CHAPTER 8

Diseño a Nivel de Código

Si bien las propiedades y principios de diseño pueden aplicarse en distintos niveles de abstracción, en este capitulo nos enfocaremos en aquellos orientados a la escritura de código de calidad, legible y mantenible en el tiempo.

8.1 Integridad Conceptual

La **integridad conceptual** sostiene que un software no debe ser simplemente un conjunto de funcionalidades aisladas, carente de coherencia o consistencia. En cambio, debe mantener una integridad conceptual que facilite tanto su uso como su comprensión.

En el código fuente, la falta de integridad conceptual puede manifestarse en aspectos como el uso inconsistente de nombres a lo largo del sistema. Por ejemplo, variables escritas con mayúscula inicial como `fullName` pueden coexistir con otras que emplean guiones bajos, como `full_name`, o con nombres de métodos y clases que no siguen un mismo criterio de nomenclatura. Estos ejemplos muestran la ausencia de una estandarización, lo cual da lugar a una carencia de integridad conceptual.

En Python, por ejemplo, existe la guía de estilo PEP 8, que ayuda a mantener un estilo de programación uniforme entre desarrolladores que la siguen. Esta guía, por ejemplo, establece, entre otras cosas, convenciones para importar módulos:

```
# Correct:
import os
import sys

# Wrong:
import sys, os
```

```
# Correct:
from subprocess import Popen, PIPE
```

8.2 Ocultamiento de Informacion

Algunos autores prefieren el término *encapsulamiento*, que se refiere al nivel de ocultamiento de información de los datos. Por ejemplo, en la programación orientada a objetos, el encapsulamiento suele estar asociado a las palabras clave `private`, `public` y `protected`, las cuales controlan el acceso a los atributos y métodos de una clase de diferentes maneras. Por ejemplo, `private` restringe el acceso a los atributos y métodos exclusivamente dentro de la misma clase, mientras que `protected` permite que las clases hijas —es decir, aquellas que heredan de la clase original— puedan acceder a dichos datos. Esta misma idea puede aplicarse no solo a nivel de clases, sino también a nivel de paquetes, componentes o subsistemas.

La idea general es que, si los componentes de un sistema son más independientes entre sí, será más fácil mantenerlos y comprenderlos. Ocultar información entre componentes facilita esta independencia. Mientras menos sepa un componente sobre la implementación o los datos internos de otro, mayor será su autonomía. Esta independencia ofrece varias ventajas:

- Permite distribuir de mejor manera las tareas entre desarrolladores. Por ejemplo, un desarrollador puede trabajar en un componente A y otro en un componente B de forma paralela.
- Facilita la sustitución o eliminación de un componente. Por ejemplo, se puede reemplazar una clase por otra que implemente un algoritmo más eficiente sin afectar el resto del sistema.
- Mejora la comprensibilidad, ya que cada componente se encarga de funcionalidades específicas, operando de manera separada del resto.

Para lograr esta independencia, las clases o componentes deben ocultar tanto sus datos como la lógica interna con la que resuelven sus operaciones. De esta manera, los demás componentes no dependerán de los detalles internos. Esto no significa que una clase no deba relacionarse con otras, sino que debe hacerlo a través de un conjunto acotado de métodos públicos (interfaces) que definan claramente cómo interactuar con ella.

Por lo comentado anteriormente, lo lógico es que las clases mantengan sus datos y la mayoría de sus métodos como privados, exponiendo únicamente un conjunto reducido de métodos públicos para comunicarse con otras clases.

En este contexto, los métodos *getters* y *setters* son funciones que permiten acceder y modificar los datos privados de una clase. Sin embargo, al utilizarlos, se abre la visibilidad de esos datos, lo que puede incrementar la dependencia de otras clases respecto a la implementación interna. Como resultado, cualquier cambio en dichos datos podría tener un efecto en todas las clases que dependan de esos *getters* y *setters*.

Por esta razón, se recomienda limitar su uso al mínimo necesario. Es preferible diseñar interfaces que expongan solo aquellas operaciones que realmente sean necesarias para la interacción con otros componentes, evitando exponer directamente los atributos internos de la clase.

Ejemplo 1.A: Caja fuerte con datos publicos. A continuación, se presenta una clase `CajaFuerte` que posee dos datos y un único método `open`para acceder al dato secreto, siempre y cuando la contraseña proporcionada sea correcta.

```
class CajaFuerte:
    def __init__(self, dato_secreto):
        self._dato_secreto = dato_secreto
        self._password = "1234"

    def open(self, password):
        if password == self._password:
            return self._dato_secreto
        else:
            raise ValueError("Contraseña incorrecta.")
```

Imaginemos que desde otra parte del programa, existe una función que recibe un objeto `CajaFuerte`. Esta puede usar el método público para acceder al dato secreto si la contraseña es correcta. Sin embargo, también puede acceder directamente al dato y a la contraseña, ya que estos son publicos.

```
def external_funcion(caja):
  print(caja.open("1234"))
  print(caja._dato_secreto)
  print(caja._password)
```

Como se puede observar, es posible escribir código que ignore el protocolo `open` definido para acceder a los datos. Además, el esta función necesita conocer los detalles internos de la implementación de la clase: qué atributos existen y cómo se llaman. Por lo tanto, si en algún momento se cambia el nombre de un atributo, esta función también deberá actualizarse.

Ejemplo 1.B: Caja fuerte con datos privados. Veamos el mismo ejemplo, pero encapsulando los datos de forma apropiada:

```
class CajaFuerte:
    def __init__(self, dato_secreto):
        self.__dato_secreto = dato_secreto. # atributo privado
        self.__password = "1234"            # atributo privado

    def acceder_secreto(self, password):
        if password == self.__password:
            return self.__dato_secreto
```

```
        else:
            raise ValueError("Contraseña incorrecta.")
```

Ahora, imaginemos que otro programador intenta escribir una función como la anterior.

```
def external_funcion(caja):
    print(caja.acceder_secreto("1234"))   # OK
    print(caja.__dato_secreto)            # Error
    print(caja.__password)                # Error
```

Esta función generará errores al intentar acceder directamente a los atributos privados. Al declarar los datos como privados, forzamos a otros desarrolladores a interactuar con nuestra clase únicamente a través de los métodos públicos que definimos. Además, evitamos que el resto del código dependa de los detalles internos de nuestra implementación. Por lo tanto, el encapsulamiento permite ocultar información y garantizar que otros componentes dependan únicamente de las operaciones públicas.

8.3 Cohesion

El diccionario define **cohesión** como la acción y efecto de reunirse o adherirse las cosas entre sí, o a la materia de la que están formadas. Si aplicamos este concepto a las clases en programación orientada a objetos, una clase debe tener una única responsabilidad o proporcionar un servicio específico. Por lo tanto, los componentes dentro de la clase —es decir, sus métodos y atributos— deben estar relacionados entre sí para cumplir con ese servicio o responsabilidad.

Por ejemplo, si un método dentro de una clase no utiliza ningún atributo ni invoca otros métodos de la misma clase, significa que no está cohesionado con el resto de la clase. En consecuencia, no aporta directamente a su razón de ser.

Esta propiedad está relacionada con el **principio de separación de responsabilidades**, el cual recomienda que cada clase tenga una única responsabilidad. Si dentro de una clase existen componentes que no están relacionados entre sí, esto puede ser un indicador de que la clase está asumiendo más de una responsabilidad. En esos casos, una parte de sus componentes colabora en un objetivo común, mientras que la otra parte colabora para resolver otro distinto.

Ejemplo 2. Considere la siguiente clase de ejemplo:

```
class AB:
    def __init__(self, d1, d2, d3, d4):
        self.d1 = d1
        self.d2 = d2
        self.d3 = d3
        self.d4 = d4
```

```
    def operation1(self):
        return self.d1 + self.d2

    def operation2(self):
        return self.d3 + self.d4
```

Esta clase por ejemplo tiene cuatro atributos, dos de ellos relacionados con la primera operacion y otros dos con la segunda. Este es un ejemplo de una clase con baja cohesion. Ya que no todos sus compomentes estan relacionados entre si. `operacion1` no tiene relacion con `operacion2`, `d1+d2` no tiene relacion con `d3+d4`. Una forma de identificar baja cohesion, es analizar la posiblidad de extraer o llevar un metodo de una clase a otra. Si esta se puede mover muy facilmente, es que no estaba cohesionada con el resto de la clase. En este ejemplo claramente se puede partir en dos clases.

```
class A:
    def __init__(self, d1, d2):
        self.d1 = d1
        self.d2 = d2

    def operation1(self):
        return self.d1 + self.d2


class B:
    def __init__(self, d3, d4):
        self.d3 = d3
        self.d4 = d4

    def operation2(self):
        return self.d3 + self.d4
```

8.4 Acoplamiento

Acoplamiento se refiere a la coneccion entre modulos. Relacionado con la indepedencia de componentes lo ideal seria que los modulos depende lo menos posible entre si. Obviamente no pueden ser islas totalmente independientes, tienen que colaborar con el resto del modulos del sistema, de forma minima y a travez de interfaces publicas bien definidas, por ejemplo, métodos públicos. Esto significa que la firma (nombre, argumento y tipo de retorno) de los metodos públicos no cambia frecuentemente. Ya que varios clases podrian comunicarse a travez de estos metodos publicos y cualquier cambio o bug, puede propagarse a las clases que depende de este metodo.

Ejemplo 3. Considere las clases `BookStore` y `Book`. La clase `BookStore` depende de `Book` en dos lugares: cuando llama a `libro.get_autor()` y a

`libro.get_titulo()`. Por lo tanto, podemos decir que estas clases están acopladas.

```
class Book:
    def __init__(self, titulo, autor):
        self.__titulo = titulo
        self.__autor = autor

    def get_titulo(self):
        return self.__titulo

    def get_autor(self):
        return self.__autor

class BookStore:
    def __init__(self):
        self._libros = []

    def agregar_libro(self, libro):
        self._libros.append(libro)

    def mostrar_libros(self):
        for libro in self._libros:
            print(f"Autor: {libro.get_autor()},
                  Título: {libro.get_titulo()}")
```

En la siguiente versión, la clase `Book` ya no expone métodos accesores. En cambio, proporciona un único método público mostrar que encapsula la forma en que se imprimen sus datos. Así, `BookStore` ya no accede directamente a los atributos ni necesita conocer los detalles internos de `Book`.

```
class Book:
    def __init__(self, titulo, autor):
        self.__titulo = titulo
        self.__autor = autor

    def mostrar(self):
        print(f"Autor: {self.__autor}, Título: {self.__titulo}")

class BookStore:
    def __init__(self):
        self._libros = []

    def agregar_libro(self, libro):
        self._libros.append(libro)

    def mostrar_libros(self):
```

```
        for libro in self._libros:
            libro.mostrar().
```

Con esta refactorización, la clase `BookStore` depende de `Book` en un solo lugar, manteniendo exactamente la misma funcionalidad que antes. Esto reduce el acoplamiento entre ambas clases. La hipotesis es que mientras menos acopladas esten las clases el mantenimiento es mas sencillo.

8.5 SOLID

Estos principios apuntan a resolver problemas y asegurarse que la solución propuesta sea facilmente mantenible y pueda evolucionar en el tiempo. Las tareas de mantencion como agregar funcionalidad, cambios y arreglar bugs, pueden llegar a convertirse en el tiempo lentas, costoas y resigosas. Los principios apuntan a ayudarnos en esta situacion.

- El Principio de responsabilidad única (**S**ingle Responsibility Principle)
- El Principio Abierto-Cerrado (**O**pen-Closed Principle)
- El Principio de sustitución de Liskov (**L**iskov Substitution Principle)
- El Principio de segregación de interfaz (**I**nterface Segregation Principle)
- El Principio de inversión de dependencia (**D**ependency Inversion Principle)

Single Responsability Principle

Este principio es la aplicacion de la idea de cohesion. Propone que cada clase tenga una sola responsabilidad..

Definition. El principio de responsabilidad simple o unica establece que *una clase debe tener una rason para cambiar.* En otras palabras, una clase debe tener una responsabilidad o un solo trabajo. Si una clase maneja multiples responsabilidades, el cambio de una responsabilidad puede afectar a otra, haciendo que la clase sea dificil de mantener y sea mas propensa a errores.

Ejemplo. Considere la siguiente clase:

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def save_to_db(self):
        print(f"Saving {self.name} to database")

    def send_welcome_email(self):
        print(f"Sending welcome email to {self.email}")
```

Esta clase tiene mutilpes responsabilidades: guardar la informacion, guardar en la base de datos y enviar emails.

Posible Mejora. Separar las responsabilitdades en diferenes clases.

```
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserRepository:
    def save(self, user):
        print(f"Saving {user.name} to database")

class EmailService:
    def send_welcome_email(self, user):
        print(f"Sending welcome email to {user.email}")
```

Interface Segregation Principle

Definición. El *Principio de Segregación de Interfaces (ISP)* establece que *ninguna clase debe verse forzada a depender de métodos que no utiliza.* En otras palabras, es mejor tener *interfaces pequeñas y específicas*, en lugar de una gran interfaz general. Si una clase depende de una interfaz que tiene métodos que no necesita, esa clase está violando este principio.

Ejemplo. Supongamos una interfaz `Trabajador` que agrupa comportamientos para distintos tipos de empleados:

```
class Trabajador:
    def trabajar(self):
        pass

    def comer(self):
        pass

class EmpleadoOficina(Trabajador):
    def trabajar(self):
        print("Trabajando en tareas de oficina")

    def comer(self):
        print("Hora de almorzar")

class Robot(Trabajador):
    def trabajar(self):
        print("Realizando trabajo automatizado")
```

```
    def comer(self):
        # El robot no come, pero se ve obligado a implementar este método
        raise NotImplementedError("Los robots no comen")
```

La clase Robot está violando el ISP porque se ve obligada a implementar un método que no necesita.

Posible Mejora. Repartir el comportamiento en mas interfaces.

```
class PuedeTrabajar:
    def trabajar(self):
        pass

class PuedeComer:
    def comer(self):
        pass

class EmpleadoOficina(PuedeTrabajar, PuedeComer):
    def trabajar(self):
        print("Trabajando en tareas de oficina")

    def comer(self):
        print("Hora de almorzar")

class Robot(PuedeTrabajar):
    def trabajar(self):
        print("Realizando trabajo automatizado")
```

Dependency Inversion Principle

Definición. El *Principio de Inversión de Dependencias* es el último de los principios SOLID y establece que: (a) *Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.*; (b) *Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.*

En otras palabras las clases de alto nivel (que contienen lógica de negocio) no deben depender directamente de clases concretas. Se debe programar contra interfaces o abstracciones, no contra implementaciones.

Ejemplo. Una clase `Orden` que depende directamente de una clase concreta `MySQLDatabase`.

```
class MySQLDatabase:
    def guardar(self, datos):
        print(f"Guardando {datos} en base de datos MySQL")

class Orden:
```

```
    def __init__(self):
        self.db = MySQLDatabase()  # Dependencia concreta

    def procesar(self):
        self.db.guardar("datos de la orden")
```

Aquí la clase Orden depende directamente de `MySQLDatabase`, lo que acopla fuertemente la lógica del negocio con una implementación específica de almacenamiento.

Posible Mejora. Usar una abstracción (`Repositorio`) que puede ser implementada por cualquier clase concreta (como `MySQLDatabase`, `PostgresDatabase`, `MockDatabase`, etc.).

```
# Abstracción
class Repositorio:
    def guardar(self, datos):
        pass

# Implementación concreta
class MySQLDatabase(Repositorio):
    def guardar(self, datos):
        print(f"Guardando {datos} en base de datos MySQL")

# Clase de alto nivel depende de la abstracción
class Orden:
    def __init__(self, repositorio: Repositorio):
        self.repositorio = repositorio

    def procesar(self):
        self.repositorio.guardar("datos de la orden")
```

Ejemplo de uso:

```
db = MySQLDatabase()
orden = Orden(db)
orden.procesar()
```

Ahora esposible cambiar la base de datos sin cambiar la logica del negocio. Por ejemplo ahora en lugar de tener una base de datos MySQL, puede ser PostgreSQL o Firebase. Reduce el acoplamiento y mejora la extensibilidad.

Prefer Composition Over Inheritance

Definición. *Preferir composición sobre herencia* es un principio de diseño que sugiere que es mejor **reutilizar código mediante la composición** (tener objetos como atributos) que mediante la **herencia** (extender clases).

¿Por qué? - La herencia crea una relación fuerte entre clases: un cambio en la clase padre puede afectar a todas sus hijas. - Con la composición, se construyen objetos complejos a partir de otros más simples, manteniendo bajo acoplamiento y mayor flexibilidad. - La composición permite reutilizar funcionalidades entre clases no relacionadas, mientras que la herencia fuerza una jerarquía.

Ejemplo. Considere un reproductor de música con modos intercambiables. Supongamos que queremos un reproductor de música que pueda reproducir canciones de forma *secuencial* o *aleatoria*, y cambiar entre modos sin tener que reiniciarlo o reescribir clases. Considere el siguiente ejemplo:

```
import random

# Clase base para el reproductor
class MusicPlayer:
    def __init__(self):
        self.canciones = []

    def agregar_cancion(self, cancion):
        self.canciones.append(cancion)

    def reproducir(self):
        raise NotImplementedError("Este método debe ser implementado por la subcl

# Reproductor secuencial (subclase)
class MusicPlayerSecuencial(MusicPlayer):
    def reproducir(self):
        for cancion in self.canciones:
            print(f"Reproduciendo: {cancion}")

# Reproductor aleatorio (subclase)
class MusicPlayerAleatorio(MusicPlayer):
    def reproducir(self):
        for cancion in random.sample(self.canciones, len(self.canciones)):
            print(f"Reproduciendo: {cancion}")

# Uso
player = MusicPlayerSecuencial()
player.agregar_cancion("Canción 1")
player.agregar_cancion("Canción 2")
player.agregar_cancion("Canción 3")

print("Modo secuencial:")
```

```
player.reproducir()

# Si queremos cambiar a modo aleatorio:
# Tenemos que crear un nuevo objeto y perder el anterior
print("\nCambiando a modo aleatorio:")
player = MusicPlayerAleatorio()  # Perdemos la lista anterior
player.agregar_cancion("Canción 1")
player.agregar_cancion("Canción 2")
player.agregar_cancion("Canción 3")
player.reproducir()
```

En código anterior cuando creamos un objecto `MusicPlayerSecuencial` este reproducira de forma secuencial todos su ciclo de vida. Para reproducir en aletaorio tenemos que crear un nuevo objeto pero de la clase `MusicPlayerAleatorio`. Esto no emula la vida real, donde uno abre Spotify, y puede cambiar de modo, sin tener que crear una nueva ventaja de Spotify.

Posible Solución. Un solución es encapsular el modo de reproducción en otra clase utilizando composición.

```
import random

class ModoSecuencial:
    def reproducir(self, canciones):
        for cancion in canciones:
            print(f"Reproduciendo: {cancion}")

class ModoAleatorio:
    def reproducir(self, canciones):
        for cancion in random.sample(canciones, len(canciones)):
            print(f"Reproduciendo: {cancion}")

class MusicPlayer:
    def __init__(self, modo):
        self.modo = modo
        self.canciones = []

    def agregar_cancion(self, cancion):
        self.canciones.append(cancion)

    def reproducir(self):
        self.modo.reproducir(self.canciones)

    def cambiar_modo(self, nuevo_modo):
        self.modo = nuevo_modo

# Uso:
```

```
secuencial = ModoSecuencial()
aleatorio = ModoAleatorio()

player = MusicPlayer(secuencial)
player.agregar_cancion("Canción 1")
player.agregar_cancion("Canción 2")
player.agregar_cancion("Canción 3")

print("Modo secuencial:")
player.reproducir()

print("\nCambiando a modo aleatorio:")
player.cambiar_modo(aleatorio)
player.reproducir()
```

Ventajas de usar composición:

- Se puede cambiar el comportamiento del reproductor en tiempo de ejecución.
- `MusicPlayer` no necesita saber cómo se implementa cada modo.
- Cada modo está encapsulado y es reutilizable.
- No se necesita herencia ni una jerarquía complicada.

Principio de Demeter

Este principio afirma que los métodos de una clase solo deben llamar a: (1) metodos de la misma clase, (2) metodos de objetos recibidos como parametros, (3) metodos de objetos creados dentro del mismo metodo, y (4) atributos de la misma clase.

Definición. El **Principio de Deméter** (también conocido como "no hables con extraños") recomienda que: *Un objeto solo debe interactuar con objetos que conoce directamente.* En otras palabras, una clase no debe acceder a objetos anidados profundos de otros objetos. Esto evita el acoplamiento excesivo y mejora la encapsulación. Si una clase necesita acceder a múltiples niveles de objetos (`a.get_b().get_c().hacer_algo()`), probablemente esté violando este principio.

Ejemplo. Considere la siguiente ejemplo:

```
class Direccion:
    def __init__(self, ciudad):
        self.ciudad = ciudad

class Cliente:
    def __init__(self, direccion):
        self.direccion = direccion
```

```
class Pedido:
    def __init__(self, cliente):
        self.cliente = cliente

    def imprimir_ciudad(self):
        # Violación del Principio de Deméter
        print(f"Ciudad del cliente: {self.cliente.direccion.ciudad}") # << vio
```

La clase `Pedido` está accediendo directamente a los detalles internos del cliente (direccion), lo cual la hace dependiente de la estructura interna del objeto `Cliente`. Si esa estructura cambia, Pedido también tendrá que cambiar.

Posible Mejora. Podemos mejorar el código delegando la solucitud a los objetos compuestos. En este caso, el pedido puede delegar la operación a cliente ya que el es el que tiene la operación.

```
class Direccion:
    def __init__(self, ciudad):
        self.ciudad = ciudad

class Cliente:
    def __init__(self, direccion):
        self.direccion = direccion
    def ciudad(self):
        self direccion.ciudad

class Pedido:
    def __init__(self, cliente):
        self.cliente = cliente

    def imprimir_ciudad(self):
        # Violación del Principio de Deméter
        print(f"Ciudad del cliente: {self.cliente.ciudad}") # Ok
```

Principio Abierto y Cerrado

Es principio afirma que una clase debe ser cerrada para modificaciones pero abierto para extensiones. El concepto principal es que cuando tu implementas una clase, debes preparar a la misma para que sea modificada los menos posible, pero permitir que en el futuro otros desarrolladores o tu mismo puedan extender funcionalidad de la misma. Esta extension se puede hacer utilizando por ejemplmo, parametros, herencia, funciones, o objetos.

Definición. El **Principio Abierto/Cerrado** establece que: *El software debe estar abierto para su extensión, pero cerrado para su modificación.* Esto significa que *deberíamos poder agregar nuevas funcionalidades al sistema sin tener que modificar el código existente*, especialmente el que ya ha sido probado. En la

práctica, esto se logra utilizando *abstracciones* como clases base, interfaces, o funciones que permiten inyectar comportamiento.

Ejemplo. Considere el siguiente código de ejemplo:

```
class CalculadoraDescuentos:
    def calcular(self, tipo_cliente, monto):
        if tipo_cliente == "regular":
            return monto * 0.95
        elif tipo_cliente == "vip":
            return monto * 0.90
        elif tipo_cliente == "supervip":
            return monto * 0.85
        else:
            return monto
```

Este código viola el principio porque cada vez que se agrega un nuevo tipo de cliente, hay que modificar la clase existente, lo que puede introducir errores y rompe el principio de estar *"cerrado para modificaciones"*.

Posible Mejora. Podemos usar composición y crear clases que sepan calcular el descuento de acuerdo a su tipo.

```
# Clase base
class EstrategiaDescuento:
    def calcular(self, monto):
        return monto

# Subclases con distintos comportamientos
class DescuentoRegular(EstrategiaDescuento):
    def calcular(self, monto):
        return monto * 0.95

class DescuentoVip(EstrategiaDescuento):
    def calcular(self, monto):
        return monto * 0.90

class DescuentoSuperVip(EstrategiaDescuento):
    def calcular(self, monto):
        return monto * 0.85

# Clase de alto nivel que usa la estrategia
class CalculadoraDescuentos:
    def __init__(self, estrategia: EstrategiaDescuento):
        self.estrategia = estrategia

    def calcular(self, monto):
        return self.estrategia.calcular(monto)
```

```
cliente_vip = CalculadoraDescuentos(DescuentoVip())
print(cliente_vip.calcular(100))  # 90.0
```

Ventajas: - Se puede agregar un nuevo tipo de descuento sin modificar las clases existentes. - Cada nueva estrategia puede probarse por separado. - Facilita la extensión y el mantenimiento del sistema. - Aplica muy bien con el patrón Strategy.

Principo de Substitucion de Liskov

Como se discutio anteriormente es preferible composicion sobre la herencia. El principio de substitucion de Liskov establece reglas para redefinir métodos en subclases. Este principo es en honor a Barbara Liskov, una profesora del MIT que gano el premio Turing en 2008.

El mismo establece que si una clase B hereda de A, un objeto de la clase B puede enviarse/usarse en cualquier lugar del programa donde esperan un objeto de la clase A. Por ejemplo, el registro civil da carnet de identidad a todos los ciudadanos, entonces un ingenierio o abogado pueden sacar carnet, porque ambos son personas. En programacion podriamos decir que abogado hereda e ingeniero hereda de persona o mas formalmente abogado e ingeniero son subtipos de persona. Desde este punto de vista la operacion de sacar carnet debe ser suficientemente generica que cualquier persona pueda realizar la operacion. Si en el futuro existen mas subtipos de persona, estos facilmente podran sacar carnet.

Definición. El **Principio de Sustitución de Liskov** establece que *Las clases derivadas deben poder sustituir a sus clases base sin alterar el comportamiento esperado del programa.* En otras palabras, si una clase `Hija` hereda de una clase `Padre`, debería poder usarse *en lugar* de `Padre` sin que el código que la utiliza falle o se comporte de forma incorrecta. Si la subclase rompe las expectativas del cliente del código, entonces *viola el principio.*

Ejemplo. Considere el siguiente ejemplo, tenemos que pingüino hereda de ave, lo que implicaria un pinguino puede hacer todo lo que una ave, en este ejemplo, volar:

```
class Ave:
    def volar(self):
        print("Estoy volando")

class Pinguino(Ave):
    def volar(self):
        raise NotImplementedError("Los pingüinos no pueden volar")

def hacer_volar(ave):
    ave.volar()
```

```
def hacer_volar(ave: Ave):
    ave.volar() # Error: Los pingüinos no vuelan
```

Aunque Pinguino hereda de Ave, no cumple con las expectativas del cliente. La función `hacer_volar` asume que todas las aves pueden volar. Esto viola el principio de sustitución de Liskov.

Posible Mejora. Podemos agregar un nueva clase que represente a las aves que si pueden volar `AveVoladora`, Entonces, pinguino heredara de `Ave`, pero no de `AveVoladora`.

```
class Ave:
    def hacer_sonido(self):
        print("¡Pío!")

class AveVoladora(Ave):
    def volar(self):
        print("Estoy volando")

class Golondrina(AveVoladora):
    pass

class Pinguino(Ave):
    def nadar(self):
        print("Estoy nadando")

def hacer_volar(ave: AveVoladora):
    ave.volar()

g = Golondrina()
hacer_volar(g)    # Okay
p = Pinguino()
# hacer_volar(p)  # Error de tipo
```

Ahora como función `hacer_volar` recibe una `AveVoladora` puede tranquilamente llamar al método `volar`. Ya que si alguien quiere enviar un pinguino a esta función deberia haber un error de tipo, por lo que el lenguaje deberia decirle al desarrollador que quiere hacer esto, que pinguino no puede pasar por aqui porque no puede volar, porque no es subtipo de `AveVoladora`.

8.6 Code Smells

Si bien en base a los principios uno ya puede tener una idea si su código va teniendo un buen diseño. Otra forma de detectar posibles problemas de calidad, incluido problemas de diseño, son los `Code Smells`. Los `Code Smells` son pedasos de código que huelen mal. Detectarlos puede ayudarte a detectar problemas de calidad en etapas tempranas.

Duplicated Code

Definición. Ocurre cuando fragmentos de código similares o idénticos se repiten en diferentes partes del sistema. Esto aumenta el esfuerzo de mantenimiento, ya que cualquier cambio debe replicarse en múltiples lugares.

```
def greet_morning(name):
    print(f"Good morning, {name}!")

def greet_evening(name):
    print(f"Good evening, {name}!")
```

Posible Mejora. Para resolver este problema, normalmente debemos identificar qué partes del código son idénticas y cuáles cambian. Luego, lo recomendable es abstraer aquello que varía y hacerlo intercambiable. Por ejemplo, en este caso, lo que cambia es el saludo (morning o evening), lo cual se puede parametrizar fácilmente.

```
def greet(name, time_of_day):
    print(f"Good {time_of_day}, {name}!")

greet("Alice", "morning")
greet("Bob", "evening")
```

Otro Ejemplo. A veces quitar el código duplicado no es tan facil, existe varias formas, por ejemplo, como vimos antes podemos parametrizar la parte que cambia. Considere el siguiente ejemplo:

```
class BookStore:
    def __init__(self):
        self.books = []

    def add(self, book):
        self.books.append(book)

    def filter_by_author(self, name):
        for book in self.books:
            if book.author == name:
                book.print_book()

    def filter_by_title(self, title):
        for book in self.books:
            if book.title == title:
                book.print_book()
```

Aqui existe codigo duplicado en los dos metodos filters. Sin embargo, ahora es un poco mas dificil porque la parte que varia entre estos dos métodos es la condición. Como puedo parametrizar la condición? En programación funcional

podriamos enviar como argumento una funcion y en programación orientada objetos podemos enviar un objeto. Veamos la solucion objetual.

```
class BookStore:
    def __init__(self):
        self.books = []

    def add(self, book):
        self.books.append(book)

    def filter(self, criteria):
        for book in self.books:
            if criteria.matches(book):
                book.print_book()


class FilterByAuthor:
    def __init__(self, author):
        self.author = author

    def matches(self, book):
        return book.author == self.author


class FilterByTitle:
    def __init__(self, title):
        self.title = title

    def matches(self, book):
        return book.title == self.title
```

Ahora enviamos la condicion encapsulada en un objeto. Para filtrar necesitariamos escribir el siguiente codigo:

```
store = BookStore()
store.add(Book("1984", "George Orwell"))
store.add(Book("Animal Farm", "George Orwell"))
store.add(Book("Brave New World", "Aldous Huxley"))

print("Books by Orwell:")
store.filter(FilterByAuthor("George Orwell"))

print("\nBooks titled '1984':")
store.filter(FilterByTitle("1984"))
```

Long Method

Definicion. Un método que realiza demasiadas tareas, dificultando su lectura, prueba y mantenimiento. La definicion de que es un metodo largo varia entre la bibliografia, pero mas alla de ver cuantas lineas tiene debemos analizar el porque este metodo se ve largo. Normalmente los metodos largos hacen muchas operaciones al mismo tiempo. Esto nos ayuda a identificar posibles mejoras.

Ejemplo. Aquí un ejemplo de una función que hace varias tareas:

```
def procesar_pedido(pedido):
    # Validar datos
    if not pedido['cliente'] or not pedido['productos']:
        raise ValueError("Datos incompletos")

    # Calcular total
    total = 0
    for producto in pedido['productos']:
        total += producto['precio'] * producto['cantidad']

    # Aplicar impuestos
    total *= 1.19

    # Registrar en sistema
    print(f"Pedido de {pedido['cliente']} registrado. Total: {total}")
```

Posible Mejora. Dividir el método en funciones más pequeñas y específicas.

```
def validar_pedido(pedido):
    if not pedido['cliente'] or not pedido['productos']:
        raise ValueError("Datos incompletos")

def calcular_total(pedido):
    subtotal = sum(p['precio'] * p['cantidad'] for p in pedido['productos'])
    return subtotal * 1.19

def registrar_pedido(pedido, total):
    print(f"Pedido de {pedido['cliente']} registrado. Total: {total}")

def procesar_pedido(pedido):
    validar_pedido(pedido)
    total = calcular_total(pedido)
    registrar_pedido(pedido, total)
```

Note que ahora es probable que tengamos igual o mayor numero de lineas, pero el código esta mas organizado, y fácil de leer. Ademas, todas las funciones creadas pueden ser potencialmente re utilizadas en un futuro.

Long Class

Definición. Una clase que agrupa demasiadas responsabilidades. Es difícil de entender, mantener y reutilizar.

Ejemplo. Considere el siguiente ejemplo:

```
class Usuario:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email

    def cambiar_email(self, nuevo_email):
        self.email = nuevo_email

    def enviar_correo_bienvenida(self):
        print(f"Enviando correo de bienvenida a {self.email}")

    def calcular_descuento(self, compras):
        return sum(compras) * 0.1
```

Mejora. Aplicar el principio de responsabilidad simple (SRP) y dividir en clases con responsabilidades claras.

```
class Usuario:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email

    def cambiar_email(self, nuevo_email):
        self.email = nuevo_email

class Notificador:
    def enviar_correo_bienvenida(self, usuario):
        print(f"Enviando correo de bienvenida a {usuario.email}")

class Descuento:
    def calcular(self, compras):
        return sum(compras) * 0.1
```

Nota este es un ejemplo de como se puede dividir, dependiendo del programa y situacion puede variar.

Feature Envy

Definición Una clase utiliza más los datos de otra clase que los suyos propios, indicando que cierta lógica debería trasladarse.

Ejemplo. Por ejemplo, en esta clase utiliza los datos del libro. La pregunta es porque pedirle los datos al libro, si el libro tiene todo lo necesario para hacer la operación porque no le pedimos al libro que haga la operación (delegación).

```
class Reporte:
    def mostrar_libro(self, libro):
        print(f"{libro.get_titulo()} - {libro.get_autor()}")
```

Posible Solución Mover la responsabilidad a la clase Book.

```
class Book:
    def mostrar(self):
        print(f"{self.__titulo} - {self.__autor}")

class Reporte:
    def mostrar_libro(self, libro):
        libro.mostrar()
```

Long Parameters List

Definición. Una función o constructor recibe demasiados parámetros, lo que puede indicar un problema de diseño. Es importante analizar si los parametros tiene relacion entre ellos, ya que tal vez deberian estar encapsulados en una o dos clases.

Ejemplo. Considere esta función con muchos parametros.

```
def crear_usuario(nombre, apellido, edad, email, telefono, direccion, ciudad,
    return ...
```

Posible Mejora. Agrupar los parámetros relacionados en clases o estructuras específicas.

```
class Contacto:
    def __init__(self, email, telefono):
        self.email = email
        self.telefono = telefono

class Direccion:
    def __init__(self, direccion, ciudad, pais, codigo_postal):
        self.direccion = direccion
        self.ciudad = ciudad
        self.pais = pais
        self.codigo_postal = codigo_postal

class Usuario:
    def __init__(self, nombre, apellido, edad, contacto, direccion):
        self.nombre = nombre
```

```
        self.apellido = apellido
        self.edad = edad
        self.contacto = contacto
        self.direccion = direccion
```

Global Variables

Las variables globales pueden introducir distintos tipos de errores inesperados. Por ejemplo, si se define una variable global que debe contener solo números positivos, no hay garantía de que, en algún lugar del programa, otra parte del código no le asigne un valor negativo. Además, se asume que todo el equipo de desarrollo —tanto actual como futuro— sabrá exactamente qué tipo de dato debe almacenarse en esa variable. Si alguien la modifica y le asigna un estado no esperado, todo el sistema podría fallar.

Este tipo de variables también puede romper supuestos básicos sobre el comportamiento del código. Por ejemplo, sabemos que en matemáticas la suma es conmutativa, es decir, A + B es igual a B + A. Sin embargo, con variables compartidas, esto puede dejar de cumplirse.

Por ejemplo, considere el siguiente ejemplo con una variable global.

```
contador = 0

def agregar_uno():
    global contador
    contador += 1
    return contador

def agregar_dos():
    global contador
    contador += 2
    return contador

# Suma de los resultados en distinto orden
resultado_1 = agregar_uno() + agregar_dos()  # 1 + 3 = 4
contador = 0  # reiniciamos para probar otra combinación
resultado_2 = agregar_dos() + agregar_uno()  # 2 + 3 = 5
```

El ejemplo anterior muestra que si se llaman las funciones en distinto orden, los resultados pueden variar. Esto evidencia, una vez más, los riesgos asociados al uso de variables globales.

Una forma más controlada de manejar datos globales es mediante el uso del patrón Singleton. Este patrón garantiza que una clase tenga una única instancia y proporciona un punto de acceso global a ella. Si desean profundizar más, pueden buscar información sobre el patrón Singleton en internet.

Primitive Obsession

El *Primitive Obsession* ocurre cuando se usan tipos primitivos (como `int`, `str`, `float`, etc.) en lugar de crear clases o estructuras compuestas que representen conceptos del dominio. Este olor puede dificultar la validación, el mantenimiento del código y la claridad del diseño.

En lugar de usar múltiples strings, enteros o booleans para representar datos complejos, deberíamos crear tipos específicos que encapsulen comportamiento, reglas y validaciones.

Ejemplo.
Un sistema que representa direcciones usando solo strings:

```
def enviar_paquete(calle, ciudad, codigo_postal):
    if len(codigo_postal) != 5:
        raise ValueError("Código postal inválido")
    print(f"Enviando paquete a {calle}, {ciudad}, {codigo_postal}")

# Uso
enviar_paquete("Av. Siempre Viva 742", "Springfield", "123")
```

Este enfoque es propenso a errores, ya que no hay una forma clara de validar o estructurar los datos de dirección. Además, la validación se repite cada vez que se usa la dirección.

Mejora Crear una clase Direccion que encapsule los detalles y reglas del dominio.

```
class Direccion:
    def __init__(self, calle, ciudad, codigo_postal):
        if len(codigo_postal) != 5:
            raise ValueError("Código postal inválido")
        self.calle = calle
        self.ciudad = ciudad
        self.codigo_postal = codigo_postal

    def __str__(self):
        return f"{self.calle}, {self.ciudad}, {self.codigo_postal}"

def enviar_paquete(direccion):
    print(f"Enviando paquete a {direccion}")

# Uso
direccion = Direccion("Av. Siempre Viva 742", "Springfield", "12345")
enviar_paquete(direccion)
```

Ahora por ejemplo, cada vez que se cree una dirección se validara en el constructor.