Principios generales de diseño

Diseño de Sistemas Software
Curso 2017/2018

Carlos Pérez Sancho

Universitat d'Alacant
Universidad de Alicante

Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Contenidos

1. Composición sobre herencia

2. Principios SOLID

3. Inyección de dependencias

Composición sobre herencia

Composición sobre herencia

La herencia supone un acoplamiento muy fuerte que puede ocasionar problemas:

- Las relaciones **“es un …”** pueden dejar de cumplirse al incorporar nuevos cambios
- Heredar de una clase para adquirir un comportamiento nos impide heredar de otra
- Los cambios de comportamiento en tiempo de ejecución son difíciles
- **La herencia es difícil de implementar cuando los modelos persisten en una base de datos relacional**

Composición sobre herencia

Síntomas de que una herencia puede no ser adecuada:

- Numerosos métodos heredados que no se usan
- Demasiados métodos sobrescritos
- Demasiados niveles de herencia

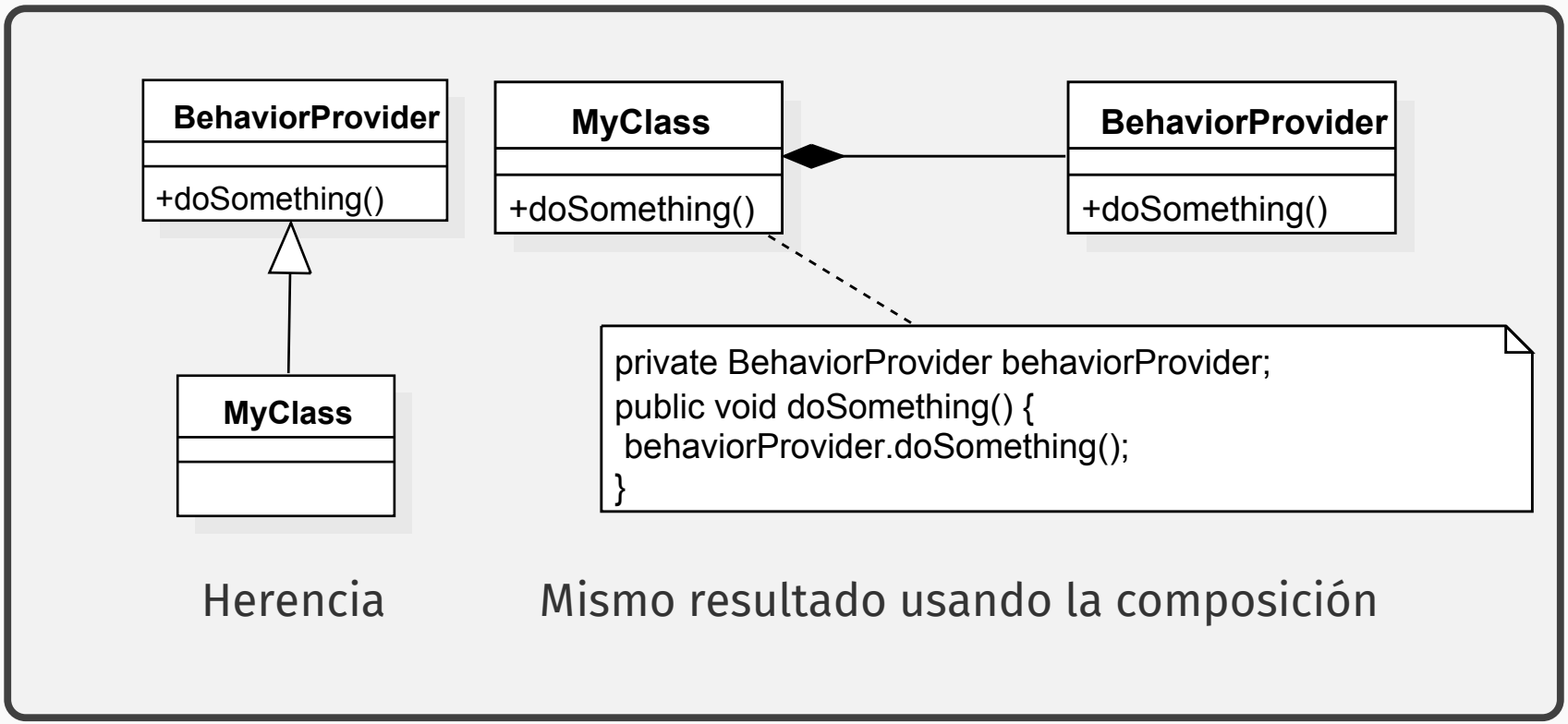
En general, la **herencia** sólo es imprescindible cuando necesitamos el **polimorfismo**.

Aún así, en algunos casos se puede evitar la herencia simplificando el problema, a costa de sacrificar otros aspectos del diseño.

Composición sobre herencia

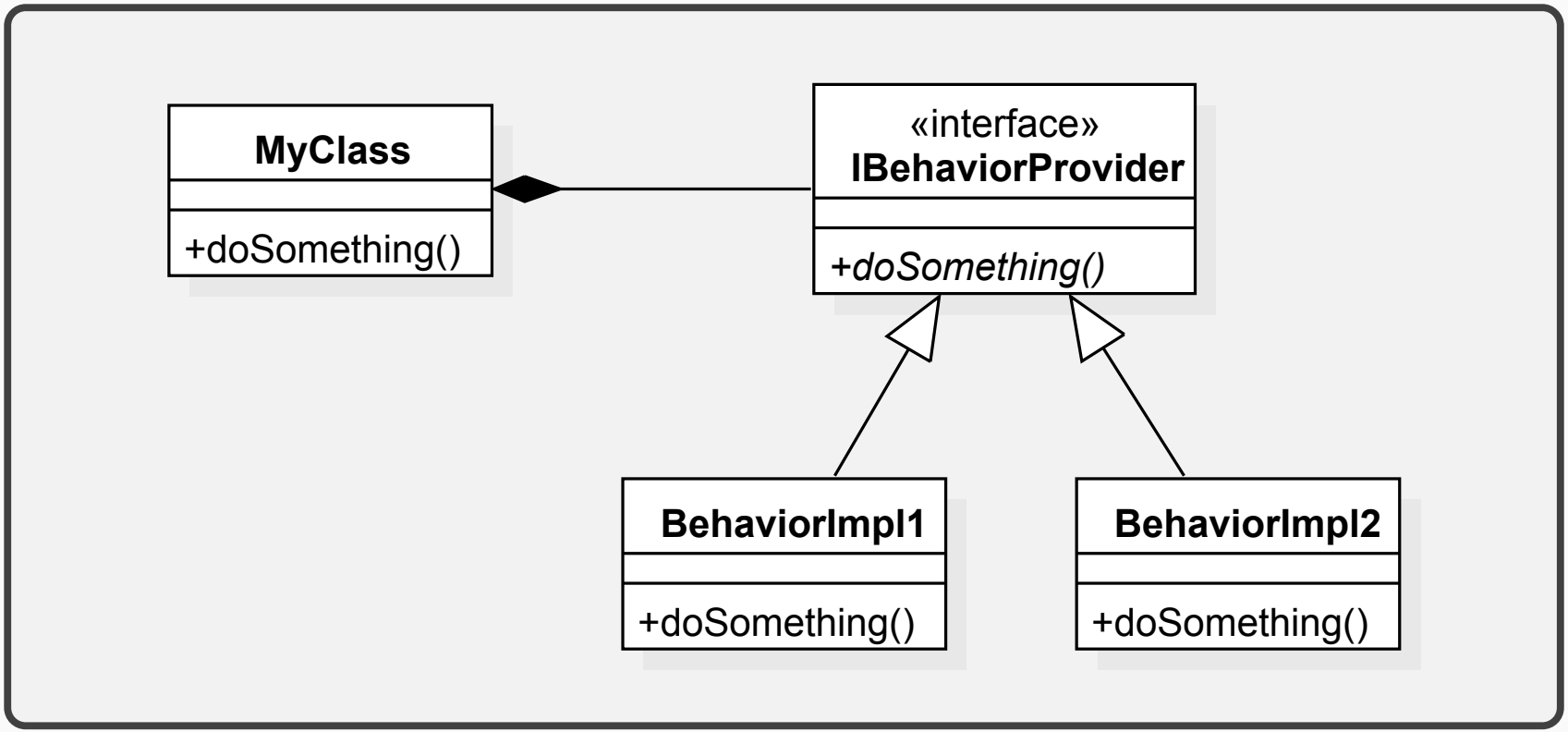
Otra forma de “heredar” un comportamiento es usando la composición, **delegando** la implementación de la responsabilidad a otra clase.

Si estamos diseñando el modelo de dominio, la nueva clase no tiene por qué persistir en base de datos.



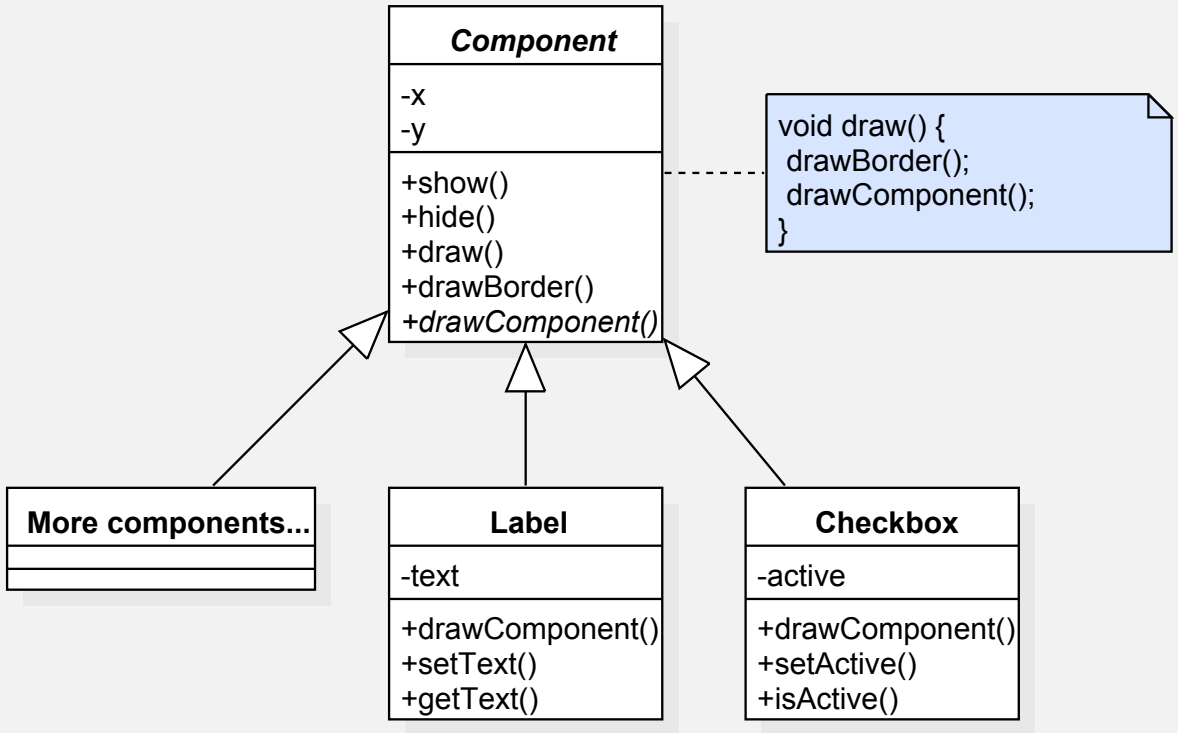
Composición sobre herencia

La composición ofrece mayor flexibilidad en tiempo de ejecución, ya que permite cambiar fácilmente el comportamiento de una clase.



Widgets 1.3.3

Supuesto: añadimos la funcionalidad `drawBorder()` a la clase base `Component`.



¿Problemas?

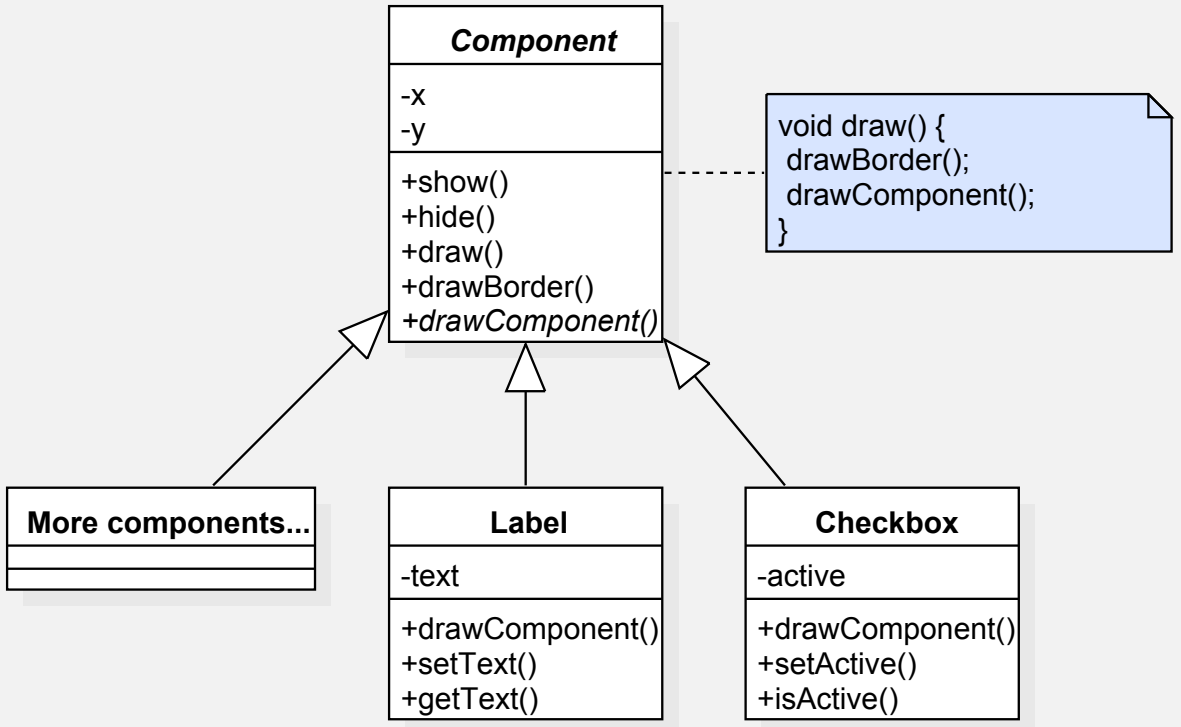
Problemas de este diseño

- Baja la cohesión de la clase Componente
- Dificulta el mantenimiento, p.ej. si queremos añadir nuevos tipos de borde

Veamos qué podría pasar si continuamos con este diseño...

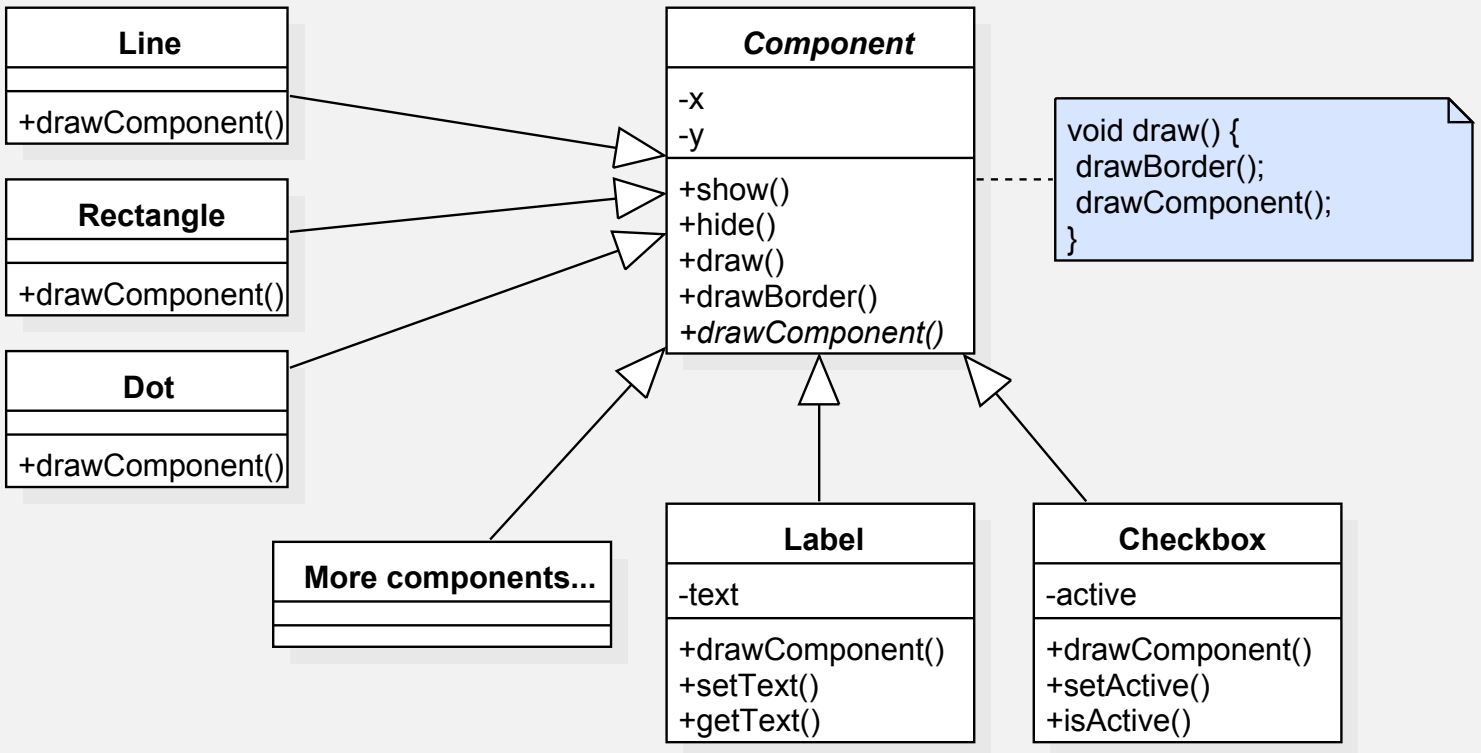
Widgets 1.3.4

Supuesto: para enriquecer aún más el interfaz, nos piden añadir líneas, puntos y algunas formas geométricas para aumentar las posibilidades de diseño gráfico.



¿Cómo modificamos el diseño?

Widgets 1.3.4



¿Qué problemas tiene este diseño?

Widgets 1.3.4

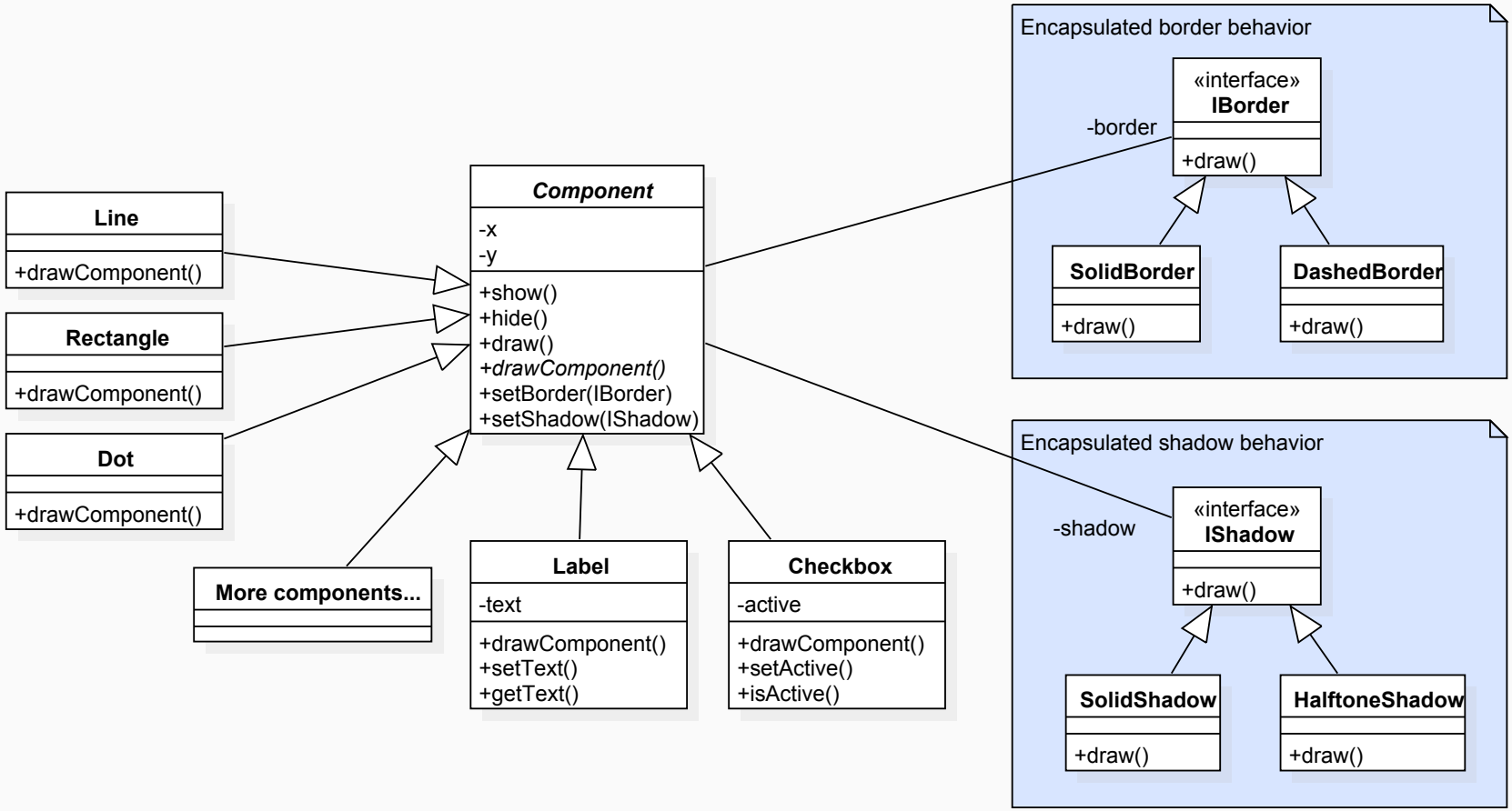
Problema

La herencia provoca un comportamiento no deseado:
los nuevos componentes también tienen bordes.

Solución

Podemos solucionar este problema usando la composición en lugar de la herencia.

Widgets 2.0



Ejemplo con Java Swing:

`https://docs.oracle.com/javase/tutorial/uiswing/components/border.html`

Principios SOLID

Principios SOLID

5 principios fundamentales de diseño [Martin, 2000]

- Principio de responsabilidad única (**S**ingle-responsibility)
- Principio abierto/cerrado (**O**pen/Closed)
- Principio de sustitución de Liskov (**L**iskov substitution)
- Principio de segregación de interfaces (**I**nterface segregation)
- Principio de inversión de dependencias (**D**ependency inversion)

Supuesto: implementación de la clase Fibonacci del ejercicio de la primera sesión de prácticas.

```
class Fibonacci {
    private ArrayList<Integer> series
        = new ArrayList<Integer>() {{ add(0); add(1); }};

    public int fibonacci(int n) {
        int len = series.size();
        if (n > len) {
            for (; len<n; len++)
                series.add(series.get(len-1) + series.get(len-2));
            return series.get(len-1);
        }
        else
            return series.get(n-1);
    }

    public void printSeries() {
        System.out.println(Arrays.toString(series.toArray()));
    }
}
```

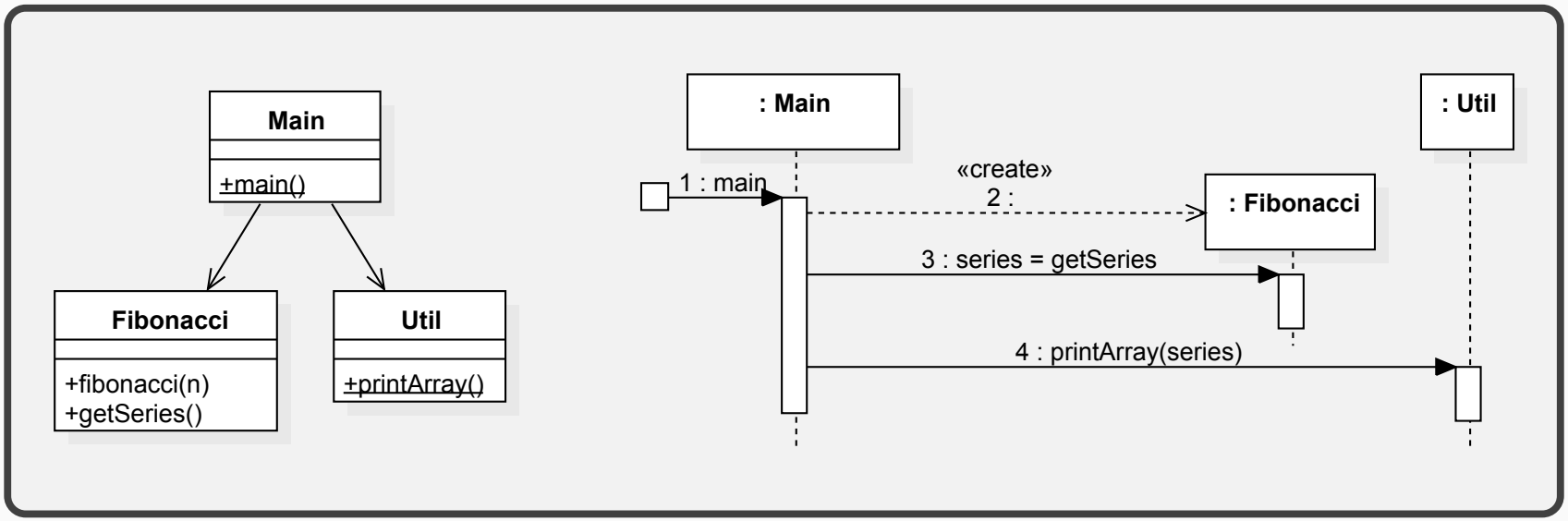
¿Hay algún problema en este diseño?

Principio de responsabilidad única

"Una clase sólo debería tener un motivo para cambiar."

En el ejemplo, la clase Fibonacci no debería ser la encargada de imprimir la secuencia.

Posible solución:



Supuesto: queremos imprimir un array usando distintos formatos (p.ej. CSV y JSON).

```
class Util {
    public void printArray(String format) {
        if (format.equals("csv")) {
            // Print as CSV
        }
        else if (format.equals("json")) {
            // Print as JSON
        }
    }
}
```

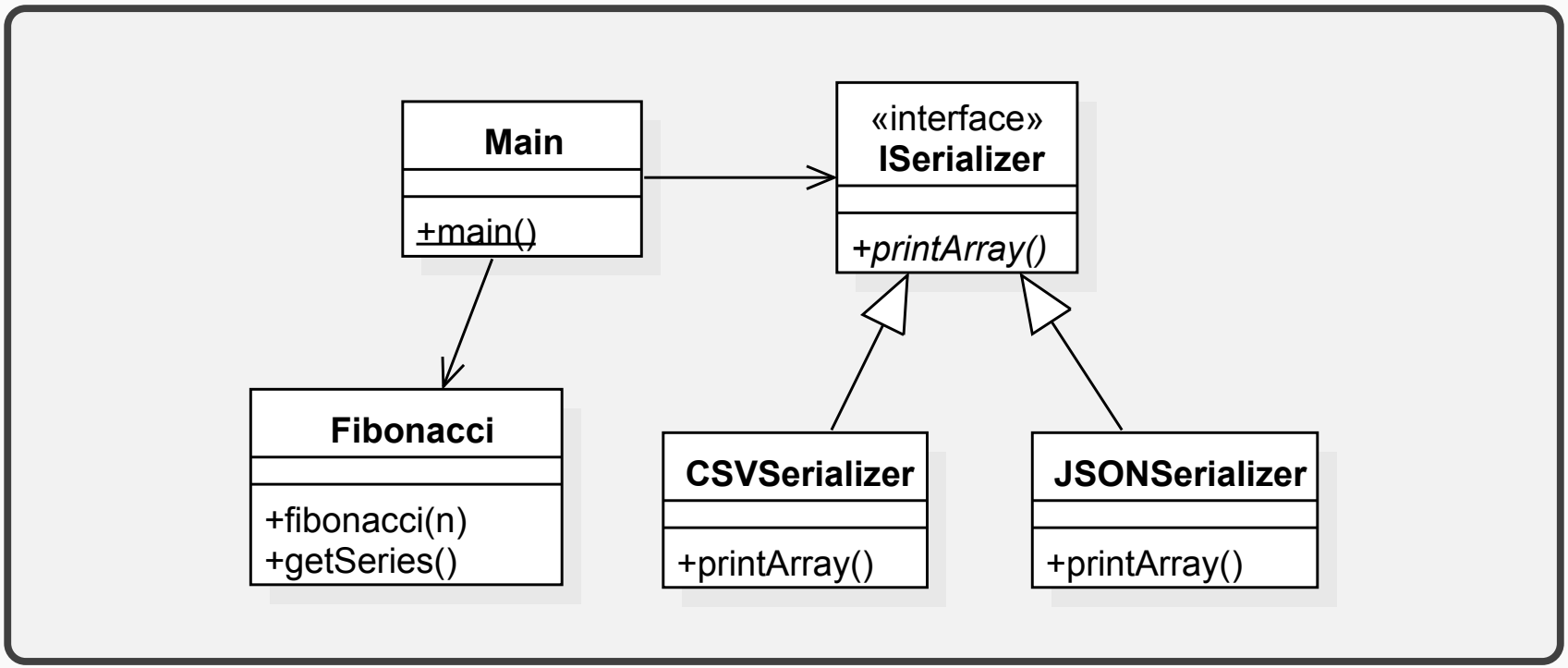
¿Hay algún problema en este diseño?

Principio abierto/cerrado

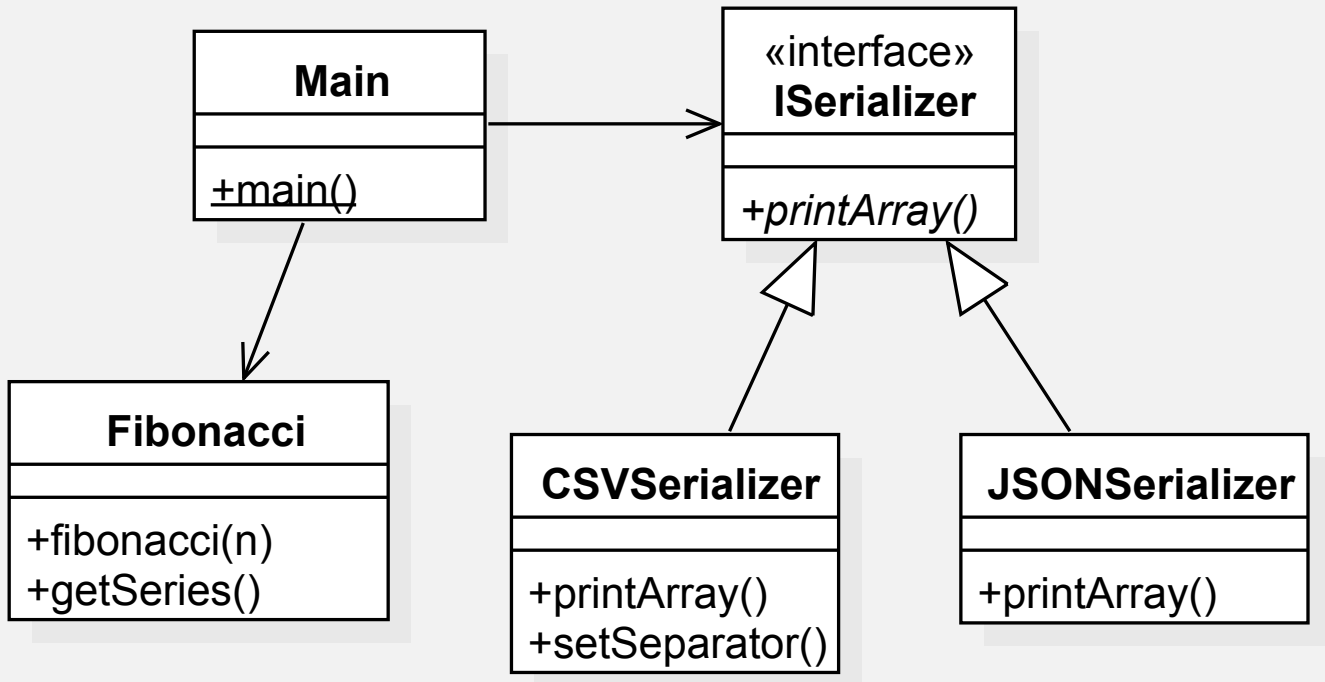
Principio abierto/cerrado

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

La forma tradicional de conseguirlo es mediante la herencia o implementación de interfaces.



Supuesto: añadimos la posibilidad de especificar el tipo de separador para archivos CSV.



¿Hay algún problema en este diseño?

Principio de sustitución de Liskov

```
class Main {
    public static void main(String args[]) {
        String format = args[1];
        ISerializer out = null;
        if (format.equals("csv")) {
            out = new CSVSerializer();
            ((CSVSerializer) out).setSeparator(":");
        }
        else if (format.equals("json"))
            out = new JSONSerializer();

        Fibonacci fib = new Fibonacci();
        // Computes the 10 first elements in the series
        fib.fibonacci(10);

        // Prints the array
        out.printArray(fib.getSeries());
    }
}
```

¿Cuál es el problema?

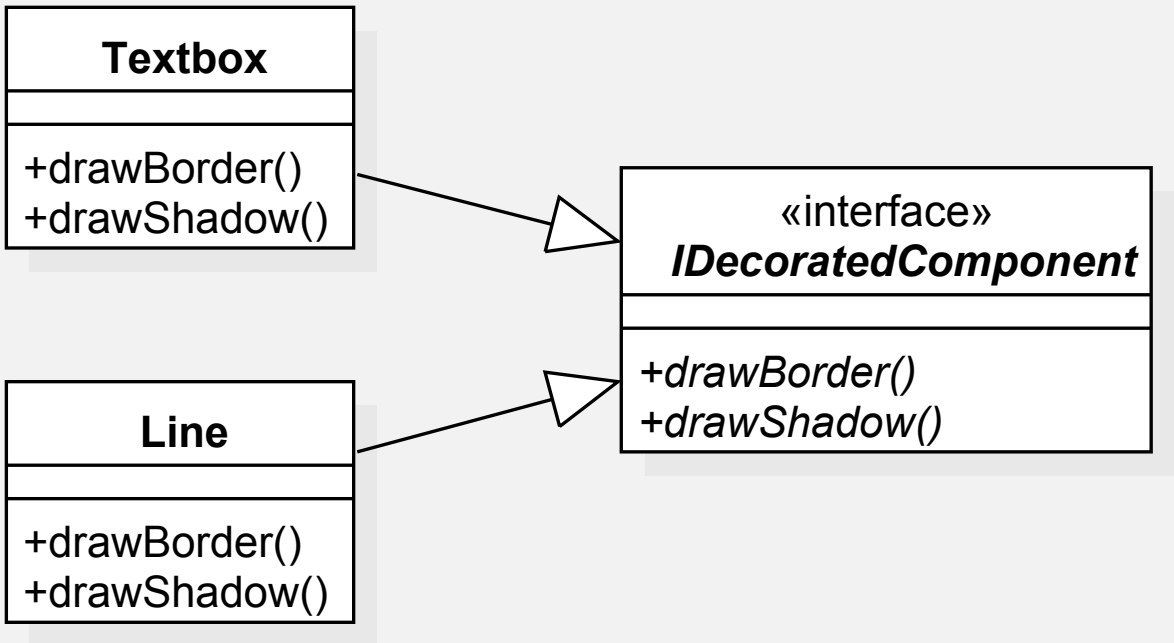
Principio de sustitución de Liskov

Principio de sustitución de Liskov

"Si S es un subtipo de T, entonces los objetos de tipo T en un programa deberían poder ser sustituidos por objetos de tipo S."

Al usar las subclases no deberíamos tener que conocer los detalles específicos de cada una de ellas, ni usarlas de forma distinta.

Supuesto: agrupamos las funcionalidades para dibujar bordes y sombras en un único interfaz `IDecoratedComponent`.

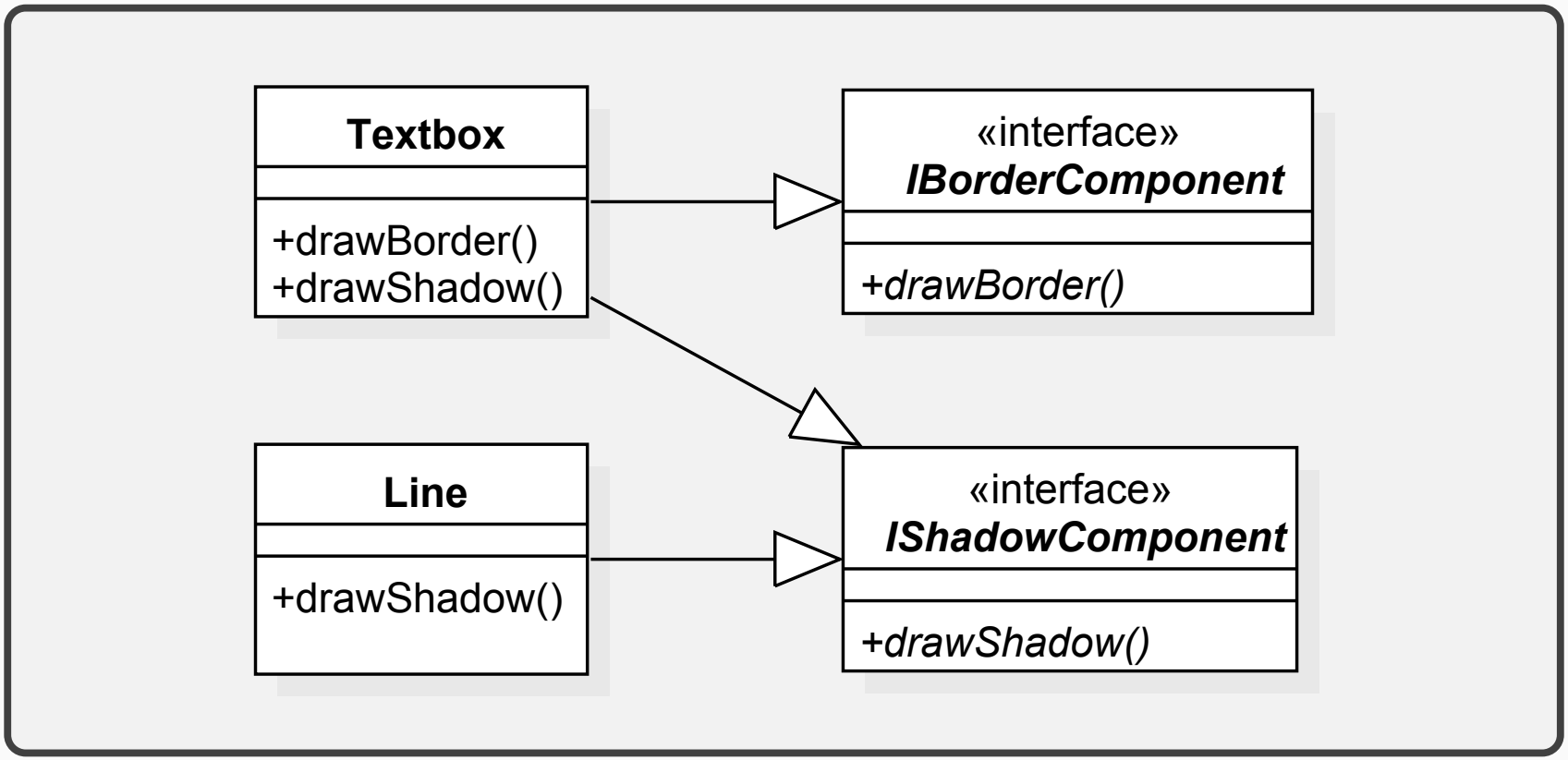


¿Hay algún problema en este diseño?

Principio de segregación de interfaces

Principio de segregación de interfaces

"Ninguna clase debería depender de métodos que no usa."



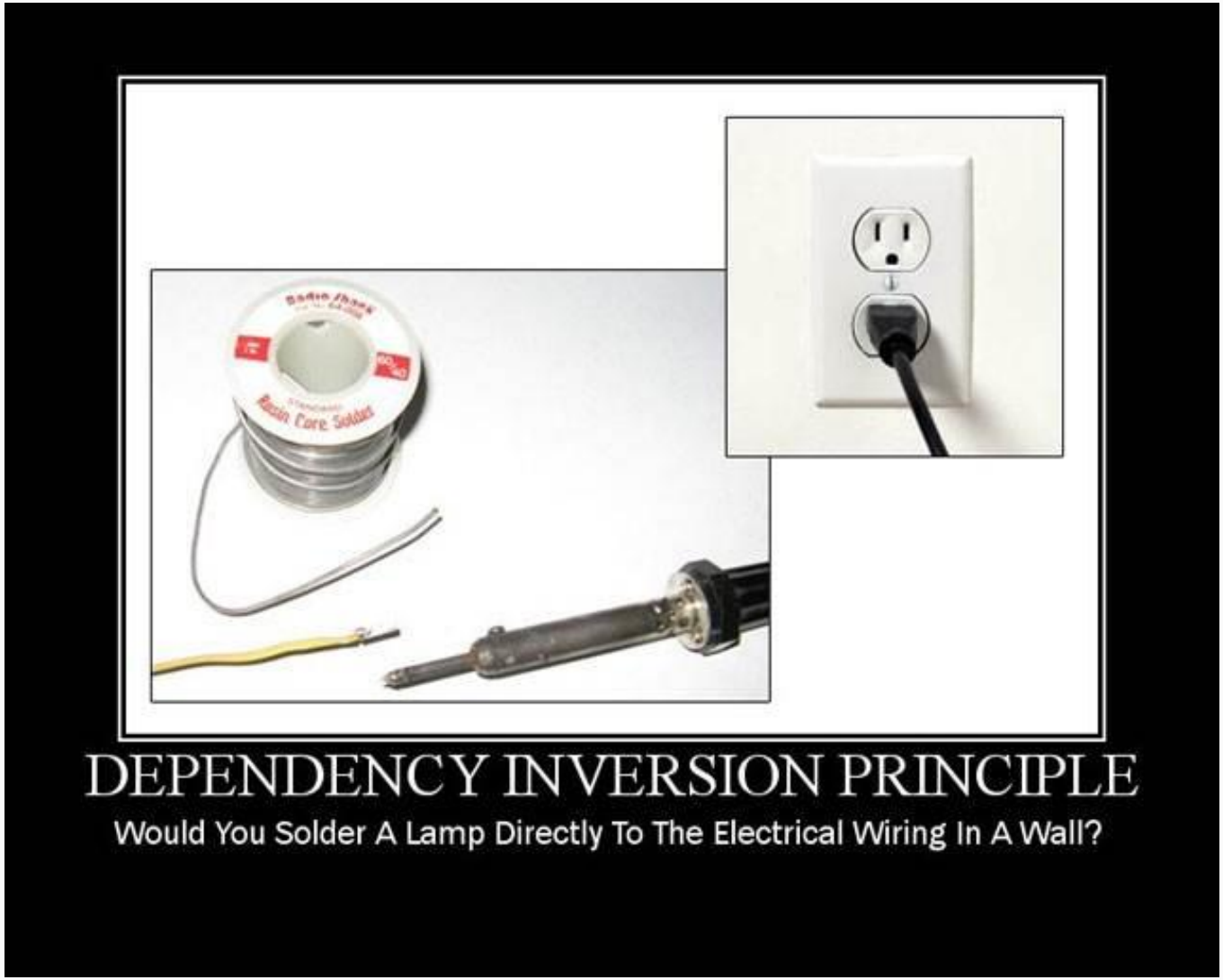
Supuesto: asumamos que queremos que la clase `Fibonacci` mantenga el método `printArray`.

En su código instanciamos la clase necesaria para serializar dependiendo del formato de salida.

```
class Fibonacci {
    public void printArray(String format) {
        ISerializer out = null;
        if (format.equals("csv"))
            out = new CSVSerializer();
        else if (format.equals("json"))
            out = new JSONSerializer();

        // Prints the array
        out.printArray(series.toArray());
    }
}
```

¿Problemas?

Principio de inversión de dependencias



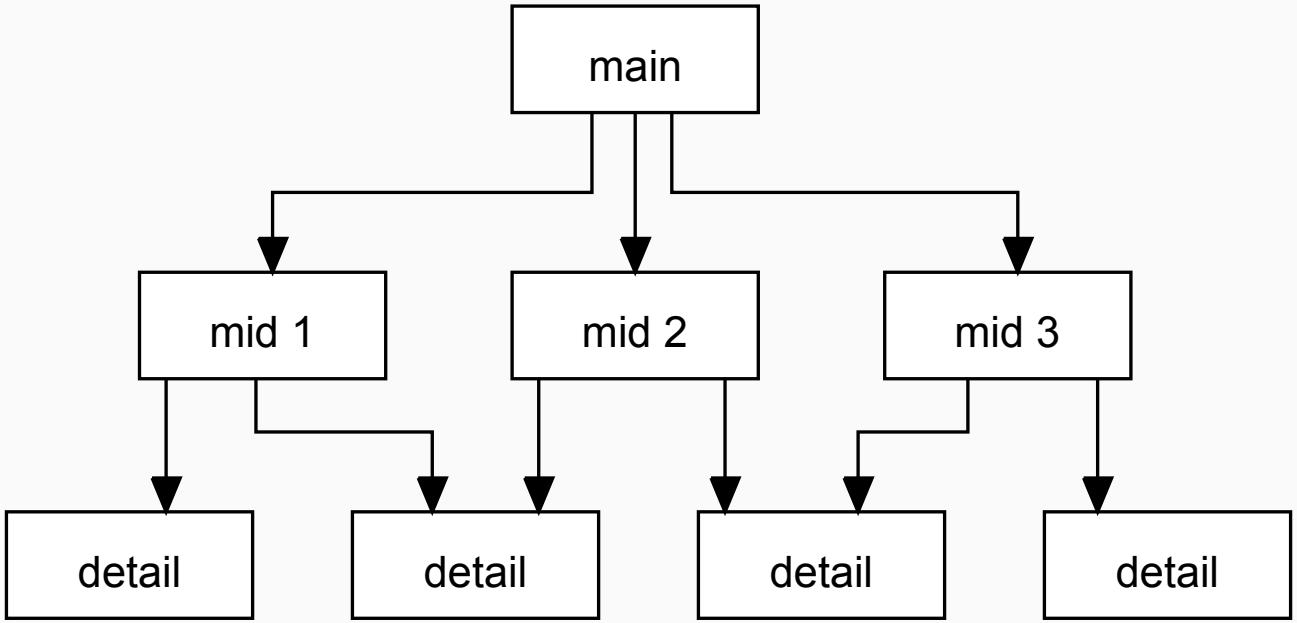
Principio de inversión de dependencias

Principio de inversión de dependencias

- “Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.”
- “Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.”

Principio de inversión de dependencias

En sistemas procedurales los módulos de alto nivel dependen de módulos de más bajo nivel.

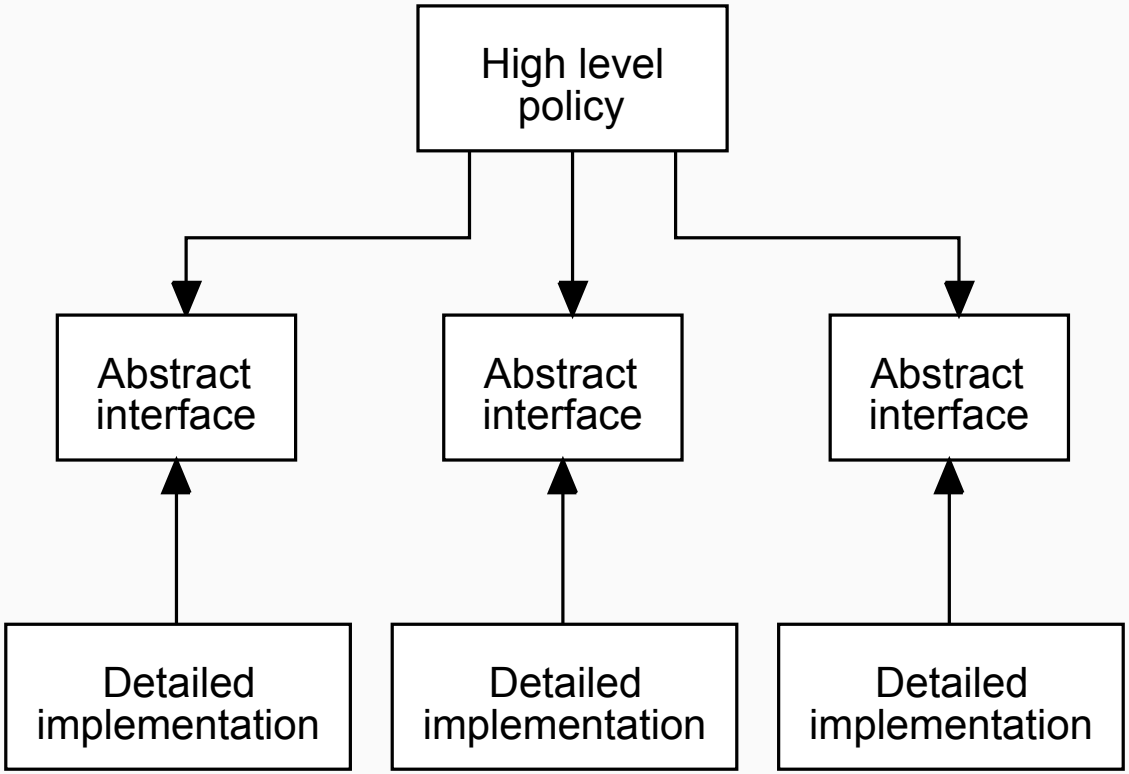


[Martin, 2000]

En sistemas orientados a objetos, los módulos de bajo nivel (implementaciones concretas) se consideran volátiles, ya que es más probable que cambien que las abstracciones.

Principio de inversión de dependencias

En la medida de lo posible se evitan estas dependencias invirtiéndolas, de manera que módulos de alto y bajo nivel dependen de abstracciones.



[Martin, 2000]

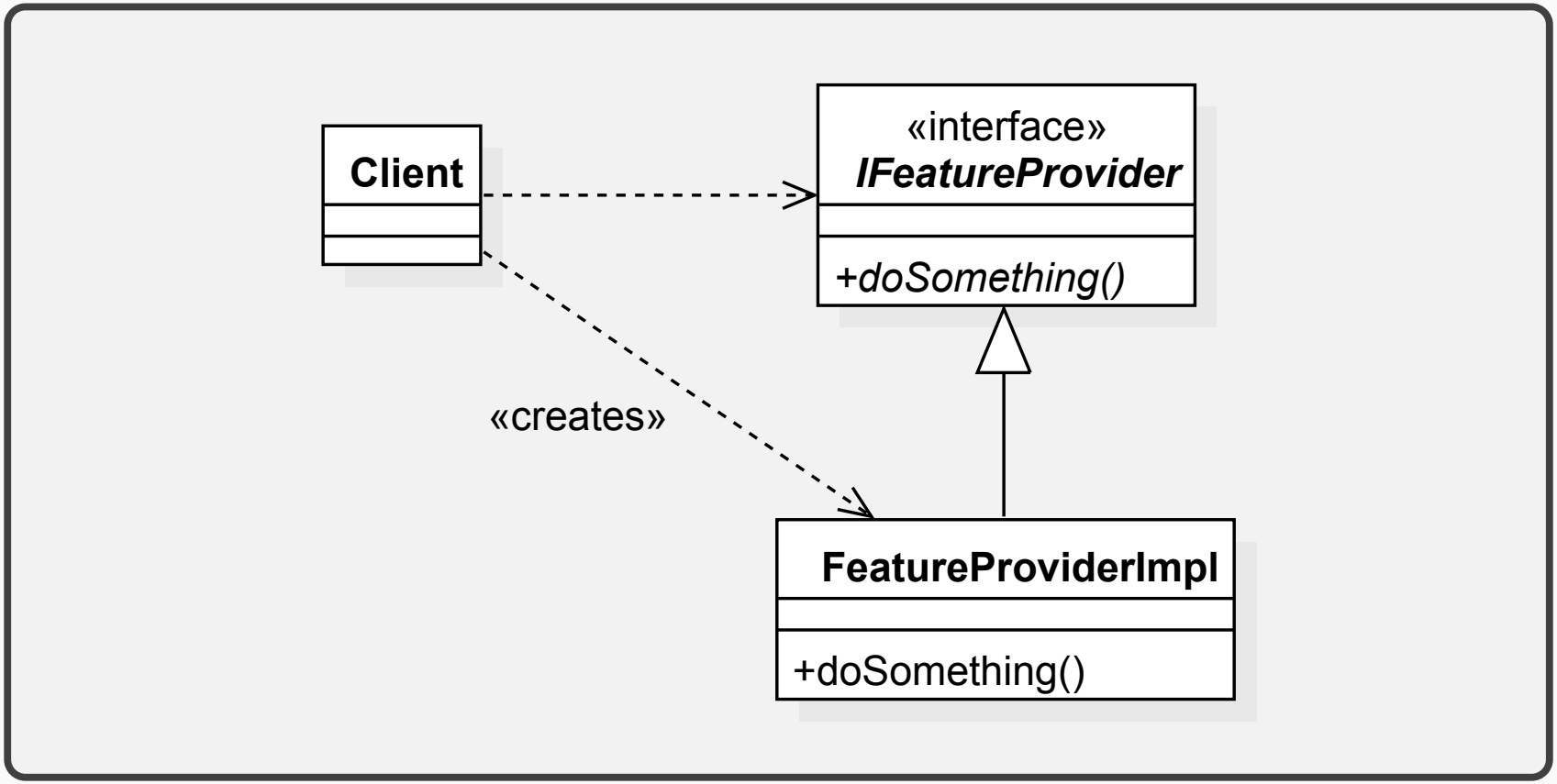
Principio de inversión de dependencias

Es preferible el acoplamiento con el interfaz, antes que con las clases que lo implementan:

```
class Fibonacci {
    public void printArray(ISerializer serializer) {
        serializer.printArray(this.series.toArray());
    }
}

class Main {
    public static void main(String args[]) {
        ISerializer serializer = // Initialize depending on args
        Fibonacci fib = new Fibonacci();

        // do stuff...

        fib.printArray(serializer);
    }
}
```

Inyección de dependencias

Inyección de dependencias



Problema

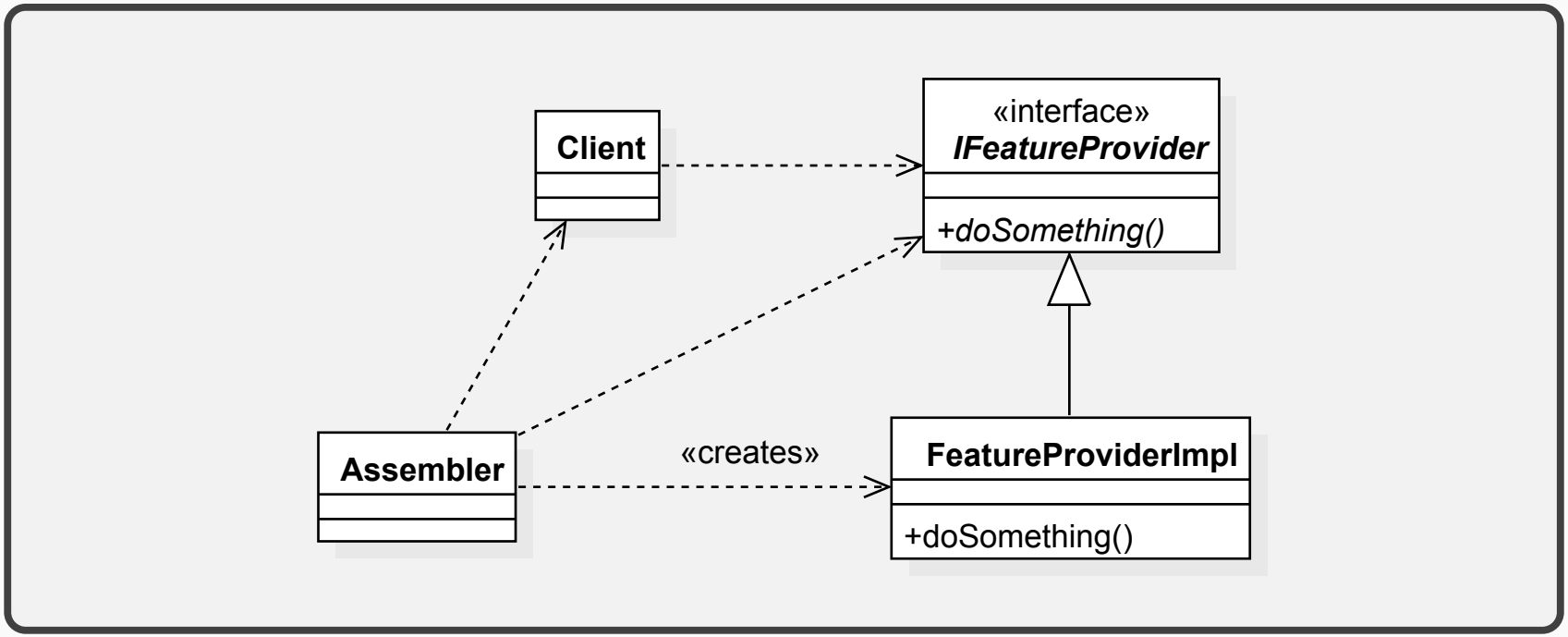
Cuando una clase A necesita una funcionalidad de otra clase B, y A crea una instancia de B, se establece una dependencia en tiempo de compilación que no se puede cambiar en tiempo de ejecución.

Inyección de dependencias

Solución

En lugar de crear la instancia directamente, la clase recibe una instancia que implementa el interfaz de la funcionalidad que necesita. Esta técnica se conoce como **Inyección de Dependencias**. [Fowler, 2004]

Inyección de dependencias



La inyección de dependencias es una forma de inversión de control: un objeto distinto se encarga de crear la instancia de una implementación concreta y proporcionársela al objeto que la usará.

Inyección de dependencias

Formas de inyección de dependencias:

- Inyección en el constructor
- Inyección con método *setter*
- Uso de un proveedor de servicios (*Service Locator*)

Inyección en el constructor

La clase cliente recibe una instancia del servicio en su constructor:

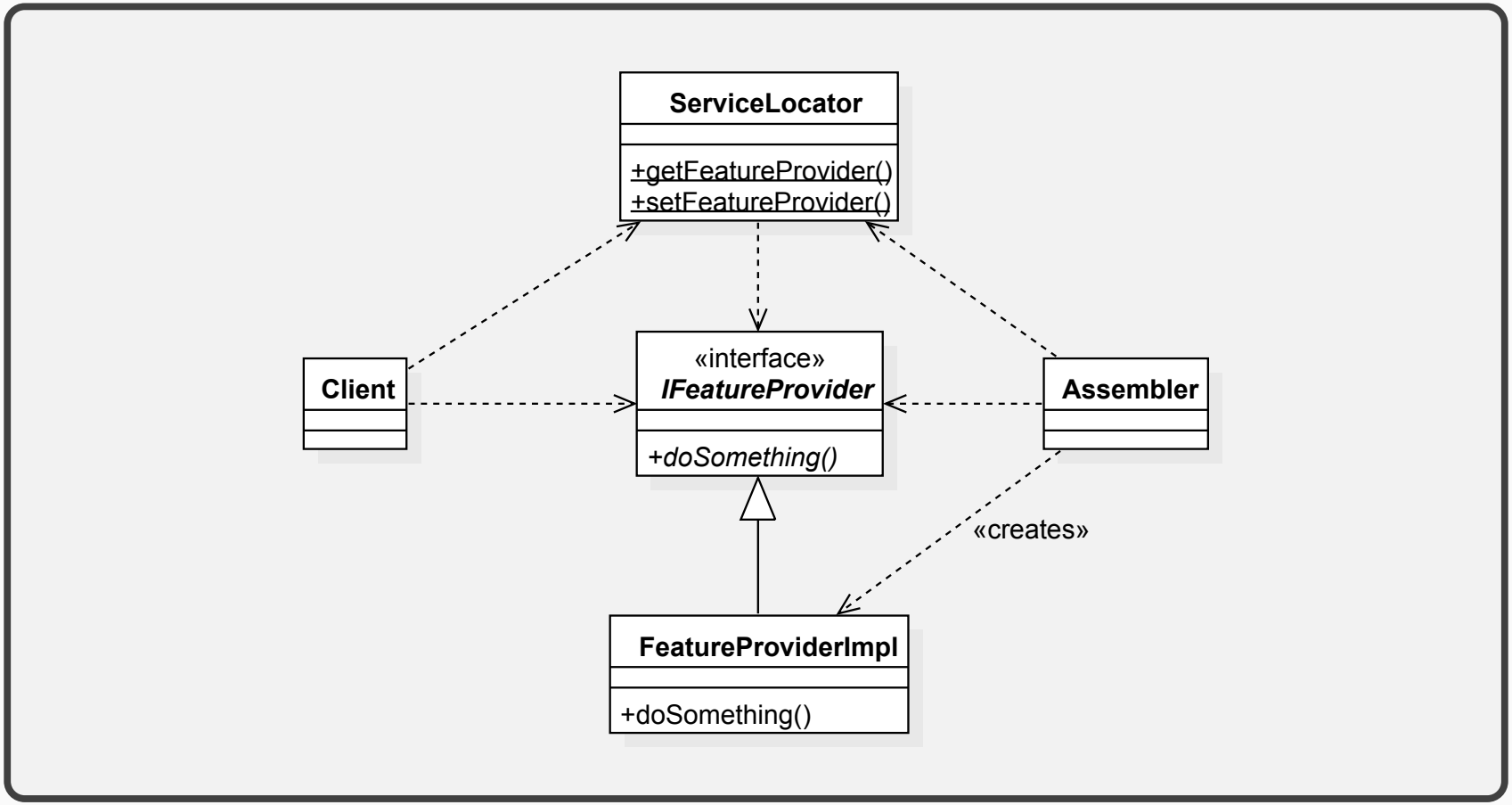
```
class Client {
    private IFeatureProvider provider = null;

    public Client(IFeatureProvider provider) {
        this.provider = provider;
    }
}
```

Inyección con método *setter*

La clase cliente recibe una instancia del servicio mediante un método setter:

```
class Client {
    private IFeatureProvider provider = null;

    public void setProvider(IFeatureProvider provider) {
        this.provider = provider;
    }
}
```

Uso de un proveedor de servicios



El objeto cliente obtiene los servicios de un *ServiceLocator*.

Uso de un proveedor de servicios

```
class Assembler {
    public void init() {
        IFeatureProvider provider = new FeatureProviderImpl();
        ServiceLocator.setFeatureProvider(provider);
    }
}

class Client {
    public void method() {
        IFeatureProvider provider =
            ServiceLocator.getFeatureProvider();
    }
}
```

Al seguir usando un objeto *Assembler* para crear las instancias tenemos más flexibilidad:

- Podemos probar el *ServiceLocator* independientemente
- Podemos inyectarle clases *mock* o *stub* para probar el cliente

Inyección de dependencias: comparativa

Inyección vs. *Service Locator*

- La inyección mediante constructor o método *setter* permite identificar mejor las dependencias, con un *Service Locator* hay que buscar sus llamadas en el código
- El *Service Locator* centraliza la inyección de dependencias en un único objeto, son más fáciles de gestionar

Inyección de dependencias: comparativa

Constructor vs. método *setter*

- La inyección mediante constructor permite crear objetos válidos desde la inicialización y proteger campos que no deben cambiar
- El método *setter* permite cambiar el comportamiento del objeto una vez inicializado
- **Lo importante es mantener la consistencia**
- Se pueden proporcionar los dos mecanismos, no son excluyentes

Inyección de dependencias: aspectos prácticos

Existen frameworks que permiten hacer la inyección de dependencias de forma automática, p.ej. Spring (`http://spring.io/`).

La configuración se realiza mediante archivos XML:

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <constructor-arg ref="bar"/>
        <constructor-arg ref="baz"/>
    </bean>

    <bean id="bar" class="x.y.Bar"/>

    <bean id="baz" class="x.y.Baz"/>
</beans>
```

Inyección de dependencias en Laravel

Laravel realiza la inyección de dependencias automáticamente en algunos tipos de objetos, como p.ej. en los controladores.

El *Service Container* se encarga de resolver las dependencias en los constructores:

```php
class TestController extends Controller
{
    protected $serializer;

    // Service Container will provide the $serializer instance
    public function __construct(ISerializer $serializer) {
        $this->serializer = $serializer;
    }
```

Ejemplo:
`https://github.com/cperezs/dss-dependency-injection`

Inyección de dependencias en Laravel

Para poder resolver las dependencias hay que registrarlas mediante *ServiceProviders*:

```
class SerializerServiceProvider extends ServiceProvider
{
    public function boot()
    {
        //
    }

    public function register()
    {
        $this->app->bind(
            'App\ISerializer',
            'App\CommaSerializer'
        );
    }
```

¿Preguntas?

Referencias I

Fowler, M. (2004).
Inversion of Control Containers and the Dependency Injection pattern.
https://martinfowler.com/articles/injection.html.
[Online; accessed on January 2018].

Martin, R. C. (2000).
Design Principles and Design Patterns.
https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.
[Online; accessed on January 2018].