Asignación de responsabilidades

Johnathan Calle, PhD.

Principios de diseño orientado a objetos

- El código debe poseer las siguientes cualidades:
  - Mantenibilidad
  - Extensibilidad
  - Modularidad

Principios de diseño orientado a objetos

- El código debe poseer las siguientes cualidades:
  - Mantenibilidad
  - Extensibilidad
  - Modularidad
- ¿Cómo se determina que el código posee estas características?
  - El tiempo lo dirá…
  - Si se mantiene bien en términos de facilidad de mantenimiento, si ha crecido y no ha sido muy crítico su cambio y si sigue fácil de modularizar → se habrá hecho un buen trabajo de diseño

Principios de diseño orientado a objetos

- El código debe poseer las siguientes cualidades:
  - Mantenibilidad
  - Extensibilidad
  - Modularidad
- ¿Cómo se determina que el código posee estas características?
  - El tiempo lo dirá…
  - Si se mantiene bien en términos de facilidad de mantenimiento, si ha crecido y no ha sido muy crítico su cambio y si sigue fácil de modularizar → se habrá hecho un buen trabajo de diseño
  - Estas características empiezan a fallar después de cierto tiempo o de la aparición de ciertos eventos → cambios en los requisitos

GRASP: *General Responsibility Assignment Software Patterns*

- **Information Expert**
  - Ayuda a los diseñadores a asignar, de manera adecuada, las responsabilidades a las clases
  - Define que se debe asignar una responsabilidad a la clase que tiene la **información necesaria** para suplir la responsabilidad
  - **Problema:** cuál es un principio básico por el cual se le debe asignar responsabilidades a un objeto?
  - **Solución:** se le deba asignar una responsabilidad a la clase que tiene la información para suplir la instanciación

GRASP: *General Responsibility Assignment Software Patterns*

- **Information Expert**
  - Ayuda a los diseñadores a asignar, de manera adecuada, las responsabilidades a las clases
  - Define que se debe asignar una responsabilidad a la clase que tiene la **información necesaria** para suplir la responsabilidad

  - **Dónde estaría mejor ubicado el método “obtener_ordenes()”?**

```
class Cliente:
  def __init__(self,id,nombre):
    self.id = id
    self.nombre = nombre
    self.ordenes = []

class Orden:
  def __init__(self,id,total,cliente):
    self.id = id
    self.total = total
    self.cliente = cliente
```

GRASP: *General Responsibility Assignment Software Patterns*

- **Information Expert**
  - Ayuda a los diseñadores a asignar, de manera adecuada, las responsabilidades a las clases
  - Define que se debe asignar una responsabilidad a la clase que tiene la **información necesaria** para suplir la responsabilidad

  - Dónde estaría mejor ubicado el método “obtener_ordenes()”?
  - **y “obtener_total_orden()”?**

```
class Cliente:
  def __init__(self,id,nombre):
    self.id = id
    self.nombre = nombre
    self.ordenes = []

class Orden:
  def __init__(self,id,total,cliente):
    self.id = id
    self.total = total
    self.cliente = cliente
```

GRASP: *General Responsibility Assignment Software Patterns*

- *Creator*
  - Ayuda a decidir qué clase debería ser responsable de crear una nueva instancia de una clase
  - **Problema:** quién crea un objeto de tipo A?
  - **Solución**: una clase B debería tener la responsabilidad de crear un objeto de la clase A si se cumple alguna de las siguientes condiciones (entre más mejor):
    - B contiene a A (Relación de composición)
    - B agrega a A (relación de agregación)
    - B tiene data de inicialización para A
    - B usa a A de manera muy estrecha

GRASP: *General Responsibility Assignment Software Patterns*

- *Creator*
  - **¿Cuáles condiciones cumple?**
    - B contiene a A (Relación de composición)
    - B agrega a A (relación de agregación)
    - B tiene data de inicialización para A
    - B usa a A de manera muy estrecha

```python
class Cliente:
  def __init__(self,id,nombre):
    self.id = id
    self.nombre = nombre
    self.carrito = []
    self.ordenes = []
  def consolidar_orden(self):
    total = 0
    for producto in carrito:
      total += producto.valor
    orden = Orden(total, self) #está bien?
class Orden:
  def __init__(self,total,cliente):
    self.total = total
    self.cliente = cliente
class Producto:
  pass
```

GRASP: *General Responsibility Assignment Software Patterns*

- *Creator*
  - **¿Seguimos bien?**
    - B contiene a A (Relación de composición)
    - B agrega a A (relación de agregación)
    - B tiene data de inicialización para A
    - B usa a A de manera muy estrecha

```python
class Cliente:
  def __init__(self,id,nombre):
    self.id = id
    self.nombre = nombre
    self.carrito = []
    self.ordenes = []
  def consolidar_orden(self):
    total = 0
    for producto in carrito:
      total += producto.valor
    orden = Orden(total, self) #Qué falta?
class Orden:
  def __init__(self,id, total,cliente):
    self.id = id
    self.total = total
    self.cliente = cliente
class Producto:
  pass
```

GRASP: *General Responsibility Assignment Software Patterns*

- *Controller*
  - **Problema**: qué objeto después de la capa de interfaz de usuario recibe y coordina (controla) las operaciones del sistema
  - **Solución**: asignar la responsabilidad a un objeto que represente una de estas opciones:
    - Representa todo el sistema
    - Un objeto raíz
    - Un subsistema mayor

GRASP: *General Responsibility Assignment Software Patterns*

- *Controller*
  - **Problema**: qué objeto después de la capa de interfaz de usuario recibe y coordina (controla) las operaciones del sistema
  - **Solución**: asignar la responsabilidad a un objeto que represente una de estas opciones:
    - Representa todo el sistema
    - Un objeto raíz
    - Un subsistema mayor

```
class GUI:
  def obtener_input():
    pass
class Juego: #clase principal
  def __init__(self):
    self.GUI = GUI()
  def process_input():
    input_data = self.GUI.obtener_input()
    #define qué se hace con el tipo de información
    #que recibe --> orquestra el sistema completo
```

GRASP: *General Responsibility Assignment Software Patterns*

- *Low coupling*
  - **Problema**: ¿cómo reducir el impacto del cambio en el código? ¿Cómo fomentar la baja dependencia e incrementar el reuso?
  - **Solución**: asignar responsabilidades tal que el acoplamiento se mantenga bajo
  - **Acoplamiento**: es una medida de cómo un elemento se relaciona con otro. A mayor acoplamiento, mayor dependencia de un elemento con otro.
  - Un bajo acoplamiento indica que los objetos son más independientes y aislados
  - Si algo está aislado, es más fácil de cambiar sin preocuparse porque los cambios afectarán a otras clases (que haremos un daño en otras secciones del código)

GRASP: *General Responsibility Assignment Software Patterns*

- *High cohesion*
  - **Problema**: ¿cómo mantenemos a las clases enfocadas, entendibles y manejables como un efecto secundario de mantener bajo acoplamiento?
  - **Solución**: asignar una responsabilidad tal que la cohesión se mantenga alta
  - **Cohesión**: es una medida de qué tan fuertemente se relacionan las responsabilidades de una clase.
  - Las clases con baja cohesión tienen datos (atributos) o comportamiento (métodos) que no se relacionan entre ellos
  - Las clases con alta cohesión se enfocan en responsabilidades específicas y sólo abarcan los datos y comportamiento necesarios para llevarlas a cabo

GRASP: *General Responsibility Assignment Software Patterns*

- *Indirection*
  - **Problema**: ¿dónde asignar una responsabilidad para evitar acoplamiento directo entre dos o más clases?
  - **Solución**: asignar la responsabilidad a un objeto intermedio que sirva de mediador entre otros componentes o servicios, haciendo que estos componentes ya no estén directamente acoplados (mitigar la dependencia)
  - **Cohesión**: es una medida de qué tan fuertemente se relacionan las responsabilidades de una clase.
  - Las clases con baja cohesión tienen datos (atributos) o comportamiento (métodos) que no se relacionan entre ellos
  - Las clases con alta cohesión se enfocan en responsabilidades específicas y sólo abarcan los datos y comportamiento necesarios para llevarlas a cabo
  - A tener en cuenta: usualmente este principio reduce la facilidad de lectura del código ya que no es muy claro quién tiene el control de ejecución entre las clases “mediadas”

GRASP: *General Responsibility Assignment Software Patterns*

- *Indirection*
  - **Problema**: ¿dónde asignar una responsabilidad para evitar acoplamiento directo entre dos o más clases?
  - **Solución**: asignar la responsabilidad a un objeto intermedio que sirva de mediador entre otros componentes o servicios, haciendo que estos componentes ya no estén directamente acoplados (mitigar la dependencia)
  - **Cohesión**: es una medida de qué tan fuertemente se relacionan las responsabilidades de una clase.
  - Las clases con baja cohesión tienen datos (atributos) o comportamiento (métodos) que no se relacionan entre ellos
  - Las clases con alta cohesión se enfocan en responsabilidades específicas y sólo abarcan los datos y comportamiento necesarios para llevarlas a cabo
  - A tener en cuenta: usualmente este principio reduce la facilidad de lectura del código ya que no es muy claro quién tiene el control de ejecución entre las clases “mediadas”
- RETO: ejemplo del patrón de diseño *Mediator* (0.1 para los 3 primeros en la próxima clase)
- **NOTA: TODOS los retos deben ser resueltos en el contexto de la universidad, es decir, clases como estudiante, profesor, biblioteca, cafetería, etc.**

GRASP: *General Responsibility Assignment Software Patterns*

- *Polymorphism*
  - **Problema**: ¿cómo manejar alternativas basado en el tipo (clase)?
  - **Solución**: cuando las alternativas varían con base en su tipo (clase), se debe asignar la responsabilidad al comportamiento de acuerdo a la variación específica.
  - **Lo exploraremos más adelante**
- RETO: ejemplo de polimorfismo (0.1 para los 3 primeros en la próxima clase)

GRASP: *General Responsibility Assignment Software Patterns*

- *Pure Fabrication*
  - **Problema**: ¿qué objeto debería tener la responsabilidad cuando no se quiere violar el principio de alta cohesión y bajo acoplamiento pero los otros principios no proveen una solución apropiada?
  - **Solución**: asignar un conjunto de responsabilidades altamente cohesivas a una clase artificial o de conveniencia que no represente un problema conceptual a nivel de dominio
  - **A tener en cuenta:** a veces es muy complicado determinar dónde deben ir asignadas las responsabilidades
- **Lo exploraremos más adelante**

GRASP: *General Responsibility Assignment Software Patterns*

- *Protected variations*
  - **Problema**: ¿cómo diseñar objetos, subsistemas y sistemas tal que las variaciones o la inestabilidad en estos elementos no tenga un impacto negativo en el resto del sistema?
  - **Solución**: se deben conocer muy bien estos puntos de variación o inestabilidad y crear interfaces estables alrededor de estos
  - **A tener en cuenta:** este proceso es altamente delicado y complejo
- **Lo exploraremos sólo en términos generales con otros conceptos más adelante**
- **Otros temas interesantes a revisar**
  - **Principios SOLID**
  - **Encapsulamiento**
  - **Virtualización y contenedores**
  - **Arquitecturas basadas en eventos**
  - **Orquestración**

SOLID: los primeros 5 principios del diseño orientado a objetos

- Establecidos con base en la experiencia de múltiples proyectos orientados a objetos por Robert C. Martin (Uncle Bob)
- S: *Single responsibility principle*
- O: *Open-closed principle*
- L: *Liskov substitution principle*
- I: *Interface segregation principle*
- D: *Dependency inversion principle*

SOLID: ***single responsibility principle (SRP)***

- Este principio define que una clase suple sus responsabilidades usando sus funciones o contratos (miembros y funciones de ayuda)

SOLID: *single responsibility principle (SRP)*

- Este principio define que una clase suple sus responsabilidades usando sus funciones o contratos (miembros y funciones de ayuda)

```
class Traductor:
  def identificar_idioma():
    pass
  def traducir_texto():
    pass
  def cargar_texto():
    pass
```

- ¿Cuántas responsabilidades tiene la clase Traductor?

SOLID: *single responsibility principle (SRP)*

- Este principio define que una clase suple sus responsabilidades usando sus funciones o contratos (miembros y funciones de ayuda)

```
class Traductor:
  def identificar_idioma():
    pass
  def traducir_texto():
    pass
  def cargar_texto():
    pass
```

- ¿Cuántas responsabilidades tiene la clase Traductor?
- → 2: traducir (*identificar_idioma* y *traducir_texto*) y cargar el texto a traducir
- **Una responsabilidad puede agrupar varias funciones**

SOLID: *single responsibility principle (SRP)*

- Este principio define que una clase suple sus responsabilidades usando sus funciones o contratos (miembros y funciones de ayuda)
- ¿Mejora esta solución?

```
class Traductor:
    def identificar_idioma():
        pass
    def traducir_texto():
        pass
    def cargar_texto():
        lector = CargaArchivo()
        lector.cargar_texto()

class CargaArchivo:
    def cargar_texto():
        pass
```

SOLID: *single responsibility principle (SRP)*

- **Una clase debe tener sólo una responsabilidad → Será factible?**

SOLID: *single responsibility principle (SRP)*

- **Una clase debe tener sólo una responsabilidad → Será factible?**
- **Cohesión:** en POO la cohesión se refiere a cómo está diseñada una clase específica. Se busca que una clase tenga un propósito específico.
    - Una alta cohesión es deseable ya que las clases serán más fáciles de mantener (y cambiadas menos frecuentemente) que clases con baja cohesión (muchos propósitos).
    - Las clases que tienen alta cohesión son más reusables que otras clases
    - Alta cohesión es cuando una clase hace un trabajo bien definido
    - Baja cohesión es cuando una clase se enfoca en muchas responsabilidades
    - ¿Cómo se mide?

SOLID: *single responsibility principle (SRP)*

- **Acoplamiento (coupling):** indica el nivel de dependencia que tienen las clases.
  - Este principio indica que se debe asignar responsabilidades a las clases siempre y cuando se mantenga una dependencia baja
  - Es deseable un bajo acoplamiento

SOLID: *single responsibility principle (SRP)*

- **Acoplamiento (coupling):** indica el nivel de dependencia que tienen las clases.
  - Este principio indica que se debe asignar responsabilidades a las clases siempre y cuando se mantenga una dependencia baja
  - Es deseable un bajo acoplamiento

- **En el ejemplo anterior obtuvimos una alta cohesión pero aumentamos también la dependencia (acoplamiento) entre las clases. ¿Cuándo paramos entonces?**

```
class Traductor:
    def identificar_idioma():
        pass
    def traducir_texto():
        pass
    def cargar_texto():
        lector = CargaArchivo()
        lector.cargar_texto()

class CargaArchivo:
    def cargar_texto():
        pass
```

S**O**LID: ***open-close principle***

- **Un módulo de software (puede ser una clase o método) debe estar abierto a extensiones pero cerrado para modificaciones**
  - Es decir, no se debería poder actualizar el código que ya se escribió para un proyecto pero sí se pueden añadir nuevas funcionalidades

SOLID: *open-close principle*

```python
class CargaArchivo:
  def cargar_texto():
    pass

class CargaPDF(CargaArchivo):
  def cargar_texto():
    #comportamiento carga pdf
    pass

class Traductor:
  def identificar_idioma():
    pass
  def traducir_texto():
    pass
  def cargar_texto():
    lector = CargaPDF()
    lector.cargar_texto()
```

- En este ejemplo, aparece una nueva clase que hereda de la clase base CargaArchivo. Esta clase se enfoca en la carga de textos en formato pdf.
- ¿Se mantienen las características de alta cohesión y bajo acoplamiento?
- ¿Qué pasa si se quiere leer un formato nuevo?
- ¿Cuáles clases cambian o deberían cambiar?

SOLID: *open-close principle*

```python
class CargaArchivo:
  def cargar_texto():
    pass

class CargaPDF(CargaArchivo):
  def cargar_texto():
    #comportamiento carga pdf
    pass

class Traductor:
  def identificar_idioma():
    pass
  def traducir_texto():
    pass
  def cargar_texto():
    lector = CargaPDF()
    lector.cargar_texto()
```

```python
class CargaArchivo:
  def cargar_texto():
    pass
class CargaPDF(CargaArchivo):
  def cargar_texto():
    #comportamiento carga pdf
    pass
class CargaTxt(CargaArchivo):
  def cargar_texto():
    #comportamiento carga txt
    pass
class Traductor:
  def identificar_idioma():
    pass
  def traducir_texto():
    pass
  def cargar_texto(tipo_texto):
    if(tipo_texto == "pdf"):
      lector = CargaPDF()
      lector.cargar_texto()
    elif(tipo_texto == "txt"):
      lector = CargaTxt()
      lector.cargar_texto()
```

SO**L**ID: ***Liskov Substitution Principle***

- **Las clases derivadas deben ser sustituibles por su clase base**
  - **Lo retomaremos más adelante**
- **RETO:** Implementar un ejemplo con este principio usando Python.
  - (0.1 x 5 primeros en la próxima clase)

SOLID: *Interface segregation principle*

- **Los clientes no deben ser forzados a depender de las interfaces que no usan**
- **RETO:** Implementar un ejemplo con este principio usando Python.
  - (0.1 x 5 primeros en la próxima clase)

SOLI**D:** ***Dependency inversion principle***

- **Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deberían depender de abstracciones**
- **RETO:** Implementar un ejemplo con este principio usando Python.
  - (0.1 x 5 primeros en la próxima clase)