1. Introducción

Como puede observarse de los apuntes anteriores la tarea de diseñar software puede ser bastante compleja, especialmente si se desea hacer software escalable de alto rendimiento. Aquí se hará una breve descripción de los 5 principios de diseño propuestos por Robert C. Martin[1]. Aplicados en conjunto, estos principios, conocidos actualmente por sus iniciales en inglés como **Principios S.O.L.I.D.**[2], hacen más probable que un sistema sea fácil de mantener y ampliar con el tiempo.

La *escalabilidad* se entiende como la capacidad de un software de modificar, expandir o ampliar su memoria, potencial de respuesta y funcionalidades. Desde el punto de vista del diseño, debe permitir agregar, modificar o quitar funcionalidades del sistema fácilmente.

2. Principio de responsabilidad única

El principio indica que **cada clase debe tener un único motivo para cambiar**. Esto significa que una clase debe tener una única responsabilidad, expresada a través de sus métodos. Si una clase se encarga de más de una tarea, entonces debe separar esas tareas en clases independientes.

Este principio está ligado al concepto de la **separación de intereses**, también denominada separación de preocupaciones o separación de conceptos (*separation of concerns*), es un principio de diseño para separar un programa informático en secciones distintas, tal que cada sección enfoca un interés delimitado.

Para ilustrar el principio de responsabilidad única, observe el siguiente ejemplo donde no se aplica en la clase `Personaje`.

[1] Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
[2] Los principios S.O.L.I.D. en inglés son:

1. The **S**ingle Responsibility Principle (*SRP*): A class should have one, and only one, reason to change.
2. The **O**pen Closed Principle (*OCP*): You should be able to extend a classes behavior, without modifying it.
3. The **L**iskov Substitution Principle (*LSP*): Derived classes must be substitutable for their base classes.
4. The **I**nterface Segregation Principle (*ISP*): Make fine grained interfaces that are client specific.
5. The **D**ependency Inversion Principle (*DIP*): Depend on abstractions, not on concretions.

Programa 1: Violación del SRP

```python
class Item:
  def __init__(self, nombre) -> None:
    self.__nombre = nombre

  def activar_efecto(self) -> None:
    print(f"{self.__nombre} se activó")

  def __eq__(self, other) -> bool:
    return self.__nombre == other.__nombre

  def __str__(self) -> str:
    return self.__nombre

class Personaje:
  def __init__(self, nombre: str) -> None:
    self.__inventario : list[Item] = []
    self.__nombre = nombre

  def mover(self) -> None:
    ... # lógica para mover el personaje

  def saltar(self) -> None:
    ... # lógica para que el personaje salte

  def recoger_item(self, item: Item) -> None:
    self.__inventario.append(item)

  def usar_item(self, item: Item) -> None:
      for i in self.__inventario:
        if i == item:
          i.activar_efecto()
          self.__inventario.remove(i)
          print(f"{self.__nombre} usó {i}")
          return # termina el bucle
      print(f"No hay {item} en el inventario")

  def mostrar_inventario(self) -> None:
    print(f"INVENTARIO de {self.__nombre}:")
    cont = 1
    for i in self.__inventario:
      print(f"{cont}) {i}")
      cont += 1
    print()

if __name__ == "__main__":
  p = Personaje("Mario")
  i = Item("Flor de fuego")
  p.recoger_item(i)
  p.mostrar_inventario()
  p.usar_item(i)
```

En este ejemplo, la clase `Personaje` tiene 2 responsabilidades, la de coordinar todo lo referido a un personaje y manejar el inventario. Lo correcto es separar esa *responsabilidad*, encapsulando el inventario en una nueva clase:

Programa 2: Aplicando el SRP

```python
class Item:
  # ...

class Inventario:
  def __init__(self) -> None:
    self.__items: list[Item] = []

  def agregar(self, item: Item) -> None:
    self.__items.append(item)

  def usar(self, item: Item) -> bool:
    for i in self.__items:
      if i == item:
        i.activar_efecto()
        self.__items.remove(i)
        return True
    return False

  def listar(self) -> None:
    cont = 1
    for i in self.__items:
      print(f"{cont}) {i}")
      cont += 1
    print()

class Personaje:
  def __init__(self, nombre: str) -> None:
    self.__inventario = Inventario()
    self.__nombre = nombre

  def mover(self) -> None:
    ... # lógica para mover el persanaje

  def saltar(self) -> None:
    ... # lógica para que el personaje salte

  def recoger_item(self, item: Item) -> None:
    self.__inventario.agregar(item)

  def usar_item(self, item: Item) -> None:
    if self.__inventario.usar(item):
      print(f"{self.__nombre} usó {i}")
    else:
      print(f"No hay {item} en el inventario")

```

```
46    def mostrar_inventario(self) -> None:
47      print(f"INVENTARIO de {self.__nombre}:")
48      self.__inventario.listar()
49
50  if __name__ == "__main__":
51    p = Personaje("Mario")
52    i = Item("Flor de fuego")
53    p.recoger_item(i)
54    p.mostrar_inventario()
55    p.usar_item(i)
```

Debe notar que la interfaz pública del objeto `p` (lo que está en el *"bloque main"*) no ha cambiado, pero ahora, la implementación del inventario fue *delegada* a una clase particular, cuya implementación puede cambiar o no, logrando un código más fácil de leer, mantener o reutilizar. Cada una de las clases tiene su responsabilidad bien separada, `Inventario` a manejar todo lo relacionado a los items, y `Personaje` a orquestar las acciones del personaje.

Aplicar el SRP aquí tiene varios beneficios:

- **Facilita el mantenimiento:** Al tener responsabilidades separadas, los cambios en el manejo del inventario no afectarán la lógica del personaje.
- **Fomenta la reutilización:** La clase `Inventario` podría reutilizarse en otros personajes u objetos que necesiten manejar items.
- **Mejora la legibilidad:** El código está mejor organizado, lo que facilita su comprensión y reduce la probabilidad de errores.

3. Principio abierto/cerrado

Este principio dice que **el comportamiento de una clase debe estar abierto a la extensión, pero cerrado a la modificación**. Esto indica que debe ser posible extender la funcionalidad sin modificar el código existente. Es por ello que en el siguiente ejemplo este principio no se cumple.

Programa 3: Violación del OCP

```python
class Hechicero:
  def __init__(self, nombre: str) -> None:
    self.__nombre = nombre

  def lanzar_hechizo(self, hechizo: str) -> None:
    if hechizo == "curación":
      print(f"{self.__nombre} cura las heridas del grupo")
    elif hechizo == "hielo":
      print(f"{self.__nombre} lanza un bloque de hielo gelido")
    elif hechizo == "fuego":
      print(f"{self.__nombre} lanza una bola de fuego")
    elif hechizo == "invisibilidad":
      print(f"{self.__nombre} se vuelve invisible")
    else:
      print(f"{self.__nombre} no puede lanzar {hechizo}...")

if __name__ == "__main__":
  h = Hechicero("Merlín")
  h.lanzar_hechizo("curación")
  h.lanzar_hechizo("hielo")
  h.lanzar_hechizo("fuego")
  h.lanzar_hechizo("invisibilidad")
  h.lanzar_hechizo("petrificación")
```

En el ejemplo el `Hechicero` utiliza un *string* para identificar qué clase de hechizo debe tirar y cómo, lo cual lo vuelve muy rígido, ya que si se quisiera agregar o quitar algún hechizo de los posibles, debemos ir a la clase y modificar el método `lanzar_hechizo()` cada vez.

La solución a este problema es crear una jerarquía de clases que implementen una interfaz pública común y aprovechar el **polimorfismo** visto en el apunte anterior.

Programa 4: Aplicando el OCP

```python
from abc import ABC, abstractmethod

class Hechizo(ABC):
  @abstractmethod
  def lanzar(self, hechizero: str) -> None:
    ...

class Curacion(Hechizo):
  def lanzar(self, hechizero: str) -> None:
    print(f"{hechizero} cura las heridas del grupo")

class BloqueDeHielo(Hechizo):
  def lanzar(self, hechizero: str) -> None:
    print(f"{hechizero} lanza un bloque de hielo gelido")
```

```
15
16 class BolaDeFuego(Hechizo):
17   def lanzar(self, hechizero: str) -> None:
18     print(f"{hechizero} lanza una bola de fuego")
19
20 class Invisibilidad(Hechizo):
21   def lanzar(self, hechizero: str) -> None:
22     print(f"{hechizero} se vuelve invisible")
23
24 class Hechicero:
25   def __init__(self, nombre: str) -> None:
26     self.__nombre = nombre
27
28   def lanzar_hechizo(self, hechizo: Hechizo) -> None:
29     hechizo.lanzar(self.__nombre)
30
31
32 if __name__ == "__main__":
33   h = Hechicero("Merlín")
34   h.lanzar_hechizo(Curacion())
35   h.lanzar_hechizo(BloqueDeHielo())
36   h.lanzar_hechizo(BolaDeFuego())
37   h.lanzar_hechizo(Invisibilidad())
```

Ahora, aprovechando este mecanismo es posible agregar el hechizo de petrificación simplemente añadiendo una nueva clase a la jerarquía, sin la necesidad de modificar nada del código ya existente:

```
class Petrificacion(Hechizo):
  def lanzar(self, hechizero: str) -> None:
    print(f"{hechizero} petrifica al enemigo")
```

4. Principio de sustitución de Liskov

El principio de sustitución de Liskov (LSP) fue presentado por Barbara Liskov en una conferencia de OOPSLA en 1987. Desde entonces, este principio ha sido una parte fundamental de la programación orientada a objetos. El principio establece que **los objetos de las subclases deben poder sustituir a las instancias de las superclases** sin alterar el correcto funcionamiento del programa.

Programa 5: Violación del LSP

```python
class Musico:
  def __init__(self, instrumento: str) -> None:
    self.__instrumento = instrumento

  def tocar_instrumento(self) -> None:
    print(f"Músico tocando su {self.__instrumento}")

  def componer_musica(self) -> None:
    print(f"Músico compone con su {self.__instrumento}")

class Guitarrista(Musico):
  def __init__(self) -> None:
    super().__init__("guitarra")

class Violinista(Musico):
  def __init__(self) -> None:
    super().__init__("violín")

class DJ(Musico):
  def __init__(self) -> None:
    super().__init__("consola mezcladora")

  def componer_musica(self) -> None:
    raise NotImplementedError("¡Error!Los DJ no componen música")

if __name__ == "__main__":
  m: Musico = Guitarrista()
  m.tocar_instrumento()
  m.componer_musica()

  m = Violinista()
  m.tocar_instrumento()
  m.componer_musica()

  m = DJ()
  m.tocar_instrumento()
  m.componer_musica()
```

Aquí el problema está centrado en que la subclase `DJ` tiene menos funcionalidad que la clase base, algo que según este principio no debería ocurrir. En este caso, se *levanta* un **error/excepción** que interrumpe la ejecución del programa.

Una manera de corregir el código anterior es reconocer que hay dos clases de músicos: los que pueden tocar instrumentos y componer y los que solo pueden tocar. Entonces quedaría de la siguiente manera.

Programa 6: Aplicando el LSP

```python
class MusicoInstrumentista:
    def __init__(self, instrumento: str) -> None:
        self.__instrumento = instrumento

    def tocar_instrumento(self) -> None:
        print(f"Músico tocando su {self.__instrumento}")

class MusicoCompositor:
    def __init__(self, instrumento: str) -> None:
        self.__instrumento = instrumento

    def tocar_instrumento(self) -> None:
        print(f"Músico tocando su {self.__instrumento}")

    def componer_musica(self) -> None:
        print(f"Músico compone con su {self.__instrumento}")

class Guitarrista(MusicoCompositor):
    def __init__(self) -> None:
        super().__init__("guitarra")

class Violinista(MusicoCompositor):
    def __init__(self) -> None:
        super().__init__("violín")

class DJ(MusicoInstrumentista):
    def __init__(self) -> None:
        super().__init__("consola mezcladora")

if __name__ == "__main__":
    mc: MusicoCompositor = Guitarrista()
    mc.tocar_instrumento()
    mc.componer_musica()

    mc = Violinista()
    mc.tocar_instrumento()
    mc.componer_musica()

    mi: MusicoInstrumentista = DJ()
    mi.tocar_instrumento()
    # mi.componer_musica() # error, no existe el método
```

Resumiendo, este principio establece que todas las subclases deben tener el comportamiento que los clientes esperan de la superclase. Por ello, como se afirma en un afiche muy conocido: *"Si tiene el aspecto de un pato y se oye como un pato, pero usa pilas, probablemente considerarlo un pato sea una abstracción equivocada."*[3]

[3] A partir de esta frase nace en contraposición el *"duck typing"* que dice: *"Si parece un pato y grazna*

5. Principio de segregación de interfaces

Este principio está relacionado con el de responsabilidad única y el de substitución de Liskov que se analizó en la sección anterior. Este indica que **deben crearse pequeñas interfaces específicas para los clientes en lugar de una general.** Es decir, que los clientes (las subclases) no deben estar obligadas a implementar partes (métodos y propiedades) que no le sean útiles.

Programa 7: Violación del ISP

```python
from abc import ABC, abstractmethod

class EmpleableParaTrabajar(ABC):
    @abstractmethod
    def trabajar(self) -> None:
        ...

    @abstractmethod
    def comer(self) -> None:
        ...

class TrabajadorDiurno(EmpleableParaTrabajar):
    def trabajar(self) -> None:
        print("Trabaja de día...")

    def comer(self) -> None:
        print("Hace una pausa para almorzar...")

class TrabajadorNocturno(EmpleableParaTrabajar):
    def trabajar(self) -> None:
        print("Trabaja de noche...")

    def comer(self) -> None:
        print("Hace una pausa para cenar...")

if __name__ == "__main__":
    td: EmpleableParaTrabajar = TrabajadorDiurno()
    td.trabajar()
    td.comer()

    tn: EmpleableParaTrabajar = TrabajadorNocturno()
    tn.trabajar()
    tn.comer()
```

¿Qué ocurriría si se agregara una nueva clase `Robot`? Si ésta implementara la interfaz `EmpleableParaTrabajar`, estaría obligada a implementar el método `comer`, lo cual no tiene

como un pato, es un pato" (sin importar si lleva pilas o no, si se ajusta a la situación, alcanza).

ningún sentido.

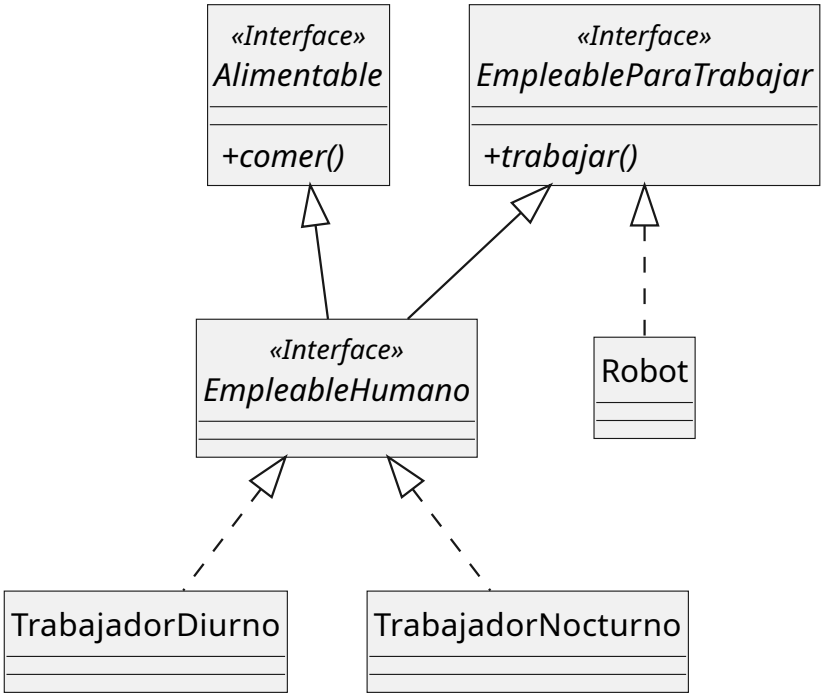
Para corregir el código anterior, es necesario dividir `EmpleableParaTrabajar`, haciendo que `comer` sea un método de la interfaz `Alimentable`. De forma análoga, si los robots debieran recargarse, el método recargar debería declararse en una interfaz nueva (por ejemplo, `Recargable`).

Programa 8: Aplicando el ISP

```python
from abc import ABC, abstractmethod
class EmpleableParaTrabajar(ABC):
  @abstractmethod
  def trabajar(self) -> None:
    ...

class Alimentable(ABC):
  @abstractmethod
  def comer(self) -> None:
    ...

# Interfaz que hereda de otras 2 interfaces...
class EmpleableHumano(Alimentable, EmpleableParaTrabajar):
  pass

class TrabajadorDiurno(EmpleableHumano):
  def trabajar(self) -> None:
    print("Trabaja de día...")

  def comer(self) -> None:
    print("Hace una pausa para almorzar...")

class TrabajadorNocturno(EmpleableHumano):
  def trabajar(self) -> None:
    print("Trabaja de noche...")

  def comer(self) -> None:
    print("Hace una pausa para cenar...")

class Robot(EmpleableParaTrabajar):
  def trabajar(self) -> None:
    print("Trabaja todo el día sin descanso...")

if __name__ == "__main__":
  td: EmpleableHumano = TrabajadorDiurno()
  td.trabajar()
  td.comer()

  tn: EmpleableHumano = TrabajadorNocturno()
  tn.trabajar()
  tn.comer()
```

```
42
43      tr: EmpleableParaTrabajar = Robot()
44      tr.trabajar()
```

De esta manera, cada clase tiene el comportamiento deseado. Note, además, que aquí se aplica un concepto no visto hasta el momento. **Las interfaces pueden heredar de otras interfaces** (línea llena con punta en triangulo), combinando sus funcionalidades y agregando nuevas si así se necesitara. En este caso, simplemente combina dos interfaces, sin agregar nada, por eso se utiliza la palabra reservada `pass` para indicar que la clase no tiene *cuerpo*. El diagrama de clases de este ejemplo se ve de la siguiente manera.



6. Principio de inversión de dependencias

Finalmente el principio de inversión de dependencias dice que **las clases deben depender de abstracciones, no de implementaciones concretas**. Esto puede sonar un poco extraño al principio, pero es lo más común al utilizar polimorfismo. Suponga el siguiente programa.

Programa 9: Violación del DIP

```python
class Operario:
  def trabajar(self) -> None:
    print("Operario trabajando...")

class Gerente:
  def __init__(self) -> None:
    self.__operario_a_cargo : Operario

  def asignar_operario(self, operario: Operario) -> None:
    self.__operario_a_cargo = operario

  def trabjar(self) -> None:
    print("Gerente dando ordenes...")
    self.__operario_a_cargo.trabajar()

if __name__ == "__main__":
  op = Operario()
  ge = Gerente()

  ge.asignar_operario(op)
  ge.trabjar()
```

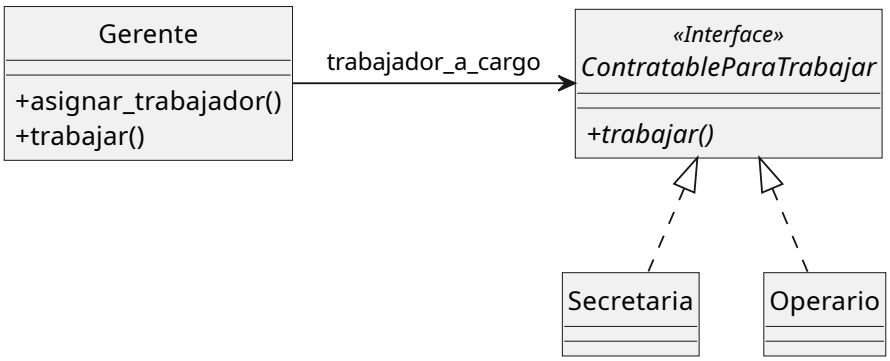
En este programa se viola el principio porque el gerente depende de la clase concreta operario. Por lo tanto, si ahora quisiera darle ordenes a otro tipo de trabajador, por ejemplo, un secretario. No podría, ya que solo admite operarios. La solución es tener una interfaz común para todos aquellos objetos que puedan trabajar para la clase `Gerente`.

Programa 10: Aplicando el DIP

```python
from abc import ABC, abstractmethod

class ContratableParaTrabjar(ABC):
  @abstractmethod
  def trabajar(self) -> None:
    ...

class Operario(ContratableParaTrabjar):
  def trabajar(self) -> None:
    print("Operario trabajando...")

class Secretario(ContratableParaTrabjar):
  def trabajar(self) -> None:
    print("Secretario trabajando...")

class Gerente:
  def __init__(self) -> None:
    self.__trabajador_a_cargo : ContratableParaTrabjar

```

```
20    def asignar_trabajador(self, t:ContratableParaTrabjar) -> None:
21      self.__trabajador_a_cargo = t
22
23    def trabjar(self) -> None:
24      print("Gerente dando ordenes...")
25      self.__trabajador_a_cargo.trabajar()
26
27  if __name__ == "__main__":
28    op = Operario()
29    se = Secretario()
30    ge = Gerente()
31
32    ge.asignar_trabajador(op)
33    ge.trabjar()
34    ge.asignar_trabajador(se)
35    ge.trabjar()
```

En el diagrama de clases queda claro que ahora Gerente depende de la interfaz `ContratableParaTrabajar` y no de las clases concretas `Operario` y `Secretario`.



7. Otros principios populares

Además de los principios **SOLID**, existen otros que son muy conocidos, y valen la pena tenerlos presentes a la hora de diseñar sistemas orientados a objetos.

7.1. KISS

Del acrónimo en inglés *Keep It Simple, S...* (digamos *Student*) que significa que deben mantenerse las cosas simples. Hace alusión a que las soluciones simples son, muchas veces, las más eficientes y mejores. Y no aplicar algoritmos extremadamente complejos cuando no son necesarios.

7.2. DRY

Del acrónimos en inglés *Don't Repeat Yourself* (no te repitas a ti mismo). Es un principio básico que dice que no se deben tener partes de código repetidas o algoritmos que difieran en una mínima parte. Lo mejor es tener los algoritmos bien definidos en una sola parte.

7.3. YAGNI

Del acrónimo en inglés *You Aren't Gonna Need It* (no vas a necesitarlo). Lo que indica este principio es que no debe *adelantarse funcionalidades* del sistema hasta que no sean necesarias. Es decir, no hacer código *"por las dudas"*. Esto viene de la mano muchas veces con técnicas de desarrollo como **TDD** (*Test-Driven Development*) donde se va escribiendo código a medida que es necesario con lo mínimo e indispensable mientras se testea.

El objetivo de todas estas guías o principios es escribir código que esté bien organizado, sea flexible, mantenible y escalable. Poder cumplir con todos ellos al mismo tiempo es una tarea extremadamente compleja y lleva varios años de práctica y diseño para lograr software de alta calidad.

8. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial: `https://docs.python.org/3/library/typing.html`
- **SOLID**: `https://realpython.com/solid-principles-python/`
- KISS, DRY y YAGNI: `https://blog.ahierro.es/principios-kiss-dry-y-yagni/`
- Interprete Visual de Python: `http://www.pythontutor.com/visualize.html`