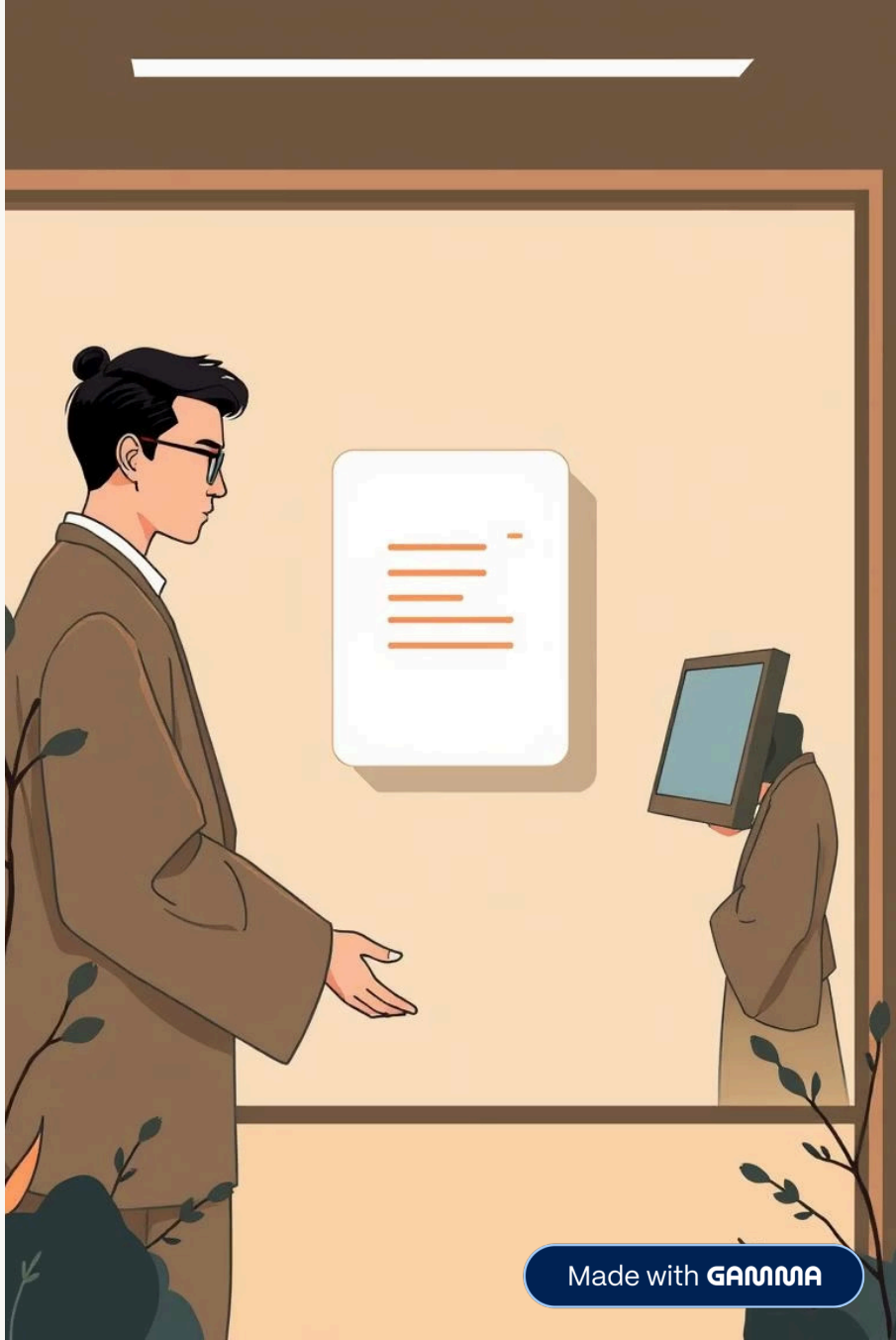
Patrón de Diseño Proxy: Controlando el acceso a objetos

El patrón Proxy, un pilar estructural en el diseño de software, actúa como un sustituto o intermediario elegante para otro objeto. Su función principal es controlar el acceso a este objeto real, ofreciendo una capa de abstracción que permite incorporar funcionalidades adicionales sin alterar la lógica interna del objeto subyacente.

La clave de su versatilidad reside en que tanto el proxy como el objeto real implementan la misma interfaz. Esta característica crucial asegura que, desde la perspectiva del cliente, ambos son intercambiables, facilitando la adición de mejoras como caché o mecanismos de seguridad de forma transparente y eficiente.



¿Por qué usar Proxy? Casos y beneficios clave

Carga Diferida

Evita la instanciación de objetos complejos o costosos hasta el momento preciso en que se necesitan, optimizando el rendimiento.

Control de Acceso (Protección)

Regula y verifica los permisos antes de permitir la operación sobre el objeto real, esencial para la seguridad.

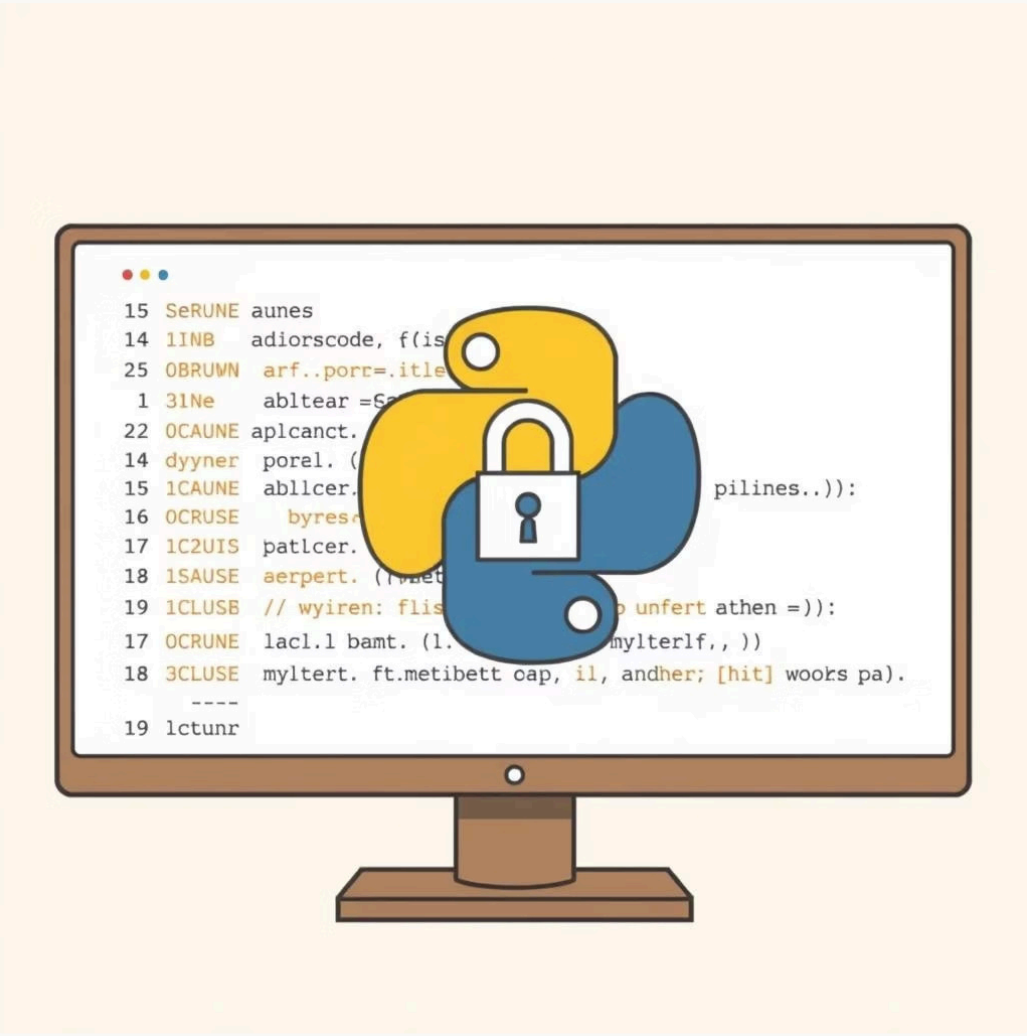
Funcionalidades Añadidas

Integra lógicas como logging, caché, o sincronización de hilos de manera transparente para el cliente.

Un ejemplo cotidiano es la tarjeta de crédito: actúa como un proxy de tu cuenta bancaria. Te permite realizar transacciones sin tener que manejar físicamente el dinero o interactuar directamente con el banco en cada compra, añadiendo seguridad y comodidad.

Ejemplo en Python: Proxy con control de acceso y logging

Imaginemos un sistema donde solo usuarios autenticados pueden acceder a una base de datos sensible. Implementaremos un proxy que verifique las credenciales antes de permitir cualquier operación.



```
class IDatosSensibles:
    def obtener_datos(self):
        pass

class BaseDeDatos(IDatosSensibles):
    def obtener_datos(self):
        return "Datos muy confidenciales de la BBDD"

class ProxyDeAcceso(IDatosSensibles):
    def __init__(self, bbdd, usuario, contrasena):
        self._bbdd = bbdd
        self._usuario = usuario
        self._contrasena = contrasena

    def obtener_datos(self):
        if self._autenticar():
            print(f"[{self._usuario}] Acceso autorizado.
Obteniendo datos...")
            return self._bbdd.obtener_datos()
        else:
            print(f"[{self._usuario}] Acceso denegado.
Credenciales incorrectas.")
            return "Acceso denegado"

    def _autenticar(self):
        # Lógica de autenticación simplificada
        return self._usuario == "admin" and self._contrasena
== "secreto"

# Uso
bbdd_real = BaseDeDatos()

# Acceso autorizado
proxy_autorizado = ProxyDeAcceso(bbdd_real, "admin",
"secreto")
print(proxy_autorizado.obtener_datos())

# Acceso denegado
proxy_denegado = ProxyDeAcceso(bbdd_real, "user",
"12345")
print(proxy_denegado.obtener_datos())
```

Este ejemplo muestra cómo el `ProxyDeAcceso` envuelve al objeto `BaseDeDatos`. Antes de ejecutar `obtener_datos`, el proxy realiza una autenticación y registra el intento de acceso. Si las credenciales son válidas, delega la llamada al objeto real; de lo contrario, niega el acceso.

Esto permite añadir seguridad (autenticación) y logging sin modificar la clase `BaseDeDatos`, cumpliendo con el principio de abierto/cerrado.