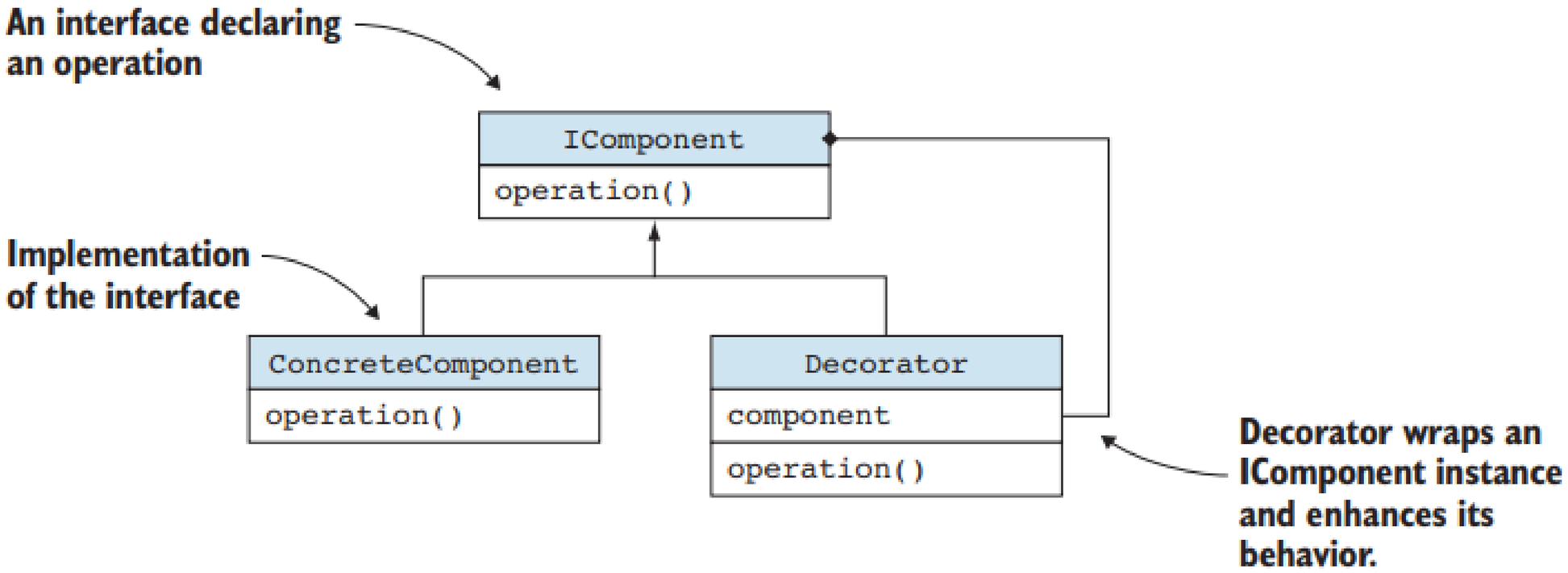
UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

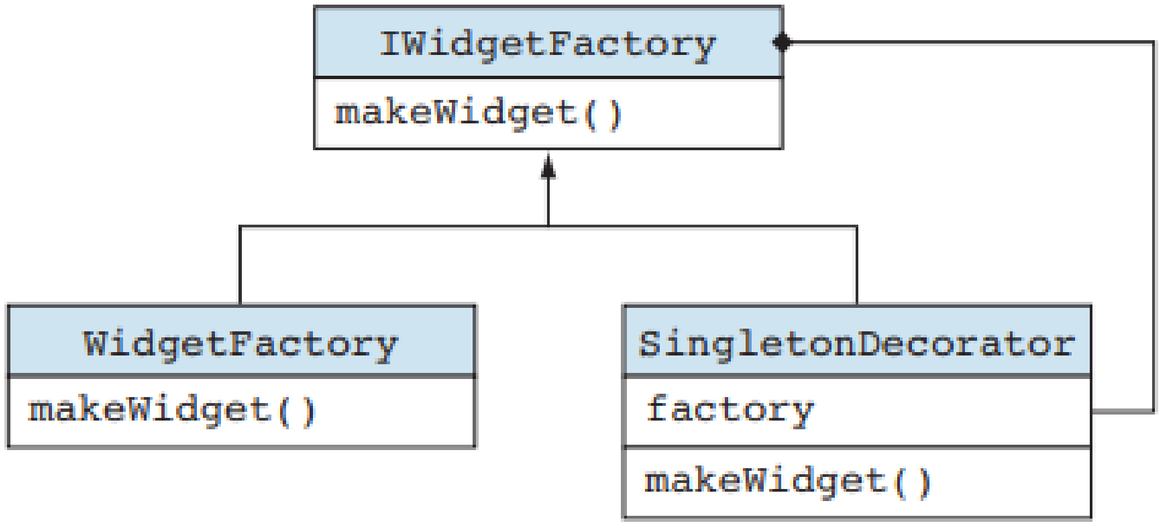
Aplicaciones avanzadas de los T. F.

Ya conocemos al patron decorador



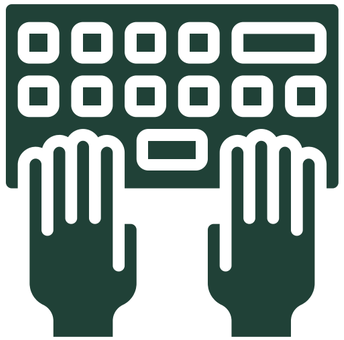
Supongamos que tenemos una interfaz **IWidgetFactory** que declara un método **makeWidget()** que devuelve un **Widget**. La implementación concreta, **WidgetFactory**, implementa el método para instanciar nuevos objetos **Widget**.

Supongamos que queremos reutilizar un Widget, de modo que en lugar de crear siempre uno nuevo, queremos crear solo uno y seguir devolviéndolo.



Sin modificar nuestro WidgetFactory, podemos crear un decorador llamado SingletonDecorator, que envuelve un IWidgetFactory, como se muestra en el siguiente listado, y extiende su comportamiento para garantizar que solo se cree un único Widget.

Implementen esto...



```
class Widget { }

interface IWidgetFactory {
    makeWidget(): Widget;
}

class WidgetFactory implements IWidgetFactory {
    public makeWidget(): Widget {
        return new Widget();
    }
}

class SingletonDecorator implements IWidgetFactory {
    private factory: IWidgetFactory;
    private instance: Widget | undefined = undefined;

    constructor(factory: IWidgetFactory) {
        // Completar...
    }

    public makeWidget(): Widget {
        // Completar...
    }
}
```

Solucion

La ventaja de usar este patrón es que apoya el **Principio de Responsabilidad Única**, que establece que una clase debe tener solo una responsabilidad.

En este caso, el **WidgetFactory** es responsable de crear widgets, mientras que el **SingletonDecorator** es responsable del comportamiento de singleton.
Si queremos múltiples instancias, usamos el **WidgetFactory** directamente. Si queremos una sola instancia, usamos **SingletonDecorator**.

```
class Widget { }

interface IWidgetFactory {
    makeWidget(): Widget;
}

class WidgetFactory implements IWidgetFactory {
    // WidgetFactory crea un nuevo widget.
    public makeWidget(): Widget {
        return new Widget();
    }
}

class SingletonDecorator implements IWidgetFactory {
    // SingletonDecorator envuelve a un IWidgetFactory
    private factory: IWidgetFactory;
    private instance: Widget | undefined = undefined;

    constructor(factory: IWidgetFactory) {
        this.factory = factory;
    }

    // makeWidget() implementa la logica de singleton
    // y asegura que solo una widget sea creado.
    public makeWidget(): Widget {
        if (this.instance == undefined) {
            this.instance = this.factory.makeWidget();
        }
        return this.instance;
    }
}
```

Decoradores Funcionales

Veamos cómo podemos simplificar esta implementación, nuevamente usando funciones tipadas.

Primero, eliminemos la interfaz **IWidgetFactory** y reemplacémosla con un tipo funcional.
Ese sería el tipo de una función que no toma argumentos y devuelve un Widget:

```
() => Widget
```

Ahora podemos reemplazar nuestra clase **WidgetFactory** con una simple función, **makeWidget()**.
Siempre que antes hubiéramos usado una **IWidgetFactory**, pasando una instancia de **WidgetFactory**,
ahora requerimos una función del tipo **() => Widget** y pasamos **makeWidget()**.

Antes

```
class Widget { }

interface IWidgetFactory {
    makeWidget(): Widget;
}

class WidgetFactory implements IWidgetFactory {
    // WidgetFactory crea un nuevo widget.
    public makeWidget(): Widget {
        return new Widget();
    }
}
```

Despues

```
class Widget { }

type WidgetFactory = () => Widget;

function makeWidget(): Widget {
    return new Widget();
}

function use10Widgets(factory: WidgetFactory) {
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
}

use10Widgets(makeWidget);
```

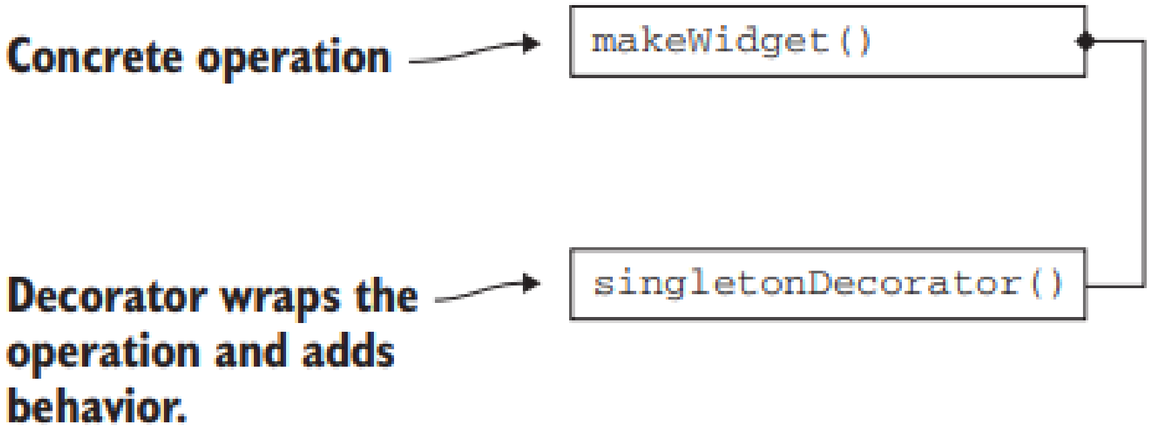
Con el **factory de widgets funcional**, usamos una técnica muy similar al patrón de strategy que vimos anteriormente: obtenemos una función como argumento y la llamamos cuando es necesario.

Ahora veamos cómo podemos añadir el comportamiento de singleton.

Proporcionamos una nueva función, **singletonDecorator()**, que toma una función de tipo **WidgetFactory** y devuelve otra función de tipo **WidgetFactory**.

Recordemos que una **lambda** es una función sin nombre, que podemos retornar desde otra función.

Ahora nuestro **decorador** tomará un **factory** y lo usará para *construir una nueva función que gestione el comportamiento de singleton*.



```typescript
class Widget { }

type WidgetFactory = () => Widget;

function makeWidget(): Widget {
    return new Widget();
}

function singletonDecorator(factory: WidgetFactory): WidgetFactory {
    let instance: Widget | undefined = undefined;

    // Devuelve un lambda que implementa el comportamiento de singleton
    // y utiliza el factory dado para crear un Widget.
    return (): Widget => {
        if (instance == undefined) {
            instance = factory();
        }
        return instance;
    };
}

function use10Widgets(factory: WidgetFactory) {
    for (let i = 0; i < 10; i++) {
        let widget = factory();
        /* ... */
    }
}

// Debido a que singletonDecorator() devuelve una WidgetFactory,
// podemos pasarlo como argumento a use10Widgets().
use10Widgets(singletonDecorator(makeWidget));
```

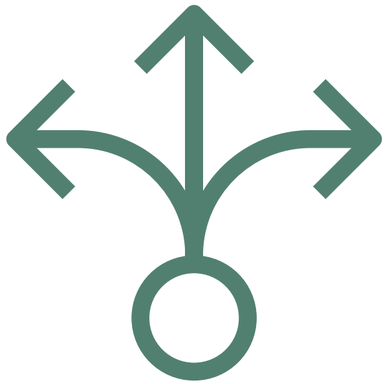
Ahora, en lugar de construir 10 objetos Widget, **use10Widgets()** llamará al lambda, que reutilizará la misma instancia de Widget para todas las llamadas.

Este código reduce el número de componentes de una interfaz y dos clases, cada una con un método (la operación concreta y el decorador), a solo dos funciones.

Tanto el enfoque orientado a objetos como el funcional implementan el Patrón Decorador, pero con distintas compensaciones.

La versión orientada a objetos, que utiliza interfaces y clases, es más verbosa, mientras que el enfoque funcional es más conciso al basarse en tipos funcionales y funciones de orden superior (posible gracias a las "funciones de primera clase").

La elección entre ambos depende de la complejidad: la implementación funcional es ideal para decorar el comportamiento de una sola función, mientras que la versión orientada a objetos es necesaria cuando el componente a decorar tiene múltiples métodos definidos en una interfaz, ya que esta complejidad no puede ser representada por un único tipo funcional.



Closures

¿***Por qué*** esto funciona?

¿Será que la variable **instance** es distinta para todas las funciones retornadas?

¿Cómo estamos guardando la *instancia* si **estamos devolviendo funciones distintas**?

¿**Instance** siempre inicia como **undefined** por lo tanto siempre **se crea un nuevo widget**... ?

```
function singletonDecorator(factory: WidgetFactory): WidgetFactory {
  let instance: Widget | undefined = undefined;

  return (): Widget => {
    if (instance == undefined) {
      instance = factory();
    }
    return instance;
  };
}
```

Incluso después de que retornemos de **singletonDecorator()**, la variable instance sigue viva, ya que fue "*capturada*" por el lambda, lo que se conoce como **captura de lambda** (lambda capture).

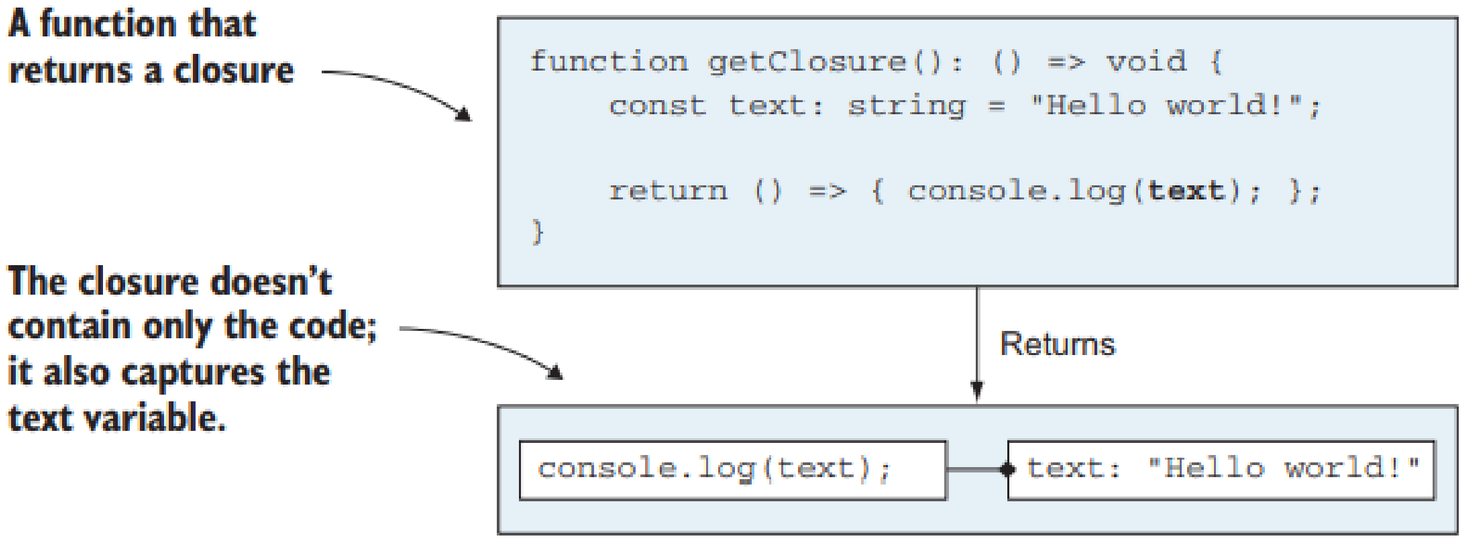
Closures: Es una variable externa capturada dentro de un lambda. Los lenguajes de programación implementan las capturas de lambda a través de **clausuras** (closures). Una clausura es algo más que una simple función: también registra el entorno en el que la función fue creada, de modo que puede mantener un estado entre llamadas.

```
function singletonDecorator(factory: WidgetFactory): WidgetFactory {
  let instance: Widget | undefined = undefined;

  return (): Widget => {
    if (instance == undefined) {
      instance = factory();
    }
    return instance;
  };
}
```

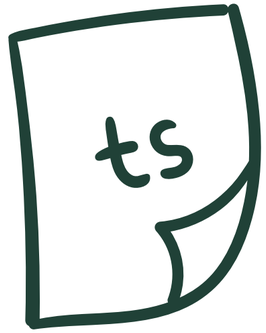
En mi opinión la mejor definición es: Las **clausuras** son funciones que hacen referencia a variables independientes (libres). En otras palabras, la función definida en la clausura "recuerda" el entorno en el que fue creada.

Una clausura tiene acceso a las variables del entorno que la encierra, en un lenguaje mas técnico, el closure tiene acceso al scope de la función que la encierra.



Las clausuras solo tienen sentido si tenemos funciones de orden superior.

Veamos el codigo



Fin