SOLID: Buenas prácticas en el diseño orientado a objetos

The Single Responsibility Principle

Diego Torres

[1]UNQ - diego.torres@unq.edu.ar

[2]UNLP-LIFIA diego.torres@lifia.info.unlp.edu.ar

Contenido

Principios SOLID

- **S** ingle responsibility: Una clase debe tener una única responsabilidad.

Principios SOLID

- **S** ingle responsibility: Una clase debe tener una única responsabilidad.
- **O** pen-closed: Una clase debe estar abierta para la extensión, y cerrada para los cambios en el código.

Principios SOLID

- **S** ingle responsibility: Una clase debe tener una única responsabilidad.
- **O** pen-closed: Una clase debe estar abierta para la extensión, y cerrada para los cambios en el código.
- **L** iskow substitution: Objetos de un programa deberían poder ser cambiados por instancias de subclases sin alterar el correcto funcionamiento de ese programa.

Principios SOLID

- **S** ingle responsibility: Una clase debe tener una única responsabilidad.
- **O** pen-closed: Una clase debe estar abierta para la extensión, y cerrada para los cambios en el código.
- **L** iskow substitution: Objetos de un programa deberían poder ser cambiados por instancias de subclases sin alterar el correcto funcionamiento de ese programa.
- **I** nterface segregation: Varias interfaces bien específicas son mejores que pocas generales.

Principios SOLID

- **S** ingle responsibility: Una clase debe tener una única responsabilidad.
- **O** pen-closed: Una clase debe estar abierta para la extensión, y cerrada para los cambios en el código.
- **L** iskow substitution: Objetos de un programa deberían poder ser cambiados por instancias de subclases sin alterar el correcto funcionamiento de ese programa.
- **I** nterface segregation: Varias interfaces bien específicas son mejores que pocas generales.
- **D** ependency inversion: Depender de abstracciones y no de clases concretas.

El principio de única responsabilidad

The Single Responsibility Principle (SRP)

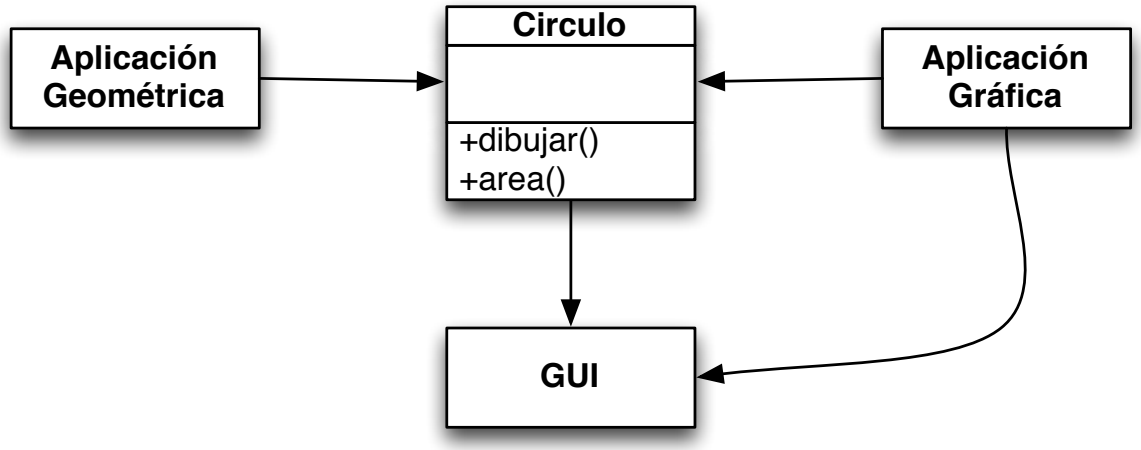


Figura: Clase Círculo

- La aplicación geométrica utiliza solamente el comportamiento relacionado a computar el area del círculo y jamas lo dibujará.
- La Aplicación Gráfica utiliza la opción de dibujar en la interfaz de usuario (GUI) al círculo.

El principio de única responsabilidad

The Single Responsibility Principle (SRP)

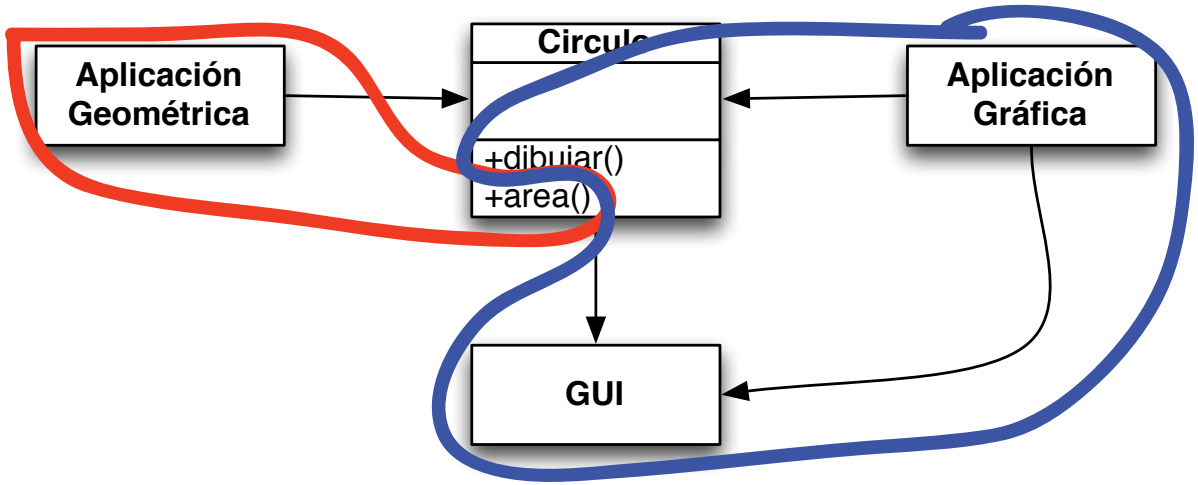


Figura: Clase Círculo

El diseño viola el principio de SRP ya que tiene dos responsabilidades

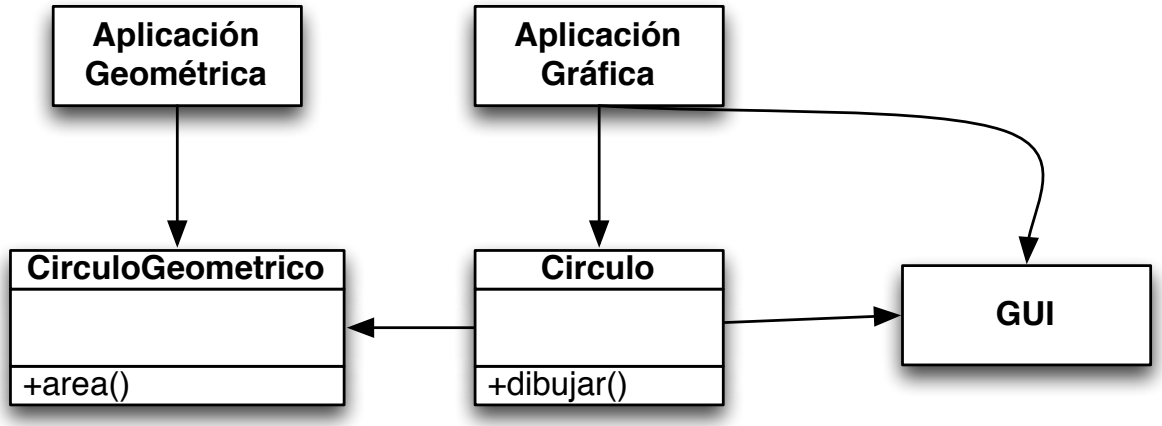
- La primera modela la representación geométrica del círculo
- La segunda es para renderizar el círculo en la interfáz gráfica.

Consecuencias

La violación del SRP genera varios problemas

- debemos incluir la GUI en la aplicacion de cálculo geométrico. En algunos lenguajes, como Java o .net debe sessre ensamblada para poder realizar el deploy con la aplicación geometrica.
- si un cambio en la aplicación Grafica causa que el circulo debe cambiar por alguna razon, entonces el cambio nos obliga a recompilar, re testear, y re-deployar la aplicacion de computación geometrica. Si olvidamos hacer esto, la aplicación puede romperse de formas poco predecibles.

Solución



Un mejor diseño es separar en las responsabilidades en dos clases totalmente diferentes. Este diseño mueve la porción computacional del círculo a la clase CirculoGeometrico. Ahora los cambios sobre la manera en que se dibuja el rectangulo no pueden afectar a la aplicación geométrica.

Definición de Responsabilidad

Responsabilidad

En el contexto de SRP, definimos *responsabilidad* como una razón de cambio. Si pensamos que existe más de una razón que motive el cambio de una clase, entonces esa clase posee más de una responsabilidad.

Ejemplo Modem

```
public interface DSLModem
{
public void Dial(String pno);
public void Hangup();
public void Send(char c);
public char Recv();
}
```

- Funcionalidad de manejo de conexión.
- Funcionalidad de comunicación de datos.

¿Deben dividirse estas responsabilidades?

Dependen de la forma en que la aplicación cambia.

Alternativa

Atención

Si la aplicación cambia de forma que afecta la signatura de las funcionalidades de conexión, el diseño va a ser un tanto rígido porque las clases que invocan read y send van a tener que ser recompiladas y vueltas a *deployar* más seguido que lo querido.

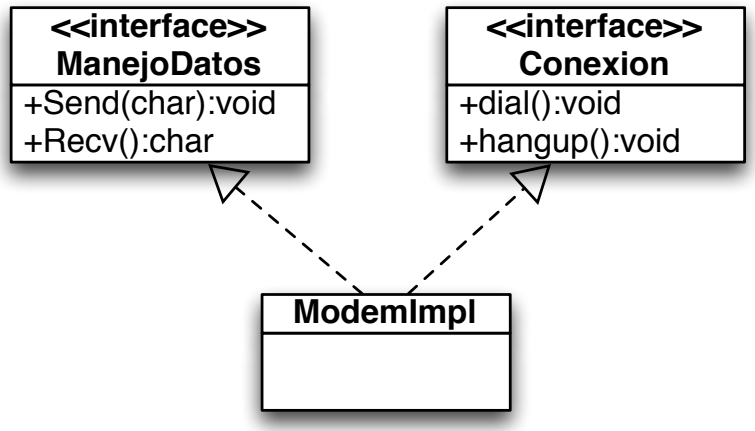


Figura: Alternativa de solución si se cumple lo anterior

Persistencia

El ejemplo típico de violación de SRP es el que muestra la figura. Los cambios de persistencia son muy diferentes a los cambios de la lógica de negocio. La lógica de negocio tiende a cambiar frecuentemente, mientras que la de persistencia no tanto. Y si esta última lo hace, los cambios se producen en una parte diferente a la de la lógica de negocio. Claramente hay dos responsabilidades marcadas.

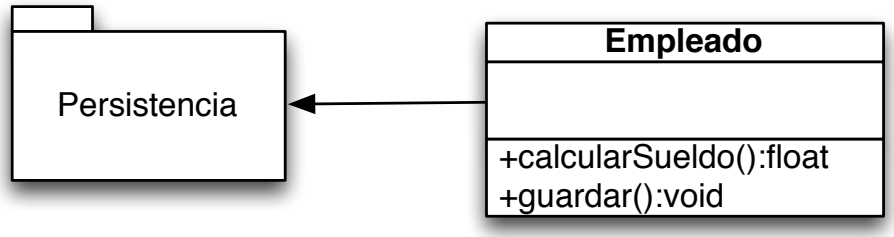


Figura: Violación de SRP en el contexto de la persistencia

Conclusiones

El principio de Única Responsabilidad (SRP) es uno de los principios más simples de comprender pero el más difícil de aplicar. Juntar responsabilidades es algo que realizamos naturalmente. Encontrar y separar esas responsabilidades es algo más relacionado a lo que el diseño de software en verdad realiza. El resto de los principios que discutiremos vuelven de una forma a este.

Principio Abierto - Cerrado

Heuristicas asociadas

- "Todas las variables de instancia deben ser privadas"
- "Deben evitarse las variables globales"
- Hacer uso de identificadores de tipos no es una buena práctica (ej. Variable de Instancia String tipo)

"Todos los sistemas cambian durante su ciclo de vida. Debemos considerar esto para hacer nuestros sistemas lo más durables posible"

— Ivar Jacobson

Las entidades de software

C

lases, módulos, funciones, etc deben ser abiertas para ser extendidas, pero cerradas para su modificación.

Heuristicas

Abiertas para ser extendidas

Significa que el comportamiento se debe poder extender.

Cerradas para la modificación

El código de la clase es inviolable. Nadie puede alterar el código fuente de la misma.

Heuristicas

Abiertas para ser extendidas

Significa que el comportamiento se debe poder extender.

Cerradas para la modificación

El código de la clase es inviolable. Nadie puede alterar el código fuente de la misma.

Parecen dos premisas opuestas. ¿Cómo las llevamos a la práctica ?

Ejemplo del Libro

Figura: Violando OCP

Ejemplo de Libro

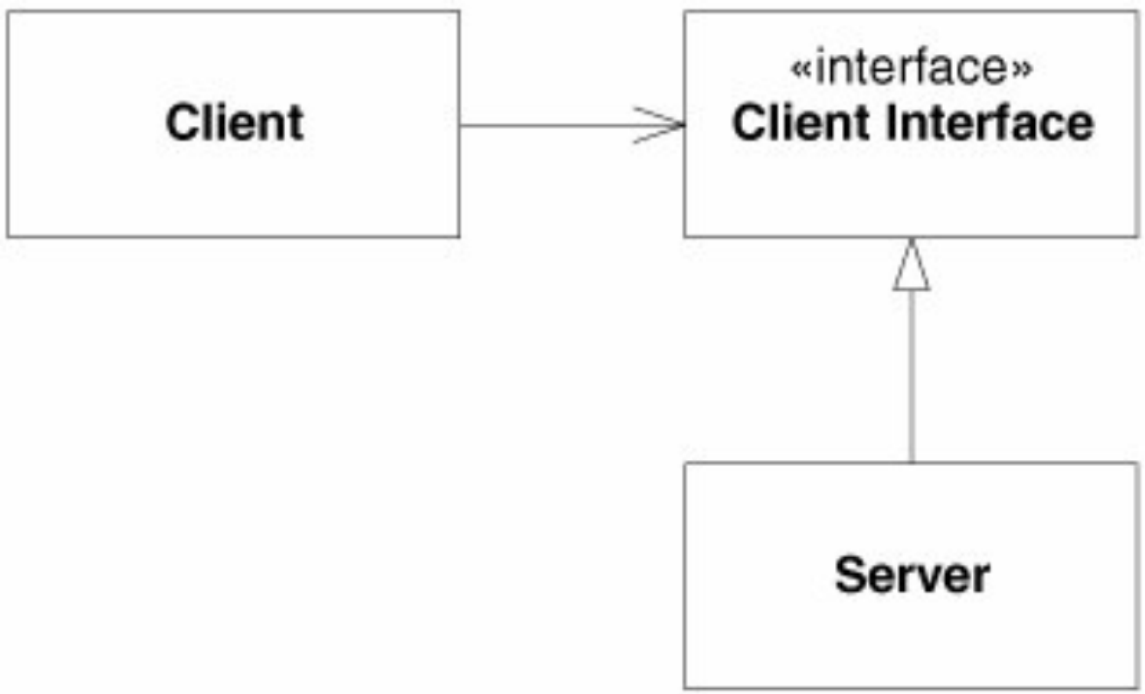


Figura: Mejorando Open Closed en Client

Ejemplo Nuestro

Opción 1)

```
CuentaBancaria>>extraer:unMonto
|rojo|
    self tipo = 'cuentacorriente'
        ifTrue:[rojo:= self rojoPermitido.]
        ifFalse:[rojo := 0.]
  (self saldo + rojo >= unMonto)
        ifTrue: [self saldo: self saldo - unMonto].
```

Figura: ¿Dónde se viola OCP?

Abstracción y más abstracción

Hacer privadas las V.I.

Cuando el valor de una variable cambia, el cambio afecta en todos los métodos que dependen de ese valor. No esta cerrado con respecto a eso. Hacer cumplir el OCP para estos casos se llama ENCAPSULAMIENTO.

No usar variables globales - NUNCA

Equivalente a las vi. Todos los objetos que hagan uso de esas variables globales, van a depender de ese valor.

RTTI - grrrrrrr

Entre todos!