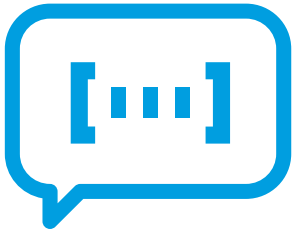
netmind
a bts company

Java 11

Principio de diseño SOLID y patrones de diseño

netmind
a bts company

01



PRINCIPIOS SOLID

SOLID

- Solid es un acrónimo inventado por Robert C.Martin (Uncle Bob) para establecer los cinco principios básicos de la programación orientada a objetos y diseño.
- Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.
- Los 5 principios son:
  - **SRP**: Single Responsibility
  - **OCP**: Open/Close
  - **LSP**: Liskov's Substitution
  - **ISP**: Interface Segregation
  - **DIP**: Dependency Inversion

Consideraciones de diseño

- El valor primario del software es su facilidad para cambiar, debido sobre todo a la incorporación de nuevas funcionalidades
- Un software que no cambia o evoluciona está destinado a morir.
- Por tanto el valor secundario es la funcionalidad

SRP: Single Responsibility Principle

Una clase debe tener sólo (una responsabilidad) una razón para cambiar

- Principio de Responsabilidad Única
- Robert C. Martin →Libro Agile Software Development, Principles, Patterns, and Practices
- **Porqué:**
  - Si hay 2 razones para cambiar → 2 equipos pueden trabajar en el mismo código por 2 razones diferentes
  - Extensión de requerimientos

Ejemplo: Objetos que se autoimprimen

- Este código **mezcla** lógica de negocio y presentación (imprimir) → más de una responsabilidad
- Porqué es un problema? Qué pasa si necesitamos imprimir en diferentes formatos?

```
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function printCurrentPage() {
        echo "current page content";
    }
}
```

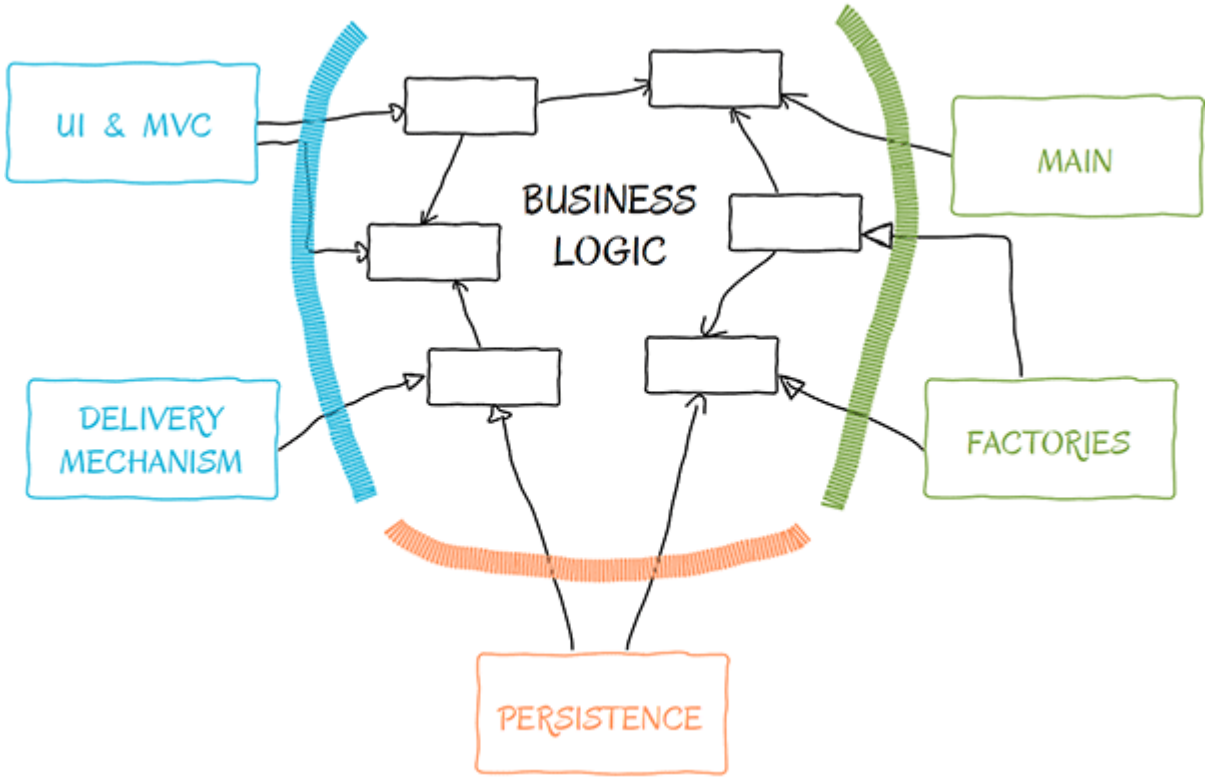
Ejemplo: Objetos que se autoimprimen – Posible solución

- Separando la lógica de negocio y generando una clase específica para la parte de presentación
- De esta manera, cada clase tiene su responsabilidad

```php
class Book {

    function getTitle() {
        return "A Great Book";
    }

    function getAuthor() {
        return "John Doe";
    }

    function turnPage() {
        // pointer to next page
    }

    function getCurrentPage() {
        return "current page content";
    }

}
```

```php
interface Printer {

    function printPage($page);
}

class PlainTextPrinter implements Printer {

    function printPage($page) {
        echo $page;
    }

}
```

En un nivel superior



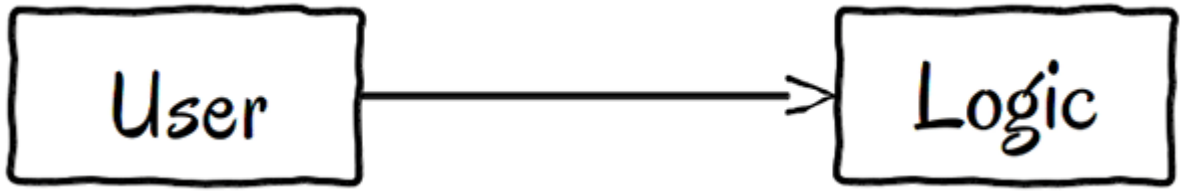
OCP: Open/Close Principle

Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión, pero cerradas para su modificación.

- Principio Abierto-Cerrado
- Bertrand Mayer → Libro Object-Oriented Software Construction
- Complementario con SRP
- **Porqué:**
  - Cuando una nueva funcionalidad es necesaria, no deberíamos modificar nuestro software, si no escribir nuevo código que añada funcionalidad al existente
  - Nuevas características sin "redeploy" de los binarios existentes

OCP: Open/Close Principle

- Una relación directa entre dos clases (fig) viola el OCP



- User usa Logic directamente
- **Problema**: cuando necesitemos implementar un segundo tipo de Logic y queramos que User pueda usar ambas, sería necesario cambiar el Logic existente
- No tenemos manera de proveer nueva Logic sin afectar la actual
- Además en lenguajes compilados, seguramente User tendría que cambiar

Ejemplo: Tracking de descarga

```php
function testItCanGetTheProgressOfAFileAsAPercent() {
    $file = new File();
    $file->length = 200;
    $file->sent = 100;

    $progress = new Progress($file);

    $this->assertEquals(50, $progress->getAsPercent());
}
```

```php
class File {
    public $length;
    public $sent;
}
```

```php
class Progress {

    private $file;

    function __construct(File $file) {
        $this->file = $file;
    }

    function getAsPercent() {
        return $this->file->sent * 100 / $this->file->length;
    }

}
```

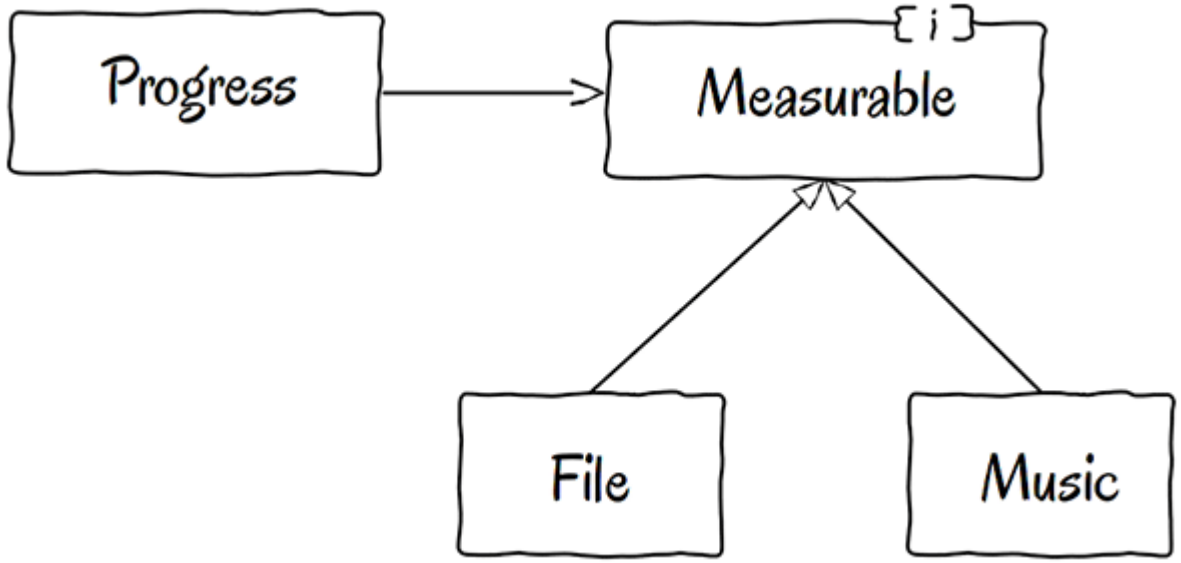
Ejemplo: Tracking de descarga

- Todas las aplicaciones que se espera que evolucione en el tiempo tendrá nuevas características.

Cambiando requerimientos:

- Una nueva característica que se podría esperar es que haga stream de música en lugar de descargas.
- Queremos ofrecer una barra de progreso de % de descarga, podemos?
- No! Nuestro progreso está unido a File → necesitamos tocar Progress (Music y File)
- Si se hubiera respetado el OCP podríamos aplicar Progress a Music

Solución: usar Strategy Design Pattern



Solución: Tracking de descarga

```
function testItCanGetTheProgressOfAFileAsAPercent() {
    $file = new File();
    $file->setLength(200);
    $file->setSent(100);

    $progress = new Progress($file);

    $this->assertEquals(50, $progress->getAsPercent());
}
```

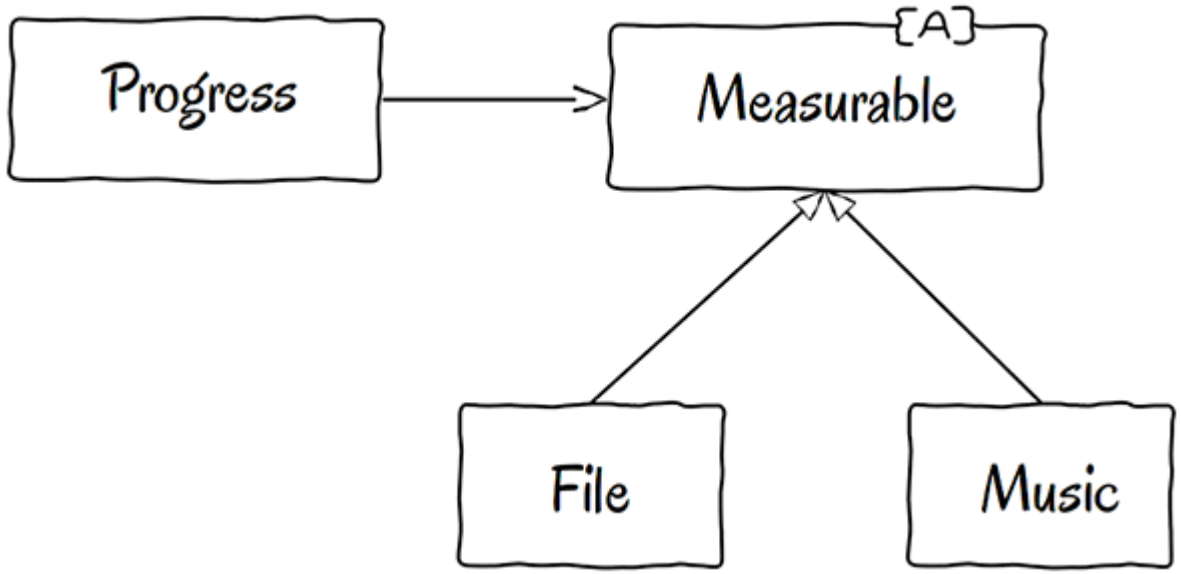
```
interface Measurable {
    function getLength();
    function getSent();
}
```

```
class Progress {

    private $measurableContent;

    function __construct(Measurable $measurableContent) {
        $this->measurableContent = $measurableContent;
    }

    function getAsPercent() {
        return $this->measurableContent->getSent() * 100 / $this->measurableContent->getLength();
    }

}
```

Solución: Tracking de descarga

```php
class File implements Measurable {

    private $length;
    private $sent;

    public $filename;
    public $owner;

    function setLength($length) {
        $this->length = $length;
    }

    function getLength() {
        return $this->length;
    }

    function setSent($sent) {
        $this->sent = $sent;
    }

    function getSent() {
        return $this->sent;
    }

    function getRelativePath() {
        return dirname($this->filename);
    }

    function getFullPath() {
        return realpath($this->getRelativePath());
    }

}
```

Solución II: usar Template Method Design Pattern

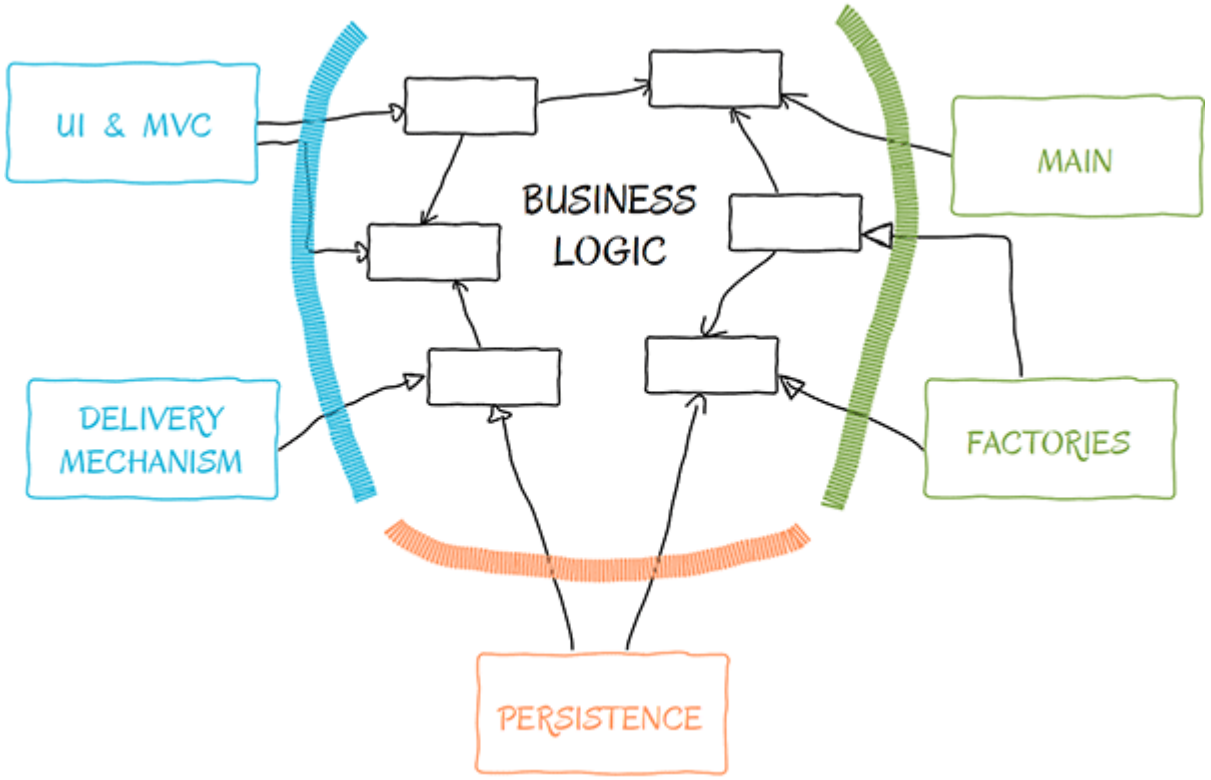


OCP: Los límites

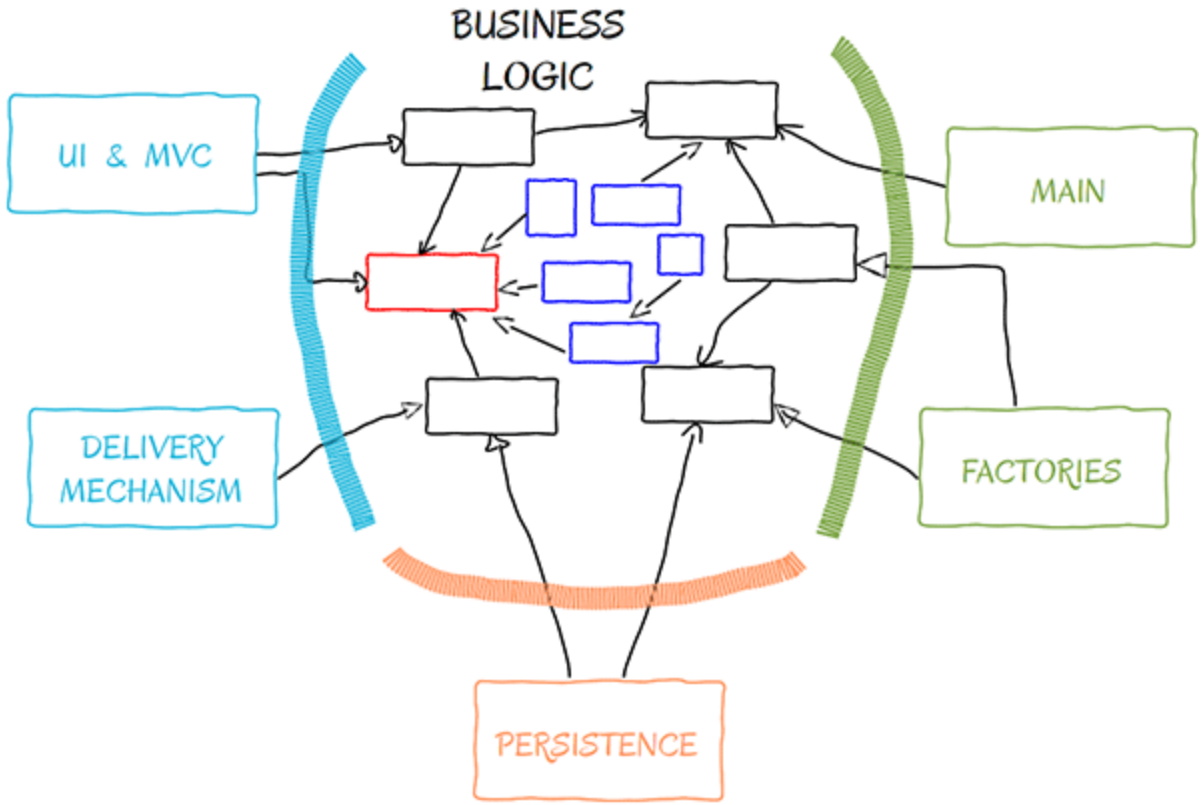
- Cualquier clase usando directamente otra viola el OCP → es necesario desacoplarlas a travñes de interfaces

- **El límite:**
  - Cuando es más dificil respetar el OCP que modificar el código existente, o
  - El coste de arquitectura no justifica el cambio del código existente

En un nivel superior



En un nivel superior



LSP: Liskov Substitution Principle

Sea q (x) una propiedad probable sobre objetos x de tipo T. Entonces q(y) debería ser probable para objetos y de tipo S donde S es un subtipo de T.

(Barbara Liskov)

Los subtipos deben ser sustituibles
por sus tipos de base

- Robert C. Martin
- **Porqué:**
  - Una subclase debe sobreescribir los métodos de la clase padre de manera que no rompa la funcionalidad desde el punto de vista del cliente.

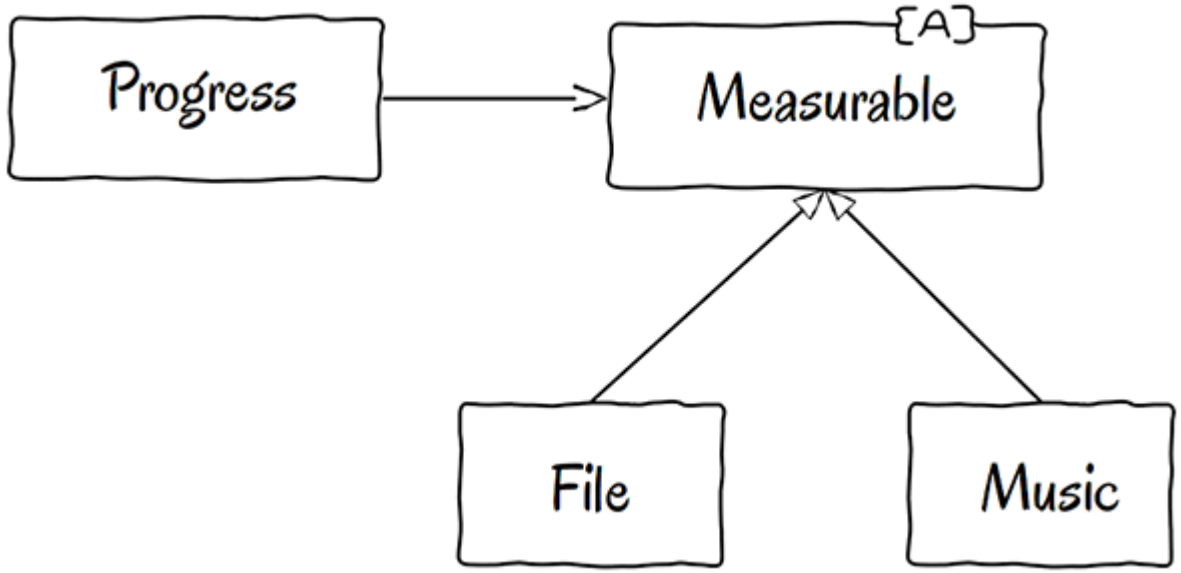
Ejemplo: Vehículo

```php
class Vehicle {

    function startEngine() {
        // Default engine start functionality
    }

    function accelerate() {
        // Default acceleration functionality
    }
}
```

```php
class Driver {
    function go(Vehicle $v) {
        $v->startEngine();
        $v->accelerate();
    }
}
```

```php
class Car extends Vehicle {

    function startEngine() {
        $this->engageIgnition();
        parent::startEngine();
    }

    private function engageIgnition() {
        // Ignition procedure
    }

}

class ElectricBus extends Vehicle {

    function accelerate() {
        $this->increaseVoltage();
        $this->connectIndividualEngines();
    }

    private function increaseVoltage() {
        // Electric logic
    }

    private function connectIndividualEngines() {
        // Connection logic
    }

}
```

Ejemplo: Abstracción de Vehículo



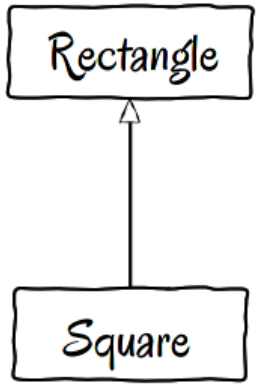
- LSP está fuertemente relacionado con OCP:
- Una violación del LSP es una violación latent del OCP.

Problema: Rectángulos y cuadrados

```php
class Rectangle {

    private $topLeft;
    private $width;
    private $height;

    public function setHeight($height) {
        $this->height = $height;
    }

    public function getHeight() {
        return $this->height;
    }

    public function setWidth($width) {
        $this->width = $width;
    }

    public function getWidth() {
        return $this->width;
    }
```

```php
function area() {
    return $this->width * $this->height;
}
```

```php
class Square extends Rectangle {

    public function setHeight($value) {
        $this->width = $value;
        $this->height = $value;
    }

    public function setWidth($value) {
        $this->width = $value;
        $this->height = $value;
    }
}
```




```php
class Client {

    function areaVerifier(Rectangle $r) {
        $r->setWidth(5);
        $r->setHeight(4);

        if($r->area() != 20) {
            throw new Exception('Bad area!');
        }

        return true;
    }

}
```

Problema: Rectángulos y cuadrados

- Probamos:

```php
class LspTest extends PHPUnit_Framework_TestCase {

    function testRectangleArea() {
        $r = new Rectangle();
        $c = new Client();
        $this->assertTrue($c->areaVerifier($r));
    }

}
```

- Si enviamos un cuadrado al cliente, no debería romperse, no?

```php
function testSquareArea() {
    $r = new Square();
    $c = new Client();
    $this->assertTrue($c->areaVerifier($r));
}
```

ISP: Interface Segregation Principle

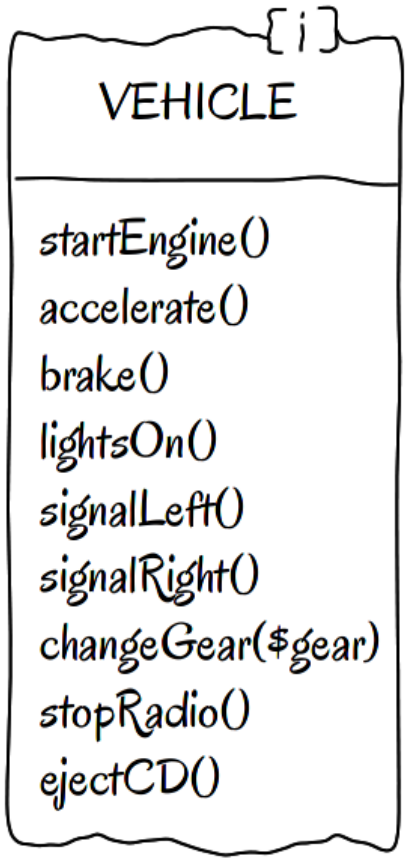
Ninguna clase cliente debe ser
obligado a depender de métodos
que no utiliza.

- Su objetivo es comunicar al código de la clase cliente cómo usar el modulo
- Complemento de SRP
- **Porqué:**
  - En todas las aplicaciones modulares debe haber algún tipo de interfaz de la que el cliente pueda depender.
  - Interfaces o patrones de diseño - Façades
  - Nos fuerza a pensar desde el punto de vista del cliente de nuestro código

ISP: Interface Segregation Principle

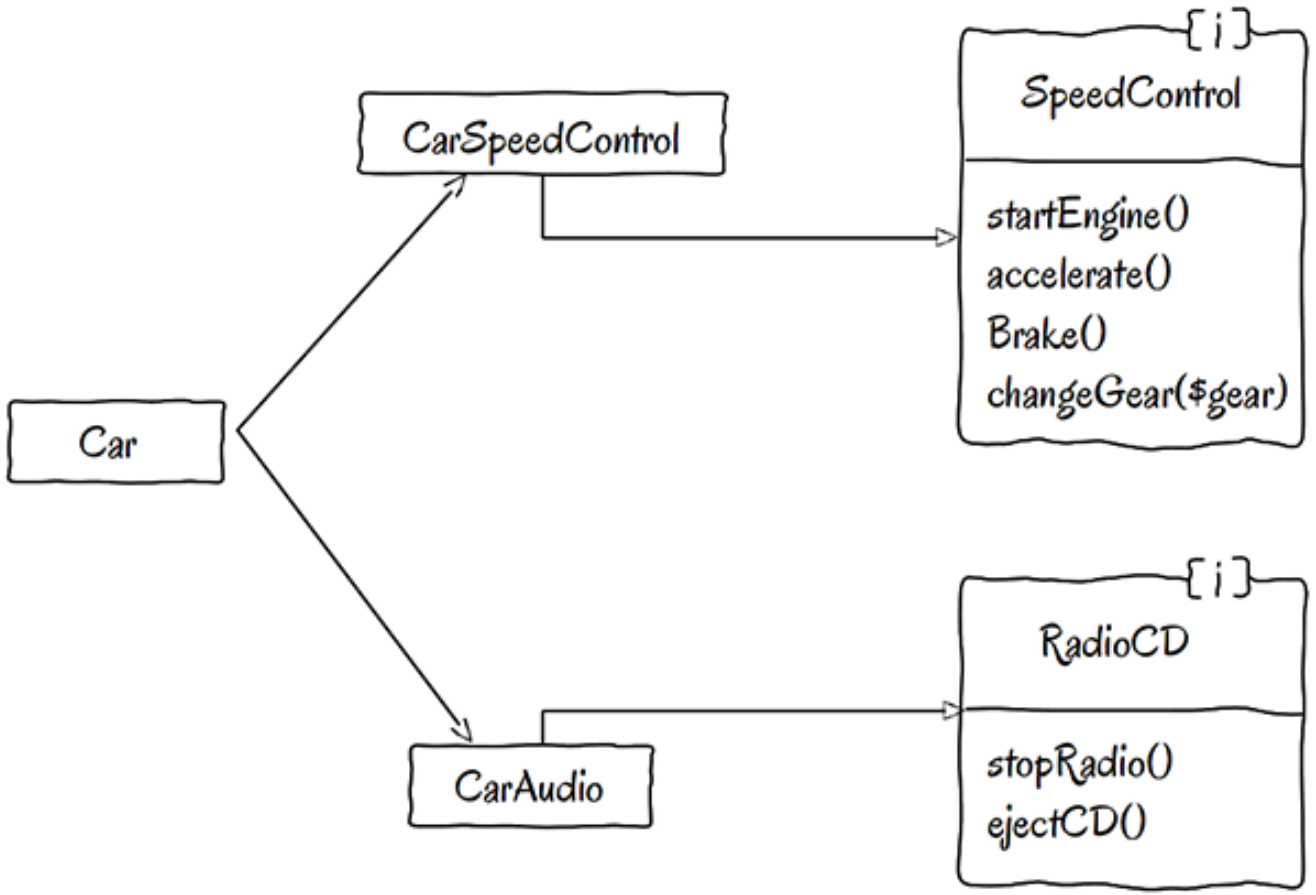
- Cómo deben ser estas interfaces?

- Exponer todas las funcionalidades de nuestro modulo
- **Consecuencias:**
  - Una clase gigante (Car o Bus) que implementa todos los métodosde la interface · Evitar a toda costa.
  - O, Muchas pequeñas clases (LightsControl, SpeedControl, or RadioCD) que implementan la interface, pero proveyendo utilidad solo para las partes que ellas implementan.
- **Ninguna es buena!**



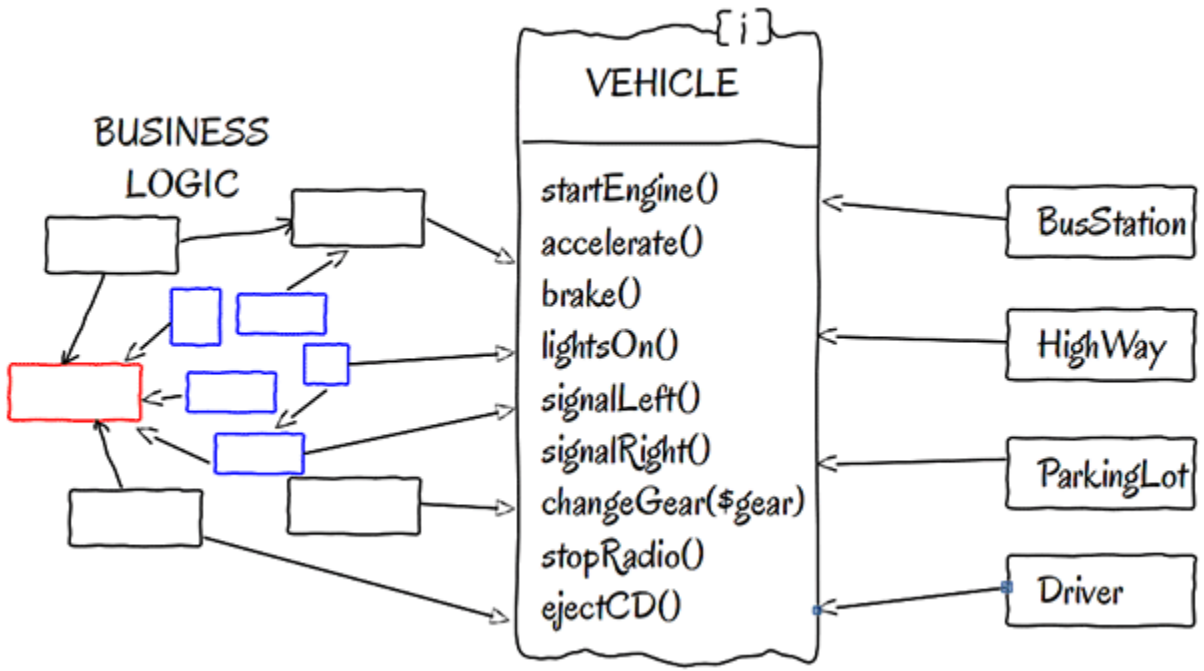
ISP: Interface Segregation Principle

- Romper la interface en piezas especializadas:



ISP: Interface Segregation Principle

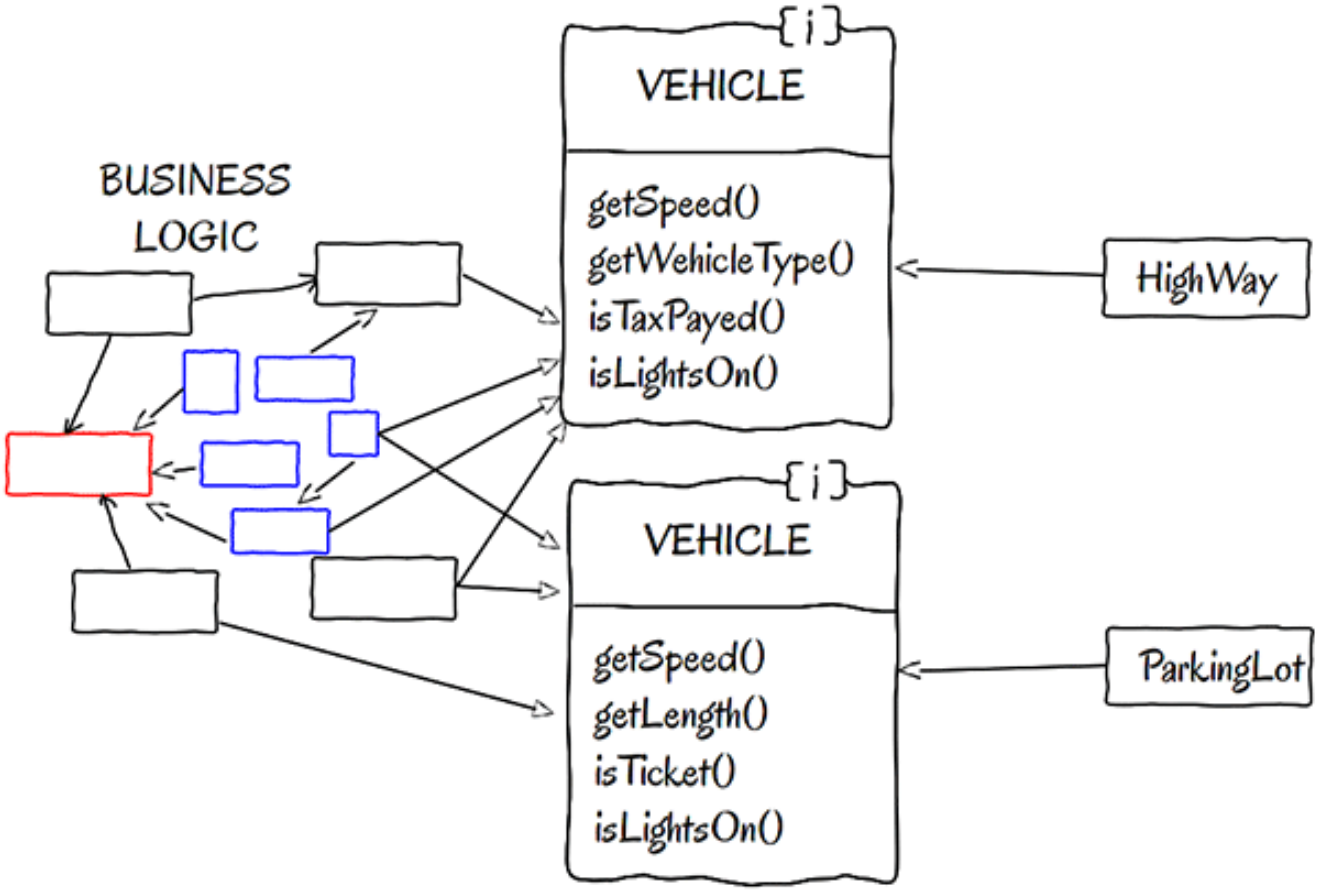
- Queremos proveer a nuestro cliente métodos para usar nuestro modulo (de tipo vehículo)



- Pero…no todos los clientes dependen de todos los métodos
- En consecuencia cambios en métodos que ni siquiera usa el cliente, pueden suponer cambios en el propio cliente

ISP: Interface Segregation Principle

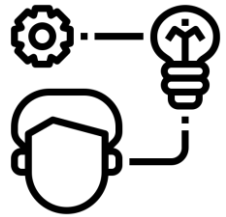
- La solución es segregar la Interface



ISP: Interface Segregation Principle

- La interfaces pertencen a los clientes, no a la lógica de negocio.
- Debemos diseñarlas de manera que se ajusten a los clientes.
- Pueden usar/necesitar varias interfaces
- En la lógica de negocio, una clase puede puede implementar varias interfaces, si es necesario.

Pongámoslo en práctica



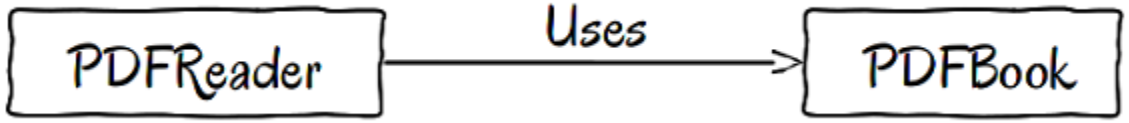
- Tenemos un cliente que, pasándole un array de formas, calcula su área: AreaCalculator.
- Queremos añadir formas tridimensionales (esferas, cuboides) y poder calcular su volume (además de su área).
- Implementa una arquitectura que permita cumplir con los principios: SRP, OCP, LSP y ISP

DIP: Dependency Inversion Principle

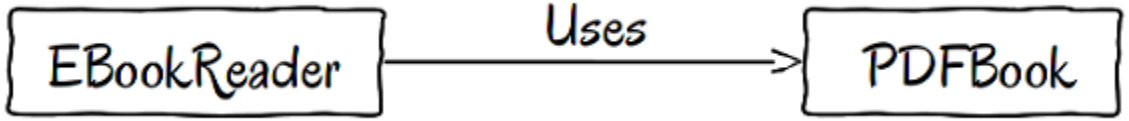
A.Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
B. Abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.

- Robert C. Martin
- **Porqué:**
  - Nos guía/ayuda a respetar todos los otros principios
  - (Casi) obliga a respetar OCP
  - Permiten separar las responsabilidades.
  - Hace utilizar correctamente los subtipos.
  - Ofrecerle la oportunidad de segregar interfaces

DIP: El lector de libros

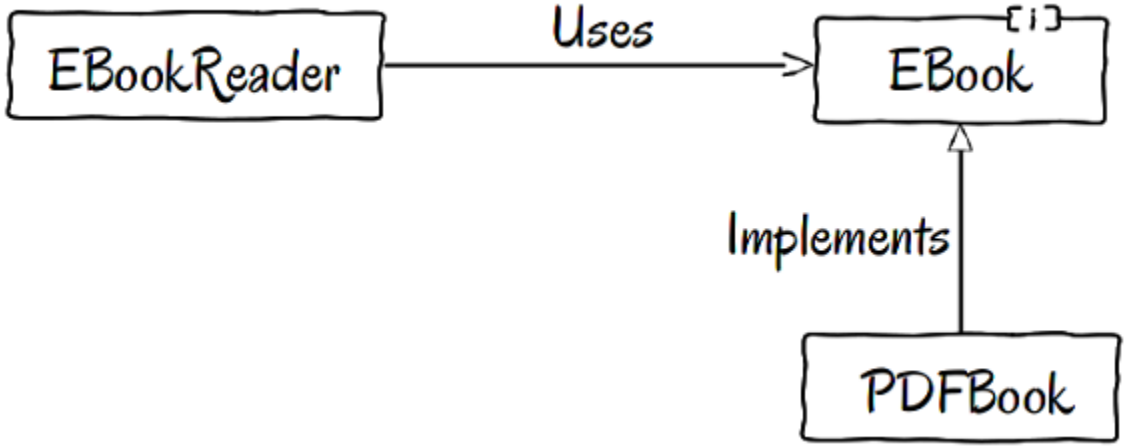


- Si es solo un lector de libros en PDF, es OK
- Pero, y si queremos implementar un lector más genérico (EbookReader)?

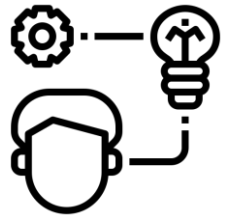


DIP: El lector de libros

- Tenemos un element genérico “EBookReader” que usa un tipo específico de libro “PDFBook” → Una abstracción depende de un detalle
- **Solución**: depender de una abstracción



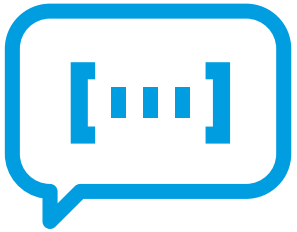
Pongámoslo en práctica



- Actualiza el código para implementar la dependencia de libros
- Y si queremos leer tipos de tipo MobiBook?

netmind
a bts company

02



PATRONES DE DISEÑO

Patrones de diseño

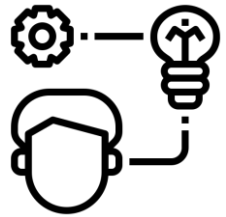
- Los **patrones del diseño** tratan los problemas del diseño que se repiten y que se presentan en situaciones particulares del diseño, con el fin de proponer soluciones a ellas.
- Por lo tanto, los patrones de diseño son soluciones exitosas a problemas comunes.
- Existen muchas formas de implementar patrones de diseño. Los detalles de las implementaciones son llamadas estrategias.
- Fueron propuestos en el libro "Design Patterns: Elements of Reusable Object Oriented Sofware" escrito por los ahora famosos **Gang of Four** (GoF, que en español es la pandilla de los cuatro) formada por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.

Tipos

- **Creacionales**: tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
- **Estructurales**: describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.
- **Comportamiento**: ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

- Se puede encontrar un catálogo detallado con ejemplos en :
  - http://www.journaldev.com/1827/java-design-patterns-example-tutorial

Pongámoslo en práctica – Patrones Básicos



Examina los siguientes patrones:

Singleton
http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples

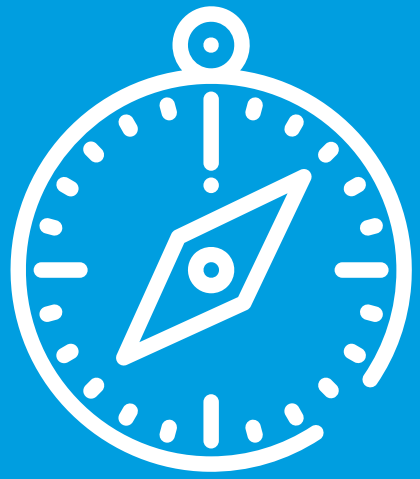
Factory
http://www.journaldev.com/1392/factory-design-pattern-in-java

Facade
http://www.journaldev.com/1557/facade-design-pattern-in-java

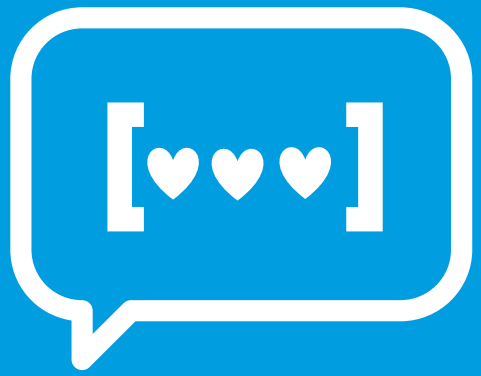
Proxy
http://www.journaldev.com/1572/proxy-design-pattern

Decorator
http://www.journaldev.com/1540/decorator-design-pattern-in-java-example

Observer
http://www.journaldev.com/1739/observer-design-pattern-in-java



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: