UNIVERSIDAD DE LIMA | 60 años

PRINCIPIOS DEL DISEÑO ORIENTADO A OBJETOS

UNIDAD 2: DISEÑO E IMPLEMENTACIÓN

Temario

- Principios DRY (Don’t Repeat Yourself)
- Principio Law of Demeter
- Principios SOLID

UNIVERSIDAD DE LIMA | 60 años

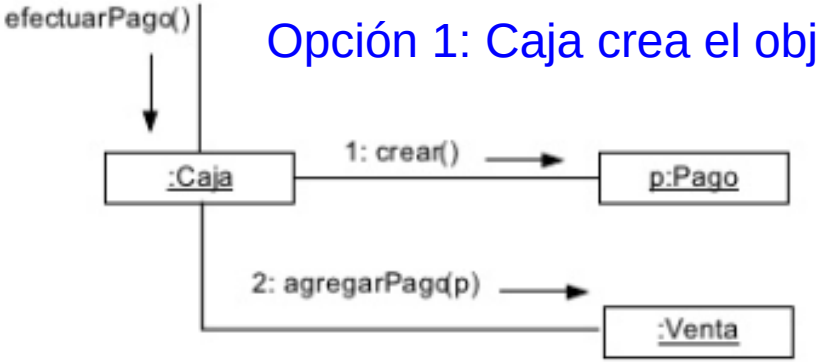
Principios en el diseño orientado a objetos

Un principio de diseño representa una guía altamente recomendada para dar forma a la lógica de la solución de cierta manera y con ciertos objetivos en mente

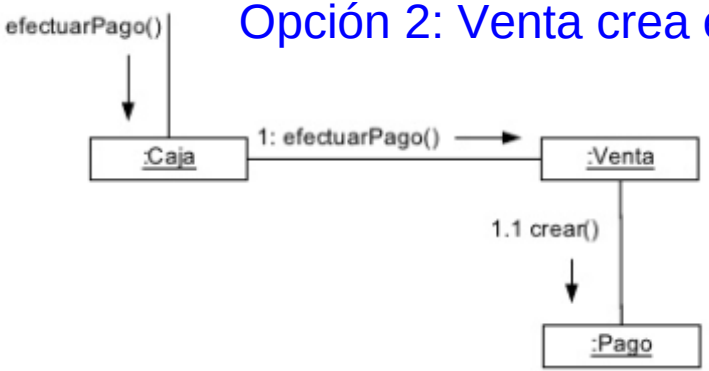
Principio: Bajo acoplamiento

- ¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?
- Una clase con alto (o fuerte acoplamiento) confía en muchas otras clases
  - Los cambios en las clases relacionadas fuerzan cambios locales
  - Son difíciles de entender de manera aislada
  - Son difíciles de reutilizar
- Asignar una sola responsabilidad para que el acoplamiento (innecesario) permanezca bajo.

Principio: Bajo acoplamiento



Opción 1: Caja crea el objeto Pago



Opción 2: Venta crea el objeto Pago

¿Cuál es la propuesta adecuada?

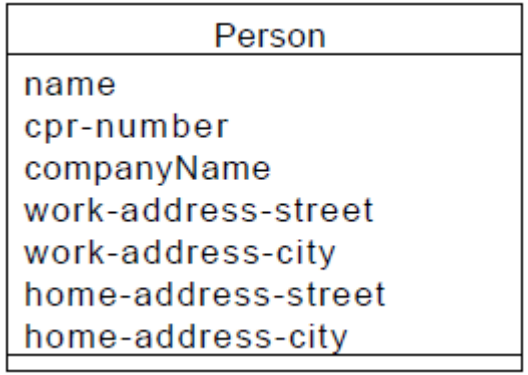
UNIVERSIDAD DE LIMA

Principio: Alta cohesión

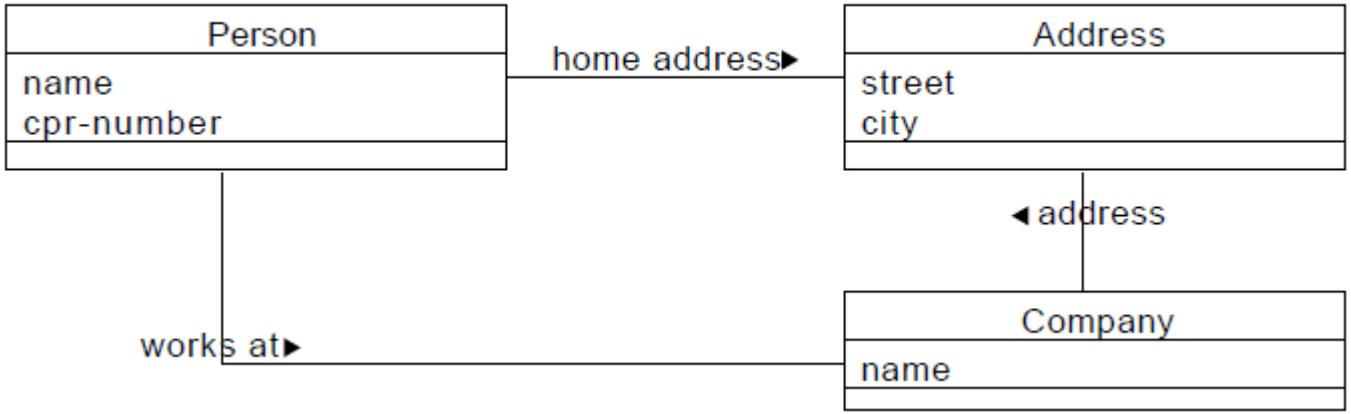
- ¿Cómo mantener la complejidad manejable?
- Asignar una responsabilidad para que la cohesión se mantenga alta.
- Una clase con baja cohesión hace demasiado trabajo:
  - Difíciles de entender
  - Difíciles de reutilizar
  - Difíciles de mantener
  - Delicadas, constantemente afectada por los cambios

¿Cuál tiene mayor cohesión?

UNIVERSIDAD DE LIMA



Baja Cohesión



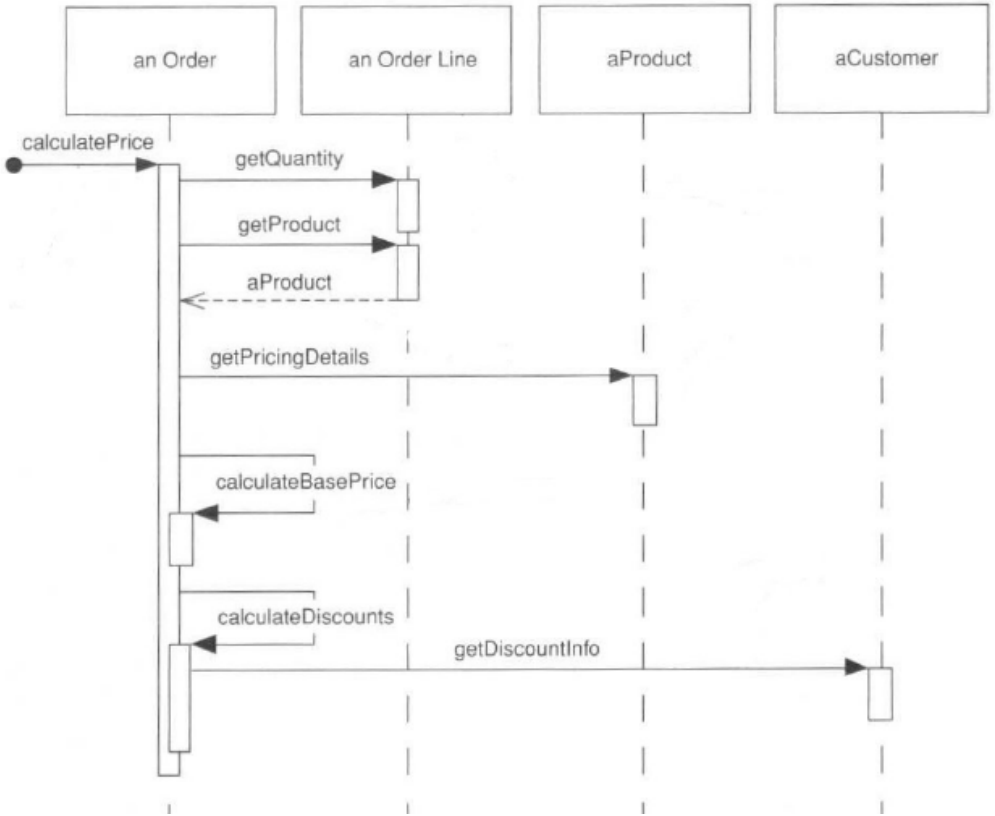
Alta Cohesión

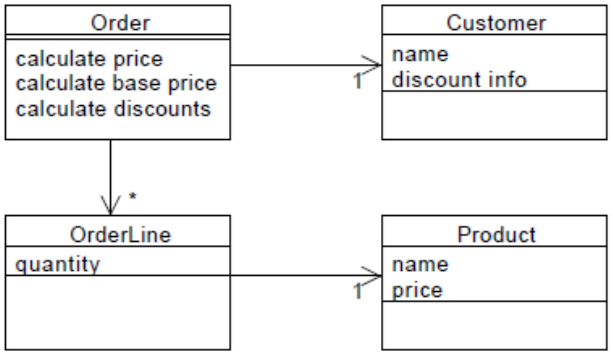
UNIVERSIDAD DE LIMA

Ley de Demeter (Law of Demeter)

- La Ley de Deméter (LoD) es una regla de estilo simple para diseñar sistemas orientados a objetos
- "No hables con extraños, solamente con los que conoces"
- ¿A quién puedo enviar un mensaje?
  1. A un objeto conectado mediante un enlace navegable (instancia de asociación).
  2. A un objeto recibido como parámetro en esta activación.
  3. A un objeto creado localmente en esta ejecución, o variable local
  4. A mí mismo, el emisor del mensaje.
- Paper: Object-Oriented Programming:An Objective Sense of Style

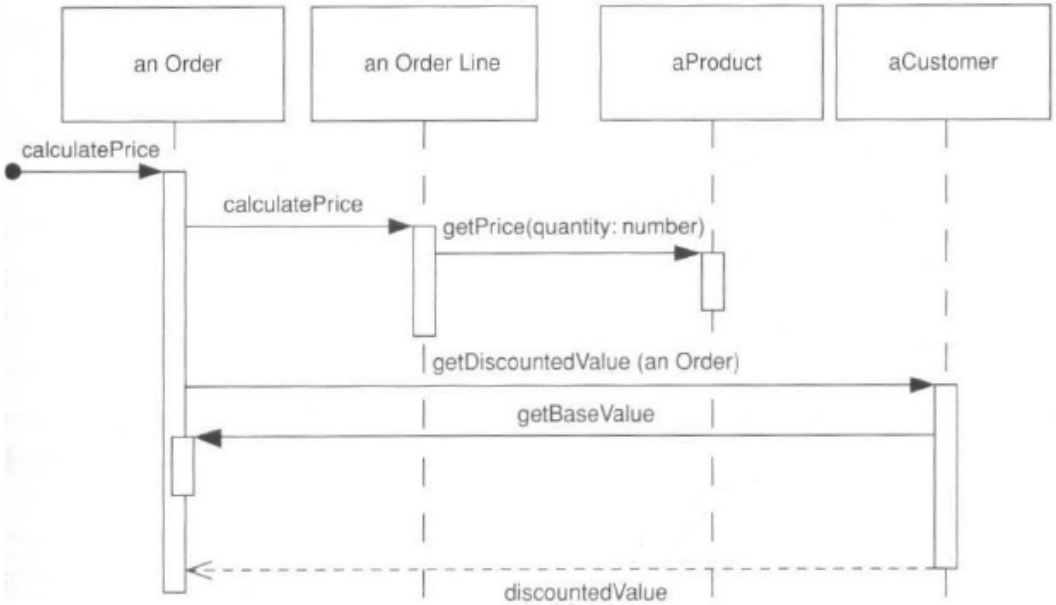
Violación de la Ley de Demeter

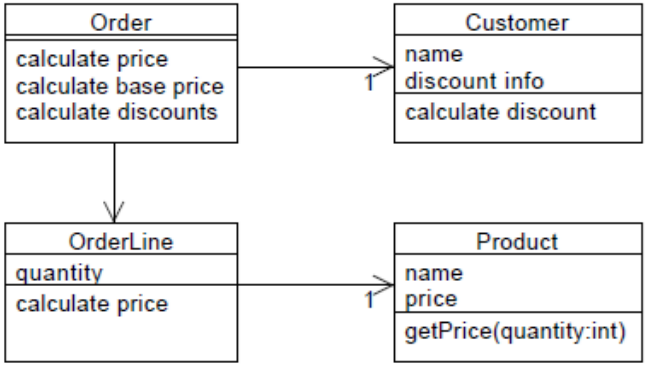




D

Aplicación de la Ley de Demeter





UNIVERSIDAD DE LIMA

Evitar los encadenamientos

```
1  public class DrawEngine
2  {
3     public void Draw(Canvas canvas, Box box)
4     {
5        canvas.SetColor(box.Border.Color);
6        canvas.DrawRectangle(box.Location.X, box.Location.Y,
7                             box.Size.Width, box.Size.Height);
8
9        canvas.SetColor(box.BackColor);
10       canvas.FillRectangle(box.Location.X, box.Location.Y,
11                            box.Size.Width, box.Size.Height);
12    }
13 }
```

Fuente: Ley de Demeter; Tell, Don’t Ask y God Object

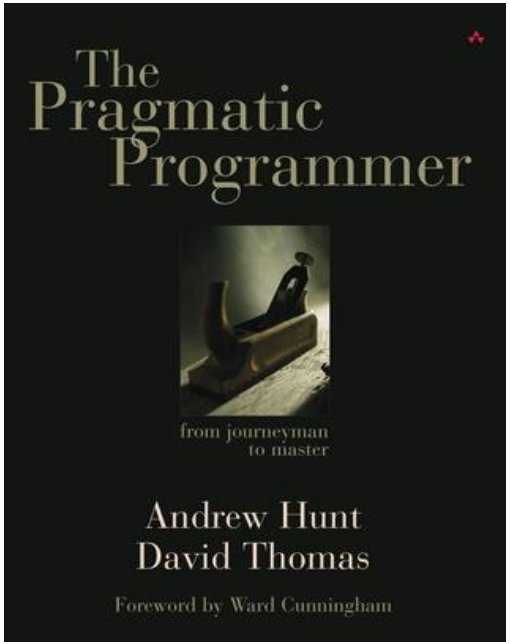
Objetivo: Evitar esto
objectA.getObjectB().doSomething();

Principio DRY

Don’t repeat yourself

- “Todo conocimiento debe tener una representación única, inequívoca y autorizada dentro de un sistema”

Conocimiento: funcionalidad y/o algoritmo



“El conocimiento de un sistema es mucho más amplio que solo su código. Se refiere a esquemas de base de datos, planes de prueba, el sistema de compilación e incluso documentación”.

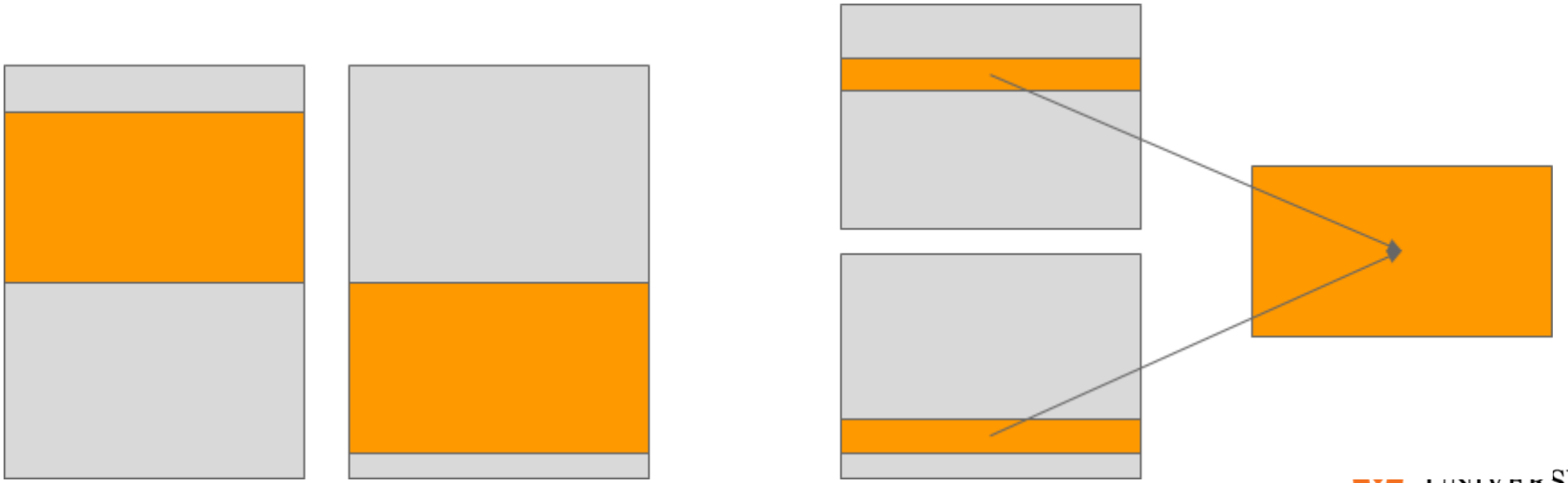
- Dave Thomas -

https://media.pragprog.com/titles/tpp20/dry.pdf

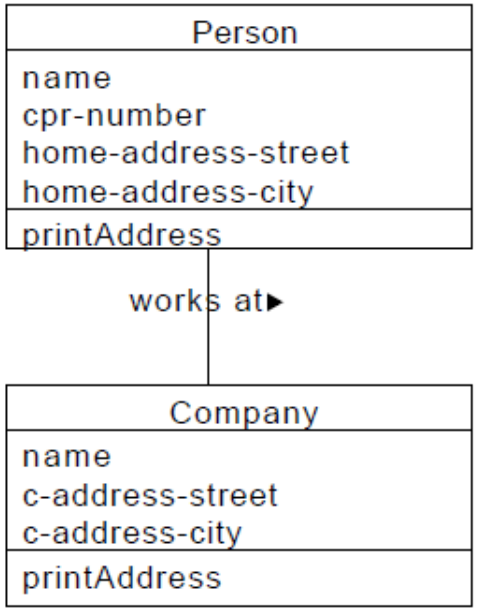
UNIVERSIDAD DE LIMA

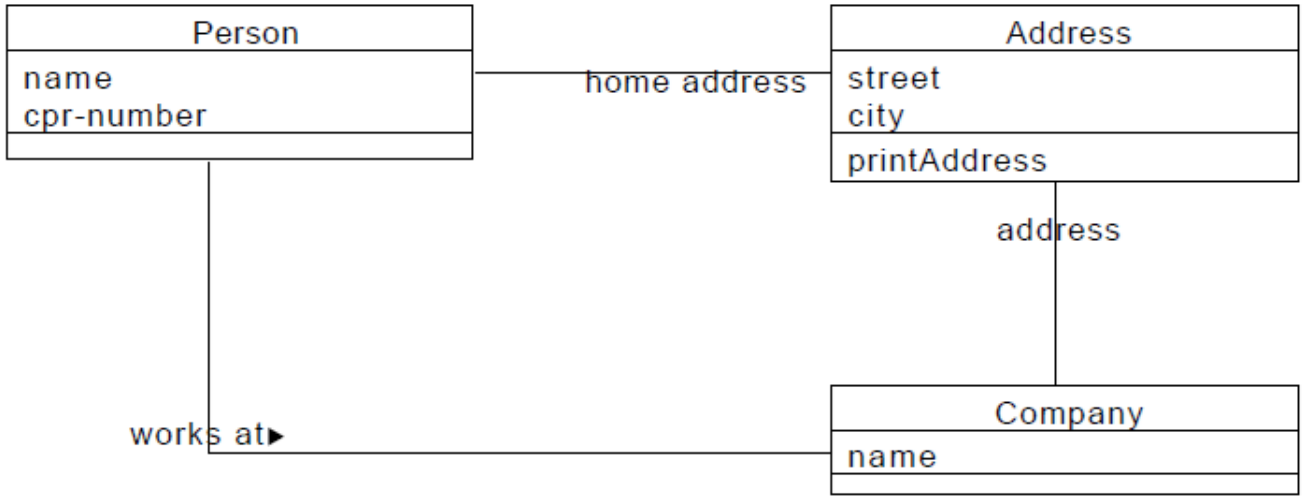
Principio: No te repitas

- La aplicación debe evitar especificar el comportamiento relacionado con un concepto particular en múltiples lugares ya que es una fuente frecuente de errores



Ejemplo de código duplicado





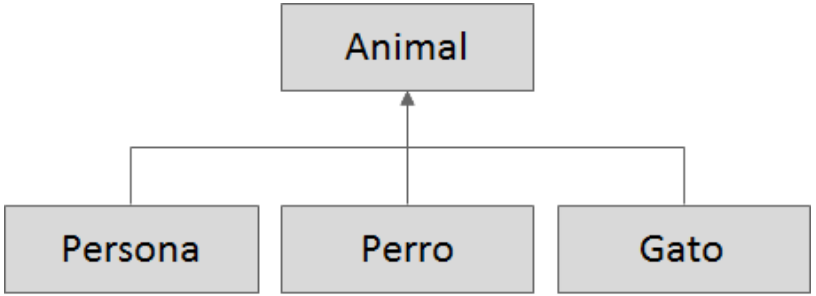
UNIVERSIDAD DE LIMA

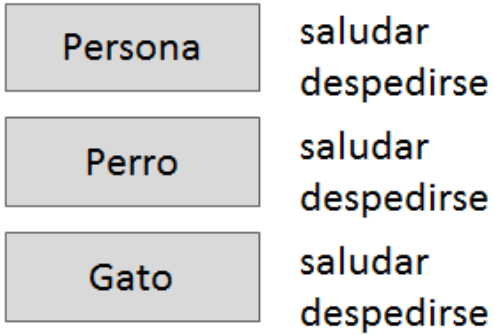
Principio DRY

- Técnicas para evitar duplicados
  - Realizar abstracciones apropiadas
  - Usar Herencia
  - Usar Clases con variables que sean instancias
  - Métodos con parámetros
- Refactorizar para remover duplicados
- Generar artefactos de una fuente común

Duck Typing

- Propiedad de algunos lenguajes de programación de tipeo dinámico.
- Nos permite definir objetos por lo que hacen (sus métodos) más que por lo que son (de quien heredan).
- Contraparte al polimorfismo.

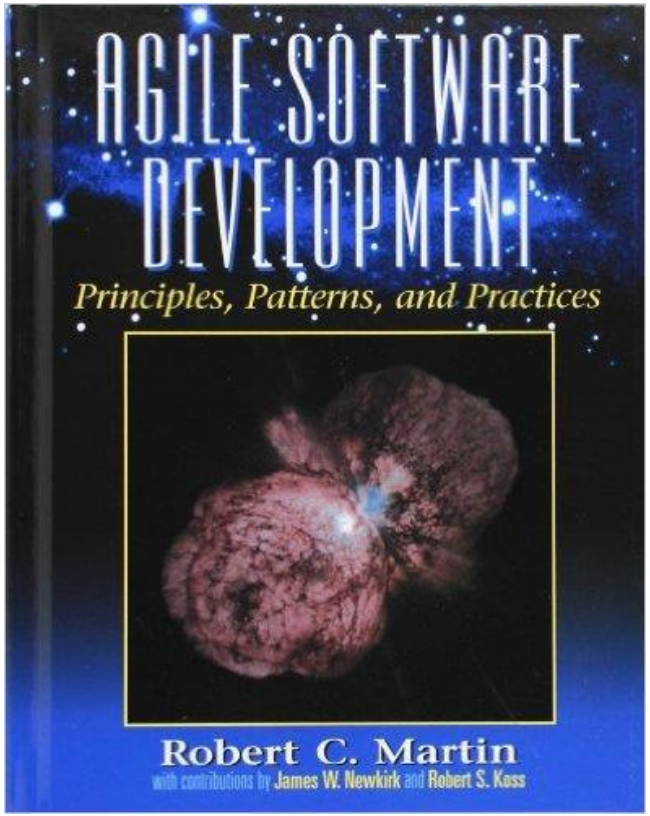




UNIVERSIDAD DE LIMA

Principios SOLID

- Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, 2002



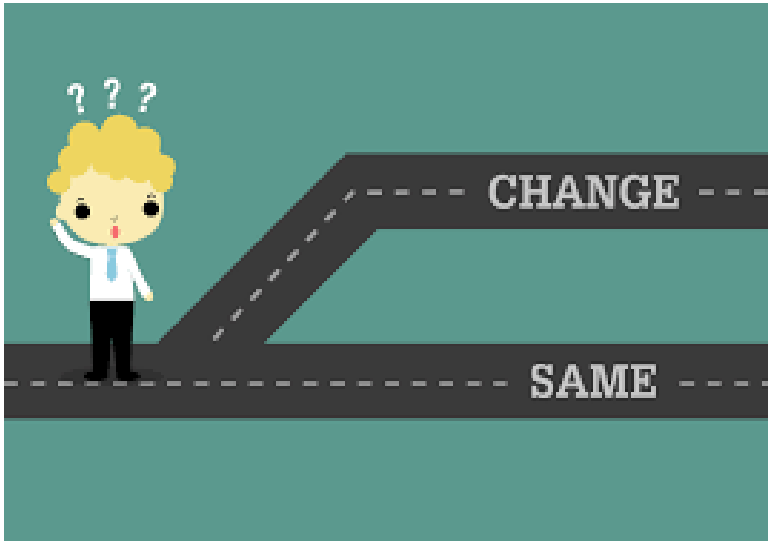
UNIVERSIDAD DE LIMA

Principios SOLID

- **S** (SRP) –Principio de responsabilidad única (Single responsibility principle)
- **O** (OCP) –Principio de abierto/cerrado (Open/closed principle)
- **L** (LSP) –Principio de sustitución de Liskov (Liskov substitution principle)
- **I** (ISP) –Principio de segregación de la interfaz (Interface segregation principle)
- **D** (DIP) –Principio de inversión de la dependencia (Dependency inversion principle)

Principio SRP: Responsabilidad única

- Cada componente de tu aplicación: Clase, función y variable debería tener una simple responsabilidad
- Nunca debería haber más de una razón en una clase para cambiar



¿Cuál es la propuesta correcta?

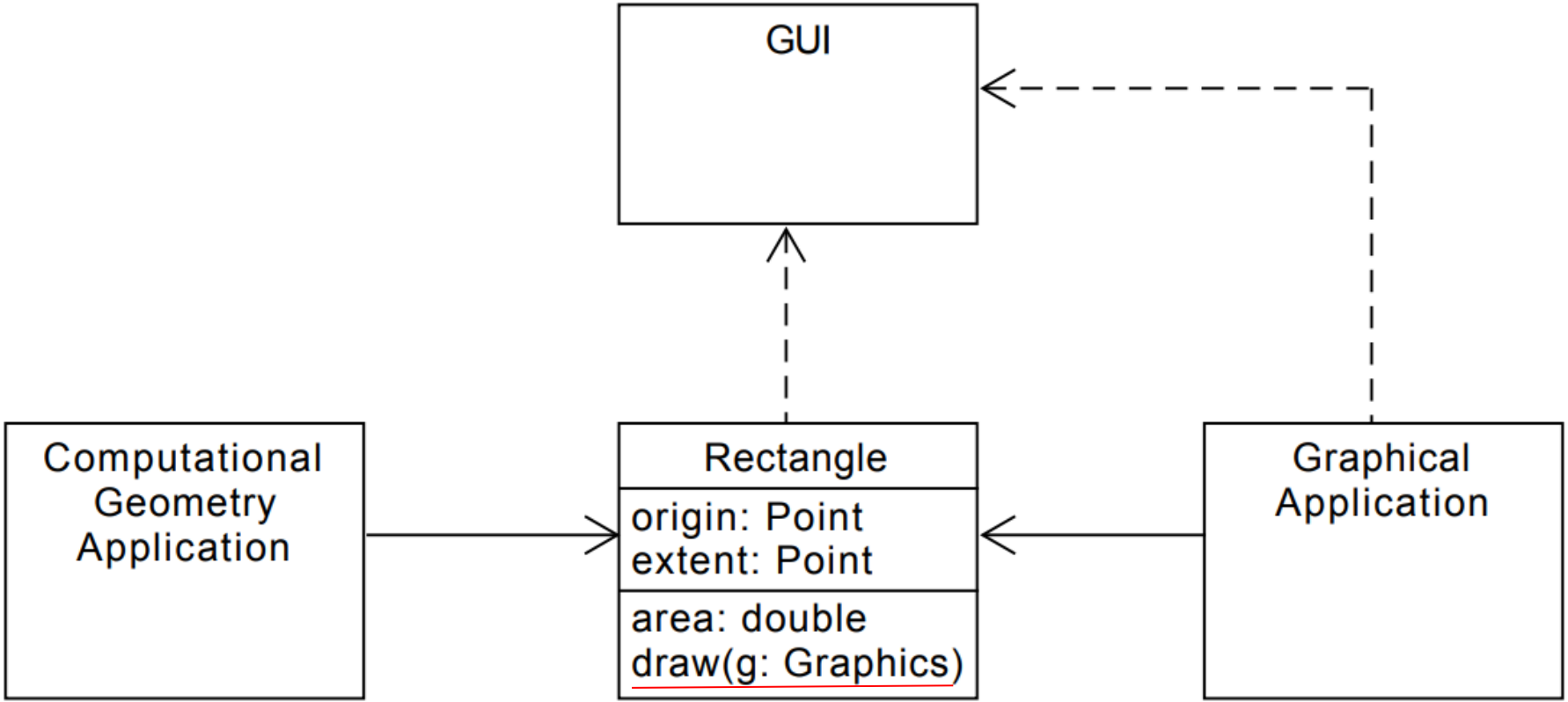
UNIVERSIDAD DE LIMA

Principio: Responsabilidad única

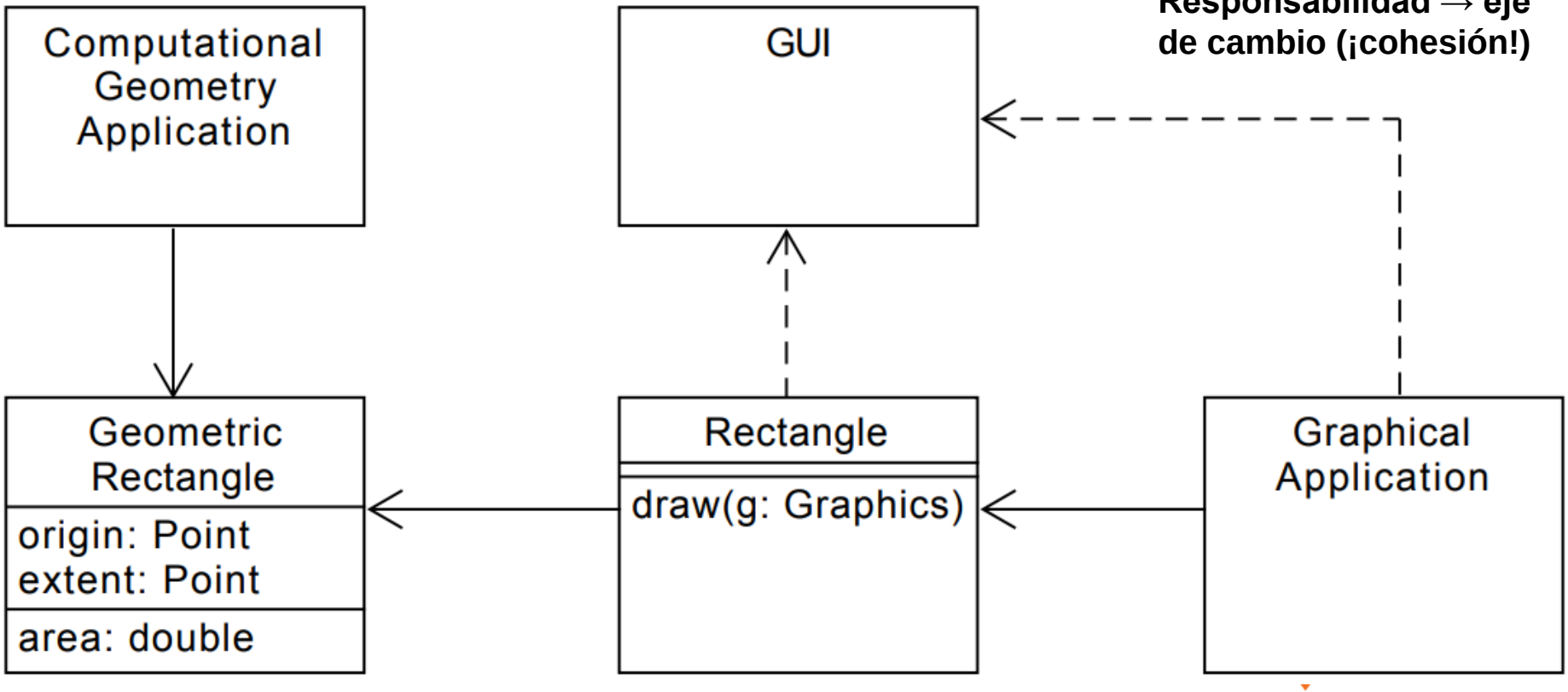
¿Cuál es la propuesta correcta?

UNIVERSIDAD DE LIMA

Ejemplo - Single responsibility principle

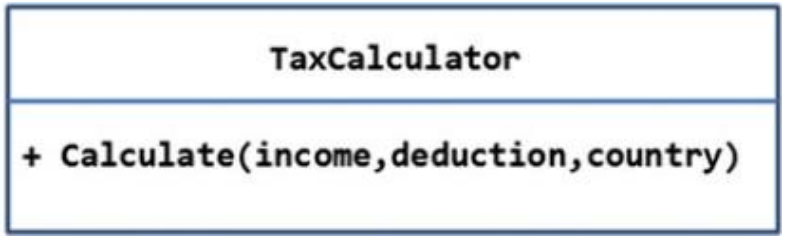


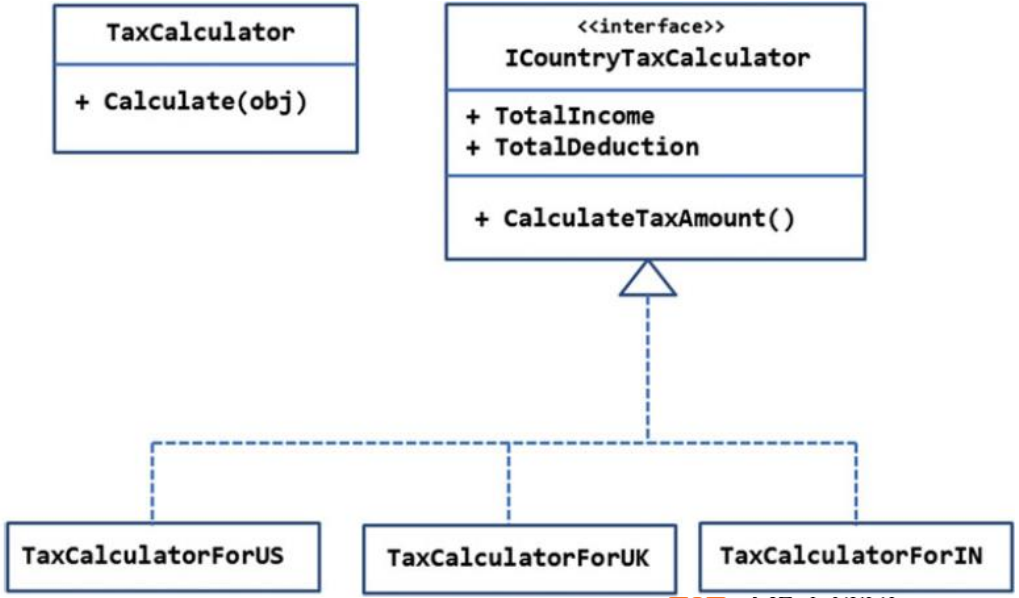
Ejemplo - Single responsibility principle



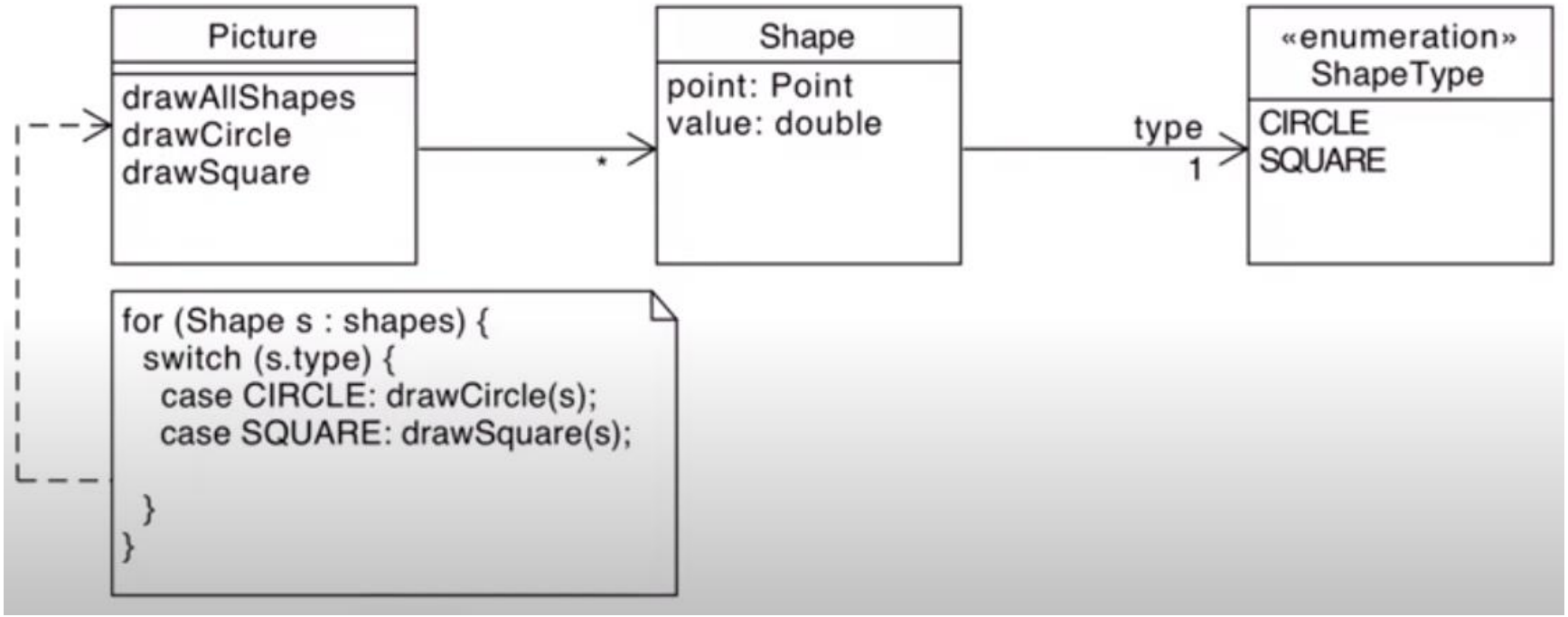
Principio OCP: Abierto-cerrado

- Entidades de software (clases, módulos, funciones, etc.) han de estar abiertas para extensiones, pero cerradas para modificaciones (Bertrand Meyer, 1988)

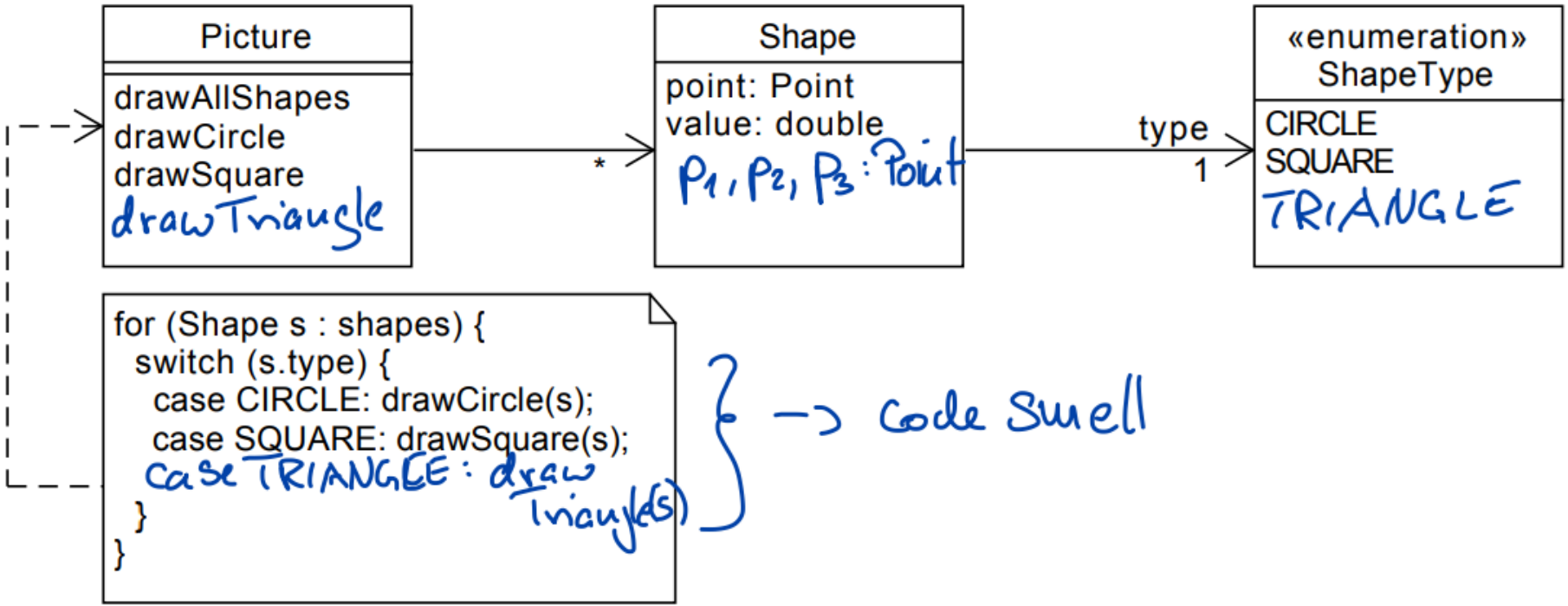




Ejemplo: Open/Closed Principle

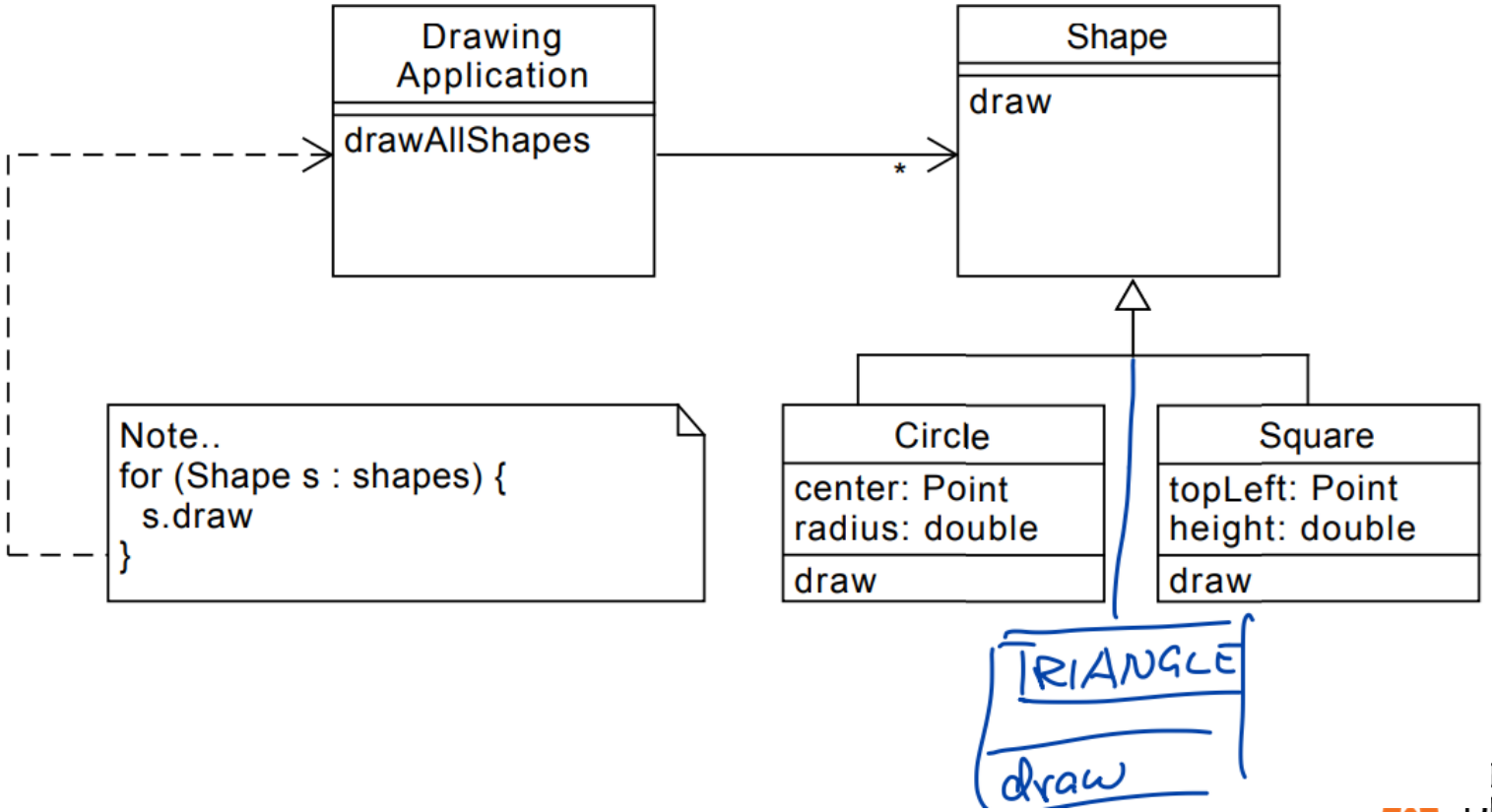


Ejemplo: Open/Closed Principle



UNIVERSIDAD DE LIMA

Ejemplo: Open/Closed Principle



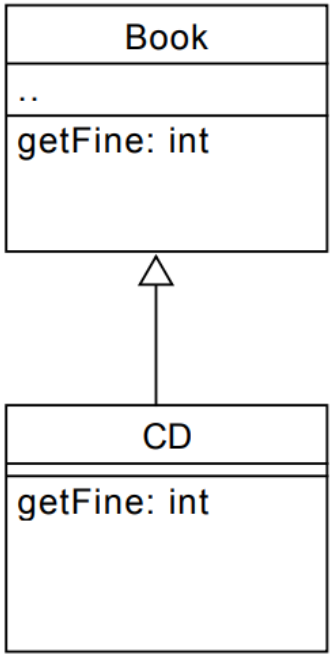
Principio LSP: Sustitución de Liskov

- Los subtipos deberían poder ser reemplazables por sus tipos base
  - Lo que queremos es algo parecido a la siguiente propiedad sustitutiva: Si por cada objeto **O1** de tipo **S** hay un objeto **O2** de tipo **T** tal que todos los programas **P** están definidos en términos de **T**, el comportamiento de **P** no cambia cuando **O1** es sustituido por **O2** siendo **S** un subtipo de **T**
- Dos clases compartiendo abstracción deben poder ser intercambiables en los clientes que las usan. Si al sustituir una por otra se produce una excepción o el comportamiento en el cliente cambia, no se está cumpliendo el principio.
- Ejemplo en Java

Ejemplo GetMulta

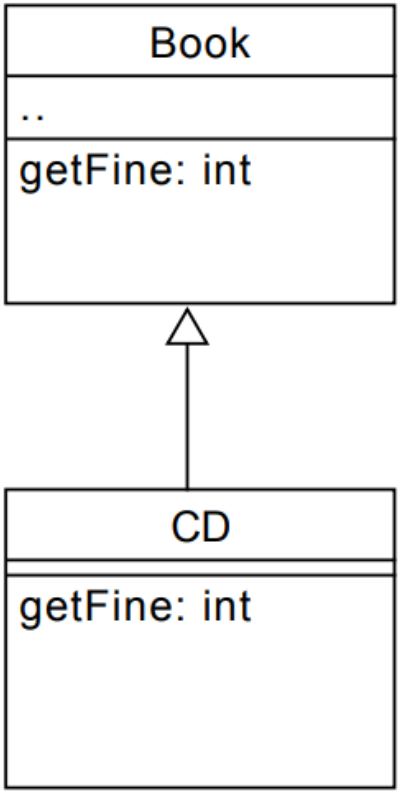
Conformidad con el comportamiento

- Sin embargo: "tiene los mismos métodos" no es suficiente



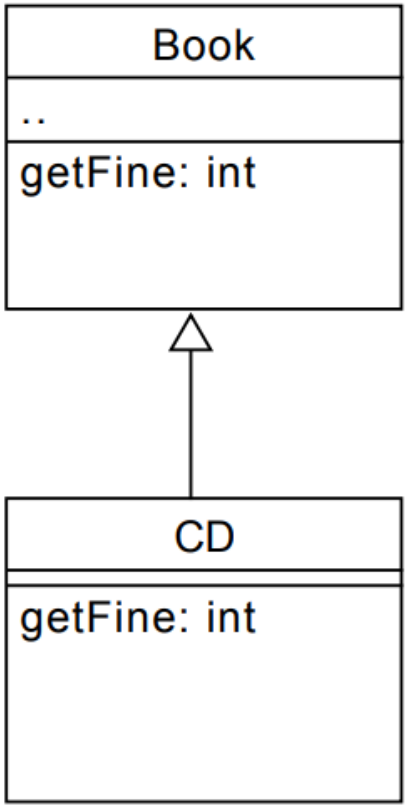
- Book getFine returns 100
- CD getFine returns 200

Ejemplo GetMulta



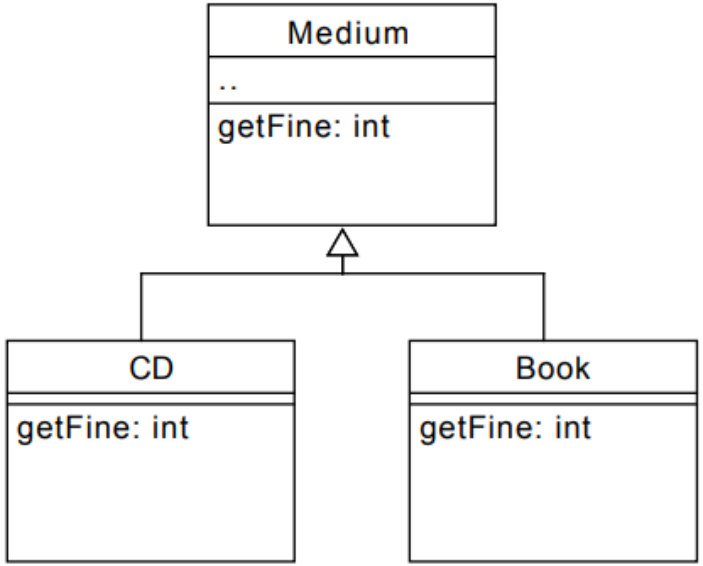
- Book getFine returns 100
- CD getFine returns 200
- Property $\Phi(b : Book)$ iff $b.getFine() = 100$ holds for books:
  - $\Phi(new\ Book()) = true$

Ejemplo GetMulta



- Book getFine returns 100
- CD getFine returns 200
- Property $\Phi(b : Book)$ iff $b.getFine() = 100$ holds for books:
  - $\Phi(new\ Book()) = true$
- LSP: should hold for subclass
- Subclass $\Phi(new\ CD()) = false$

Solución - Ejemplo GetMulta

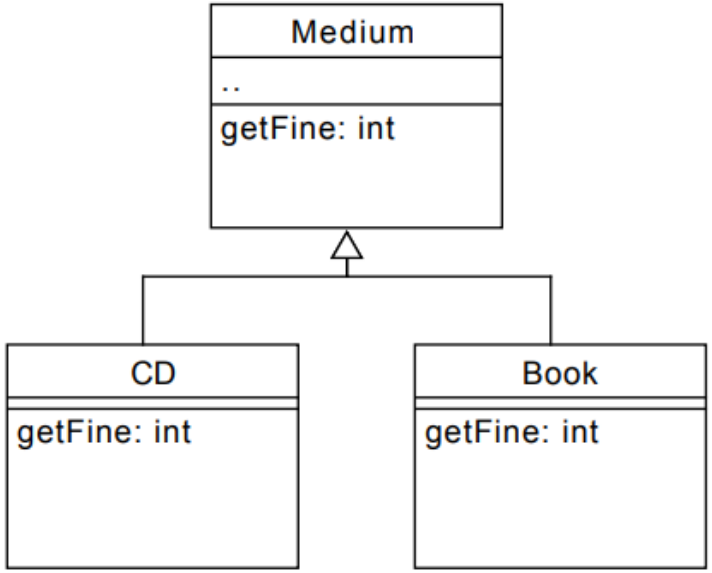


- Medium and getFine are abstract
- Expectation/contract for getFine()
  - $\Phi(m : Medium)$ iff $m.getFine() \geq 0$

  $\rightarrow$ Design by contract

UNIVERSIDAD DE LIMA

Solución - Ejemplo GetMulta



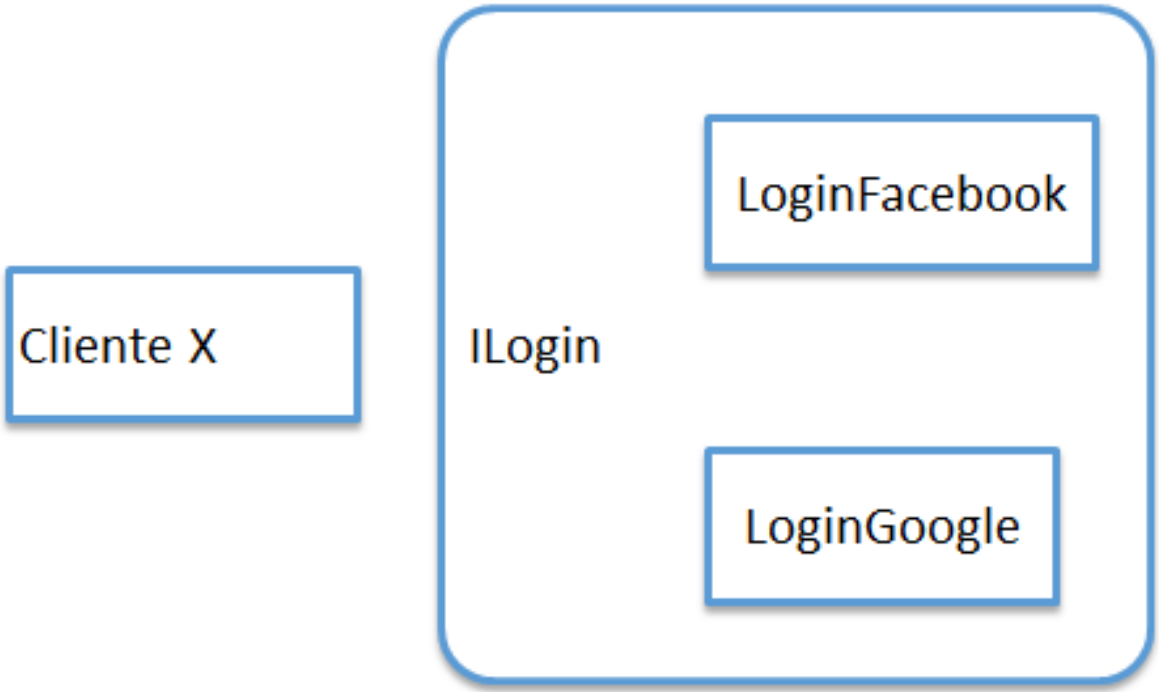
- Medium and getFine are abstract
- Expectation/contract for getFine()
  - $\Phi(m : Medium)$ iff $m.getFine() \geq 0$

  $\rightarrow$ Design by contract
- $\Phi(newBook()) = true$
- $\Phi(newCD()) = true.$

Conclusión: al crear una subclase, asegúrese de que satisfaga todas las expectativas / contratos de la superclase

UNIVERSIDAD DE LIMA

Principio: Sustitución de Liskov



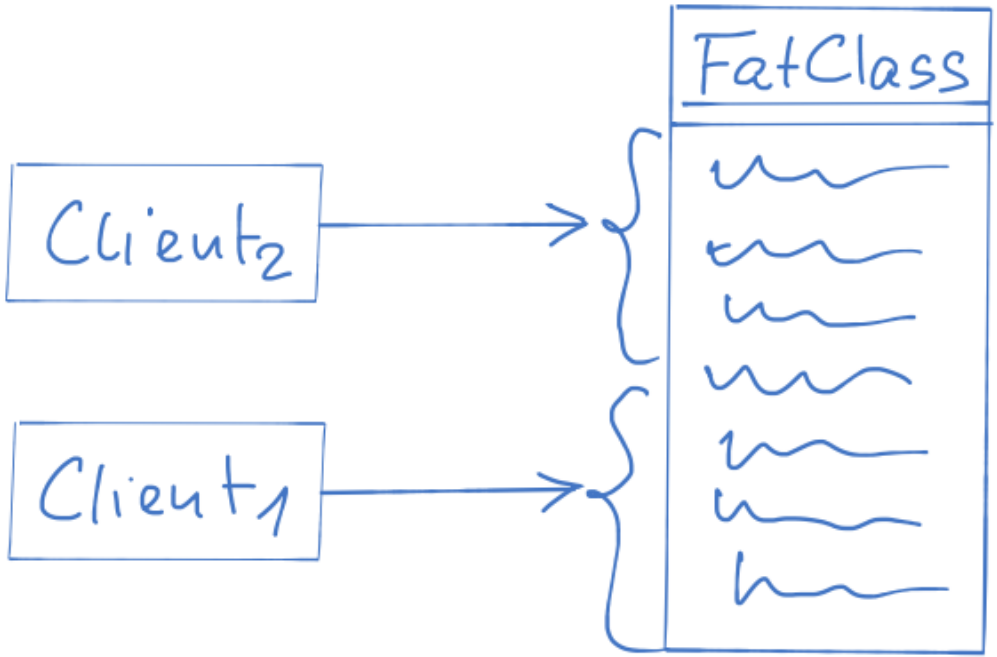
Principio ISP: Segregación de la interfaz

- Ninguna clase debería depender de métodos que no usa
- Las interfaces deben de tratar también ser enfocadas (no tener interfaces que tengan muchas operaciones.

UNIVERSIDAD DE LIMA

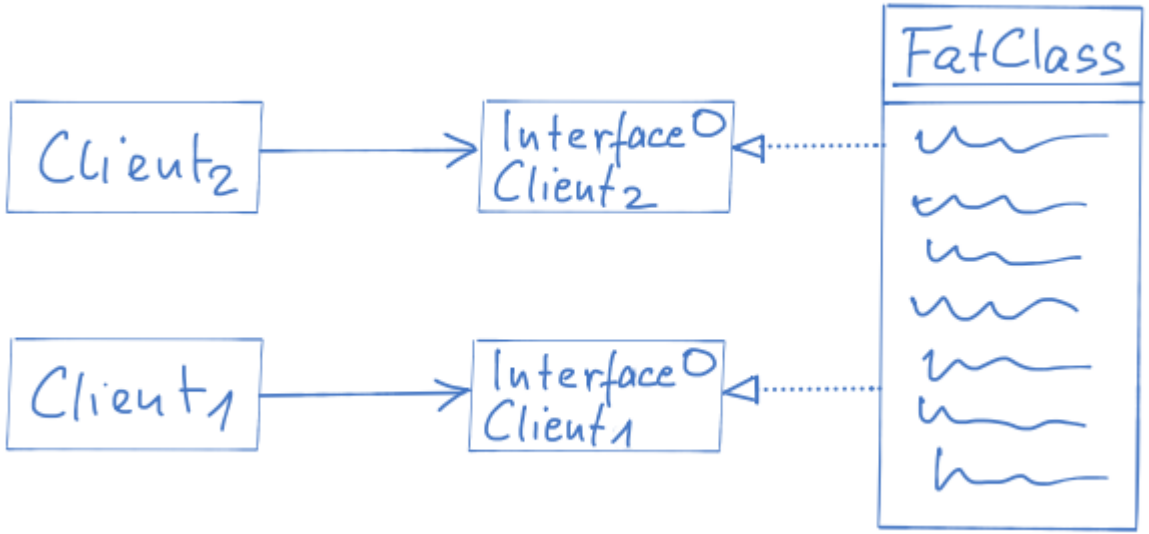
Principio: Segregación de la interfaz

Las clientes 1 y 2 dependen de la funcionalidad que no necesitan.



Principio: Segregación de la interfaz

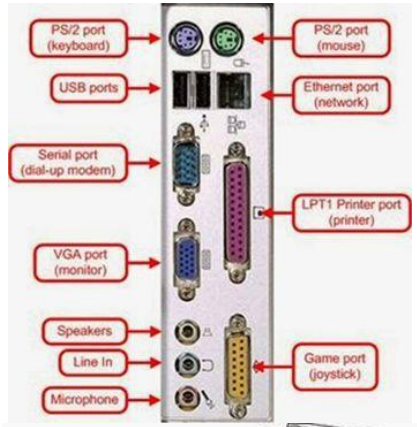
Separe la funcionalidad necesaria en interfaces

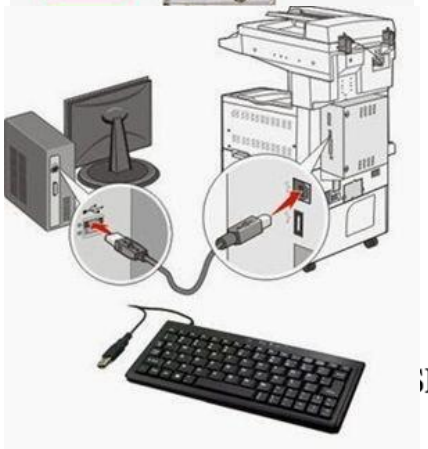


Principio DIP: Inversión de la dependencia

- Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.

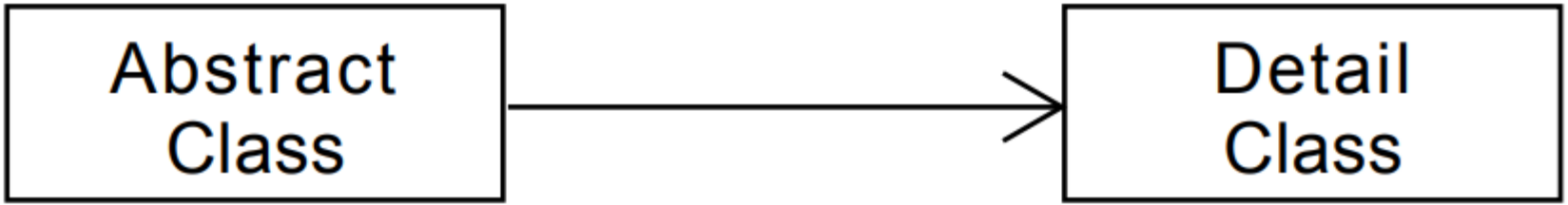
- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.



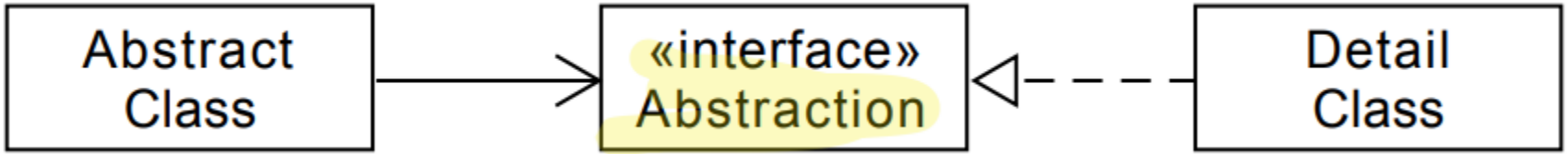


IDAD

Principio: Inversión de la dependencia



”Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deberían depender de abstracciones ".

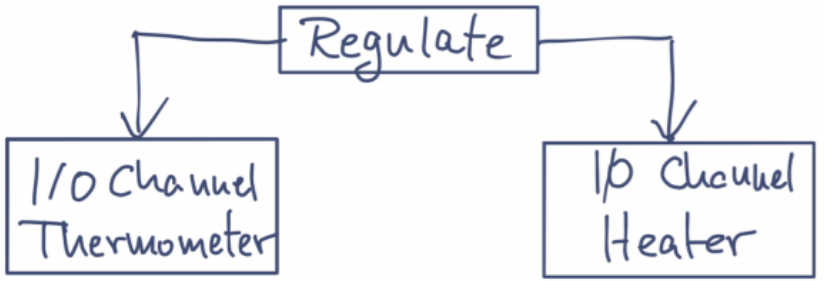


UNIVERSIDAD DE LIMA

Principio: Inversión de la dependencia
Ejemplo de horno

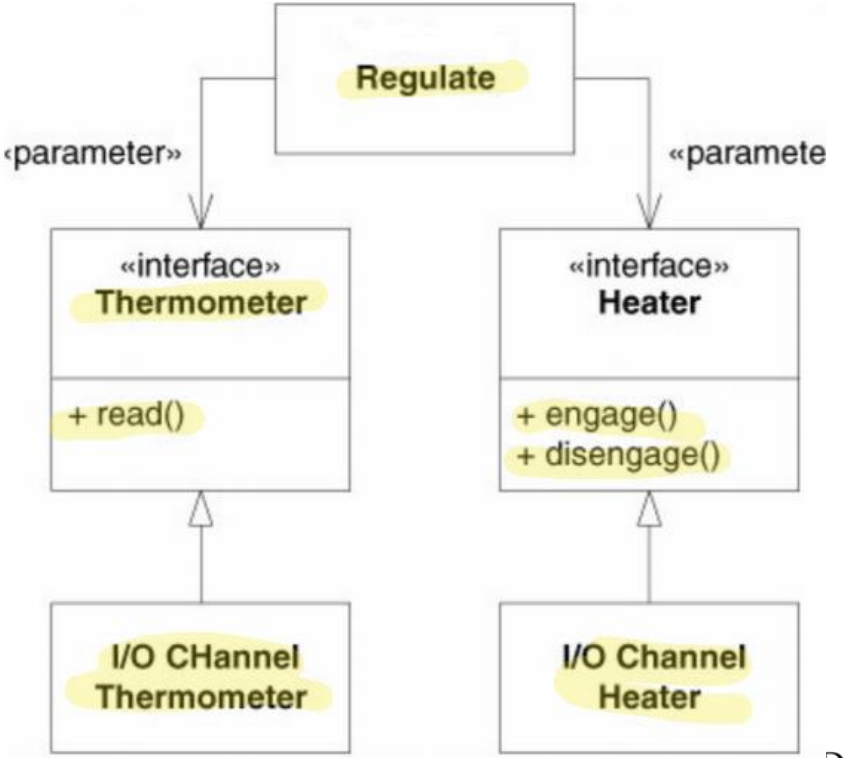
```
const byte TERMOMETER = 0x86; const byte FURNACE = 0x87;
const byte ENGAGE = 1;
const byte DISENGAGE = 0;

void Regulate(double minTemp, double maxTemp) {
for(;;) {
    while (in(THERMOMETER) > minTemp)
      wait(1);
    out(FURNACE,ENGAGE);
    while (in(THERMOMETER) < maxTemp)
      wait(1);
    out(FURNACE,DISENGAGE);
  }
}
```



Principio: Inversión de la dependencia
Ejemplo de horno - Solución

```
void Regulate(Thermometer t, Heater h,
      double minTemp,
      double maxTemp)
{
for(;;)
  {
    while (t.Read() > minTemp)
      wait(1);
    h.Engage();
    while (t.Read() < maxTemp)
      wait(1);
    h.Disengage();
  }
}
```



DAD

Robert C. Martin (2007) Agile Principles, Patterns, and Practices in C#

Referencias Bibliográficas

1. Fowler, M. (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.
2. Martin, R. C., Newkirk, J., & Koss, R. S. (2003). Agile software development: principles, patterns, and practices (Vol. 2). Upper Saddle River, NJ: Prentice Hall.