Programación Distribuida

Inversion of Control (IoC)

JAIME SALVADOR

AGENDA

- **Requisitos**
- Introducción
- Inversion of Control

Requisitos

Conocimientos previos de

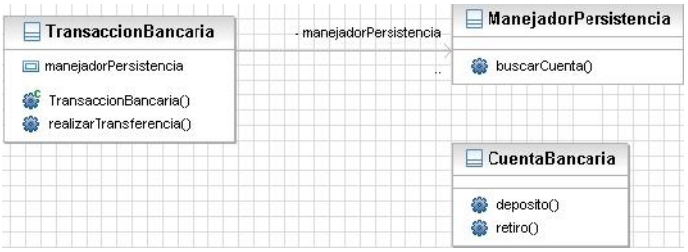
- Interfaces
- Clases
- Herencia

AGENDA

- Requisitos
- **Introducción**
- Inversion of Control

Introducción

- Consideremos el siguiente ejemplo:
  - Supongamos que una compañía está realizando un sistema bancario, el sistema debe permitir realizar transferencias entre cuentas del banco
  - Luego de un estudio previo se procede a crear el siguiente diagrama



Introducción

```
public class TransaccionBancaria {
     private ManejadorPersistencia manejadorPersistencia = null;

     public TransaccionBancaria() {
        manejadorPersistencia = new ManejadorPersistencia();
     }
     public void realizarTransferencia(String cuenta1, String cuenta2,float cantidad) {
          CuentaBancaria objCuenta1 = manejadorPersistencia.buscarCuenta(cuenta1);
          CuentaBancaria objCuenta2 = manejadorPersistencia.buscarCuenta(cuenta2);

          objCuenta1.deposito(cantidad);
          objCuenta2.retiro(cantidad);
     }
}
```

Introducción

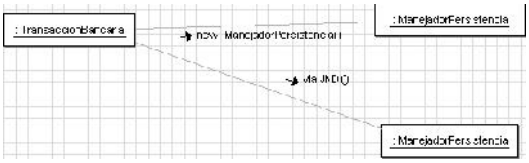
```
public class ManejadorPersistencia {
     public CuentaBancaria buscarCuenta(String numeroCuenta){
        ...
     }
}
public class CuentaBancaria {
     public void deposito(float cantidad) {
     ...
     }
     public void retiro(float valor) {
     ...
     }
}
```

Introducción

Quién llama a quién?

Introducción

- El principal problema con la clase `TransaccionBancaria` es cómo obtiene la referencia a `ManejadorPersistencia`
- Cada instancia de la clase `TransaccionBancaria` es responsable de la creación de la instancia de `ManejadorPersistencia`

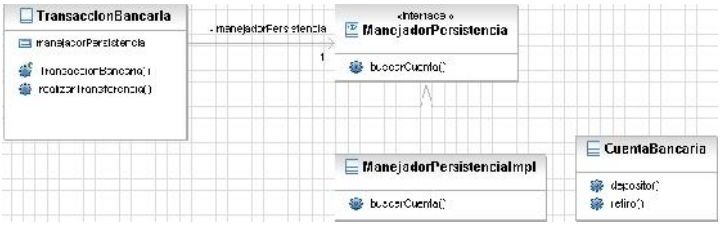


Introducción

- Para resumir el problema: el código está altamente **acoplado**
- `TransaccionBancaria` está altamente acoplada con la clase `ManejadorPersistencia`
- Es imposible tener una instancia de `ManejadorPersistencia` sin tener una instancia de `TransaccionBancaria`
- Para poder hacer algo interesante en el código, las clases necesitan saber de otras clases de alguna forma. El acoplamiento entre clases es necesario, pero hay que tratarlo con cuidado

Introducción

- Una técnica para reducir el nivel de acoplamiento es ocultar los detalles de la implementación detrás de interfaces, de esta forma la implementación de la interface puede ser aislada sin impactar a las clases clientes



Introducción

```
public class TransaccionBancaria {
     private ManejadorPersistencia = null;

     public TransaccionBancaria() {
        manejadorPersistencia = new ManejadorPersistenciaImpl();
     }

     public void realizarTransferencia(String cuenta1, String cuenta2,float cantidad) {
          CuentaBancaria objCuenta1 = manejadorPersistencia.buscarCuenta(cuenta1);
          CuentaBancaria objCuenta2 = manejadorPersistencia.buscarCuenta(cuenta2);

          objCuenta1.deposito(cantidad);
          objCuenta2.retiro(cantidad);
     }
}
```

Introducción

- Ocultar la implementación de una clase detrás de una interface es dar un paso en la dirección correcta
- Sin embargo en donde la mayoría de programadores falla es en cómo obtener una instancia de `ManejadorPersistencia`

```
public TransaccionBancaria() {
   manejadorPersistencia = new ManejadorPersistenciaImpl();
}
```

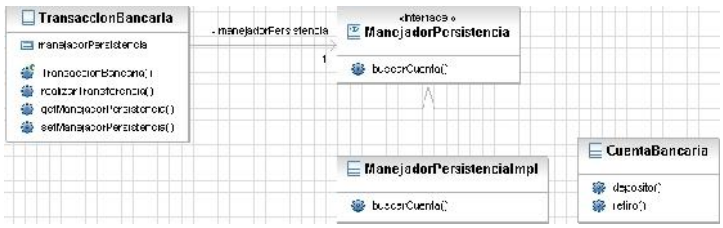
- Esta no es una mejor solución a la anterior debido a que la clase `TransaccionBancaria` sigue obteniendo solamente instancias de `ManejadorPersistenciaImpl` y no otro tipo de implementación

Introducción

Cómo se obtienen referencias?

Introducción

- La pregunta obvia que se obtiene a partir de los ejemplos anteriores es si la clase `TransaccionBancaria` es responsable de obtener una referencia a `ManejadorPersistencia` o de alguna forma la referencia se la pasa a `TransaccionBancaria`?



Introducción

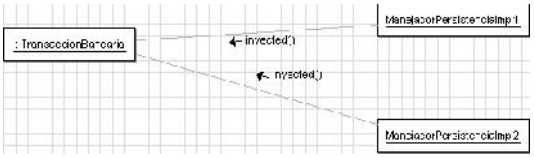
```
public class TransaccionBancaria {
     ManejadorPersistencia = null;

     public TransaccionBancaria() {
     }
     ...
     public ManejadorPersistencia getManejadorPersistencia() {
          return manejadorPersistencia;
     }
     public void setManejadorPersistencia(ManejadorPersistencia manejadorPersistencia) {
        this.manejadorPersistencia = manejadorPersistencia;
     }
}
```

Introducción

Diferencia con las soluciones anteriores?

Introducción

- A la clase `TransaccionBancaria` se le pasa una instancia de `ManejadorPersistencia`
- `TransaccionBancaria` no es responsable de recuperar una instancia de `ManejadorPersistencia`
- Como `TransaccionBancaria` conoce al manejador de persistencia solo a través de su interface, se puede pasar cualquier implementación de la clase



Introducción

Todo lo anterior se lo conoce como **Dependency Injection** (DI) también conocido como **Inversion of Control** (IoC)

La responsabilidad de coordinar la colaboración entre objetos dependientes es transferida fuera de los objetos implicados

AGENDA

- Requisitos
- Introducción
- **Inversion of Control**

IoC

- El objetivo de IoC es proporcionar un mecanismo simple para la provision de dependencias y el manejo de las mismas (dependencias) a través de un ciclo de vida.
- Un component que require una determinada dependencia se denomina *Dependent object* (Objeto dependiente) o también *target* (objetivo)
- Dependency Injection (DI)
- Dependency Lookup

IoC: Tipos

- Dependency Lookup

  Similar a la forma tradicional de obtener referencias a objetos

  El componente dependiente debe obtener la referencia al objeto

- Dependency Injection (DI)

  Más flexible que la anterior

  Las dependencias son proporcionadas por un tercero (IoC Container)

IoC: Dependency Lookup

- Dependency Pull

  Las dependencias se las busca en un registro

  ej. EJB 2.1 o anterior a través de JNDI (fuente Apress – Pro Spring 4ta. Ed.)

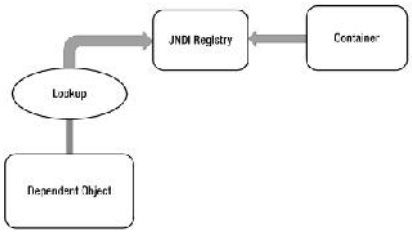


Figure 3-1. Dependency Pull via JNDI lookup

IoC: Dependency Lookup

- Contextualized Dependency Lookup (CDL)

  Similar al anterior, las dependencias se las busca directamente en el contenedor

  (fuente Apress – Pro Spring 4ta. Ed.)

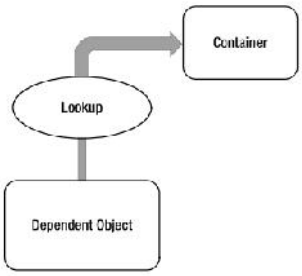


Figure 3-2. Contextualized Dependency Lookup

IoC: Dependency Injection (DI)

- Constructor Dependency Injection

  Las dependencias de un componente se proporcionan en el constructor

  (fuente Apress – Pro Spring 4ta. Ed.)

```
public class ConstructorInjection {
    private Dependency dependency;

    public ConstructorInjection(Dependency dependency) {
        this.dependency = dependency;
    }
```

IoC: Dependency Injection (DI)

- Stter Dependency Injection

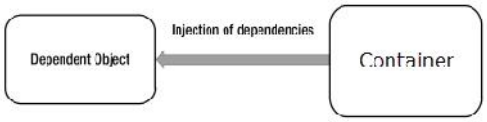
  Las dependencias de un componente se proporcionan vía el método SET de un componente JavaBean

  Es la más utilizada

  (fuente Apress – Pro Spring 4ta. Ed.)

```
public class SetterInjection {
    private Dependency dependency;

    public void setDependency(Dependency dependency) {
        this.dependency = dependency;
    }

    @Override
    public String toString() {
        return dependency.toString();
    }
}
```

IoC: Dependency Injection (DI)



PREGUNTAS?