INYECCIÓN DE DEPENDENCIAS

CASO PRÁCTICO: Implementación de una orquesta (2)

Implementación de Orquesta v.08

Retocamos un poco la implementación de la orquesta para introducir partituras...

```
public class Partitura
             implements Iterable<String> {
  private String score;
  public Partitura(String score) {
    this.score = score;
  }
  public Iterator<String> iterator() {
    return Arrays.stream(score.split(" "))
           .iterator();
  }
}
```

```
public abstract class Instrumento {
  protected String nombre;
  protected Partitura partitura =
              new Partitura("G D7 C D7 G");

  public abstract String tocar(String nota);
  public String tipo() {
    return getClass().getSimpleName().toLowerCase();
  }
  public String afinar() {
    return "Afinando "+nombre;
  }
  public String tocarPartitura() {
    StringBuffer sb = new StringBuffer();
    partitura.forEach(
      nota -> sb.append(tocar(nota))
    );
    return sb.toString();
    //for (String nota: partitura)
    //  tocar(nota);
  }
}
```

```
public class Viento extends Instrumento {
  public Viento(String nombre) {
    this.nombre = nombre;
  }
  public String tocar(String nota) { soplar(nota); return nota; }
  private void soplar() { System.out.println(nombre+" soplando "+partitura); }
  private void soplar(String nota) { System.out.println(nombre+" soplando "+nota); }
}

public class Cuerda extends Instrumento {
  public Cuerda(String nombre) {
    this.nombre = nombre;
  }
  public String tocar(String nota) { rasgar(nota); return nota; }
  private void rasgar() { System.out.println(nombre+" rasgando "+partitura); }
  private void rasgar(String nota) { System.out.println(nombre+" rasgando "+nota); }
}

public class Percusion extends Instrumento {
  public Percusion(String nombre) {
    this.nombre = nombre;
  }
  public String tocar(String nota) { golpear(nota); return nota; }
  private void golpear() { System.out.println(nombre+ "golpeando "+partitura); }
  private void golpear(String nota) { System.out.println(nombre+" golpeando "+nota); }
}
```

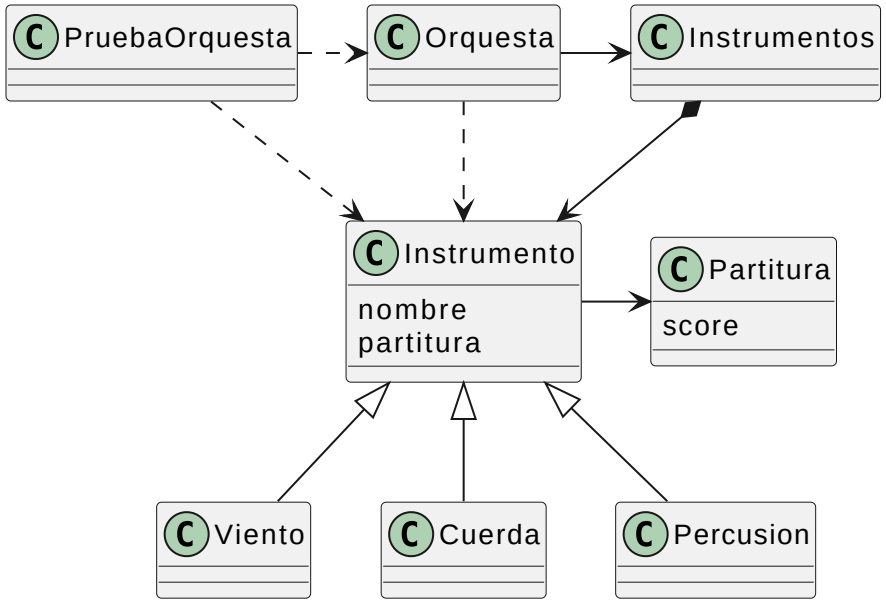
```java
public class Orquesta
             implements Iterable<Instrumento> {
  private Instrumentos instrumentos;
  public Orquesta() {
      instrumentos = new Instrumentos(3);
  }
  public boolean addInstrumento(Instrumento i) {
      return instrumentos.addInstrument(i);
  }
  public boolean removeInstrumento(Instrumento i) {
      return instrumentos.removeInstrument(i);
  }
  public Iterator<Instrumento> iterator() {
      return instrumentos.iterator();
  }
  public String tocar() {
      StringBuffer sb = new StringBuffer();
      for (Instrumento i: instrumentos)
        sb.append( i.tocarPartitura() );
      return sb.toString();
  }
  public String afinar(Instrumento i) {
    StringBuffer sb = new StringBuffer();
    return sb.append( i.afinar() ).toString();
  }
}
```

```java
public class Instrumentos
             implements Iterable<Instrumento> {
  /* Es lo mismo si se implementa internamente con
     una List o con un Map, pues esta clase oculta
     la implementación concreta de la colección de
     instrumentos */
}
```

```java
import java.util.Iterator;
import java.util.Map;
import java.util.List;
import java.util.LinkedHashMap;
import java.util.ArrayList;
import java.util.Arrays;

public class PruebaOrquesta {
  public void main() {
    Orquesta orquesta = new Orquesta();
    orquesta.addInstrumento(new Viento("trompeta"));
    orquesta.addInstrumento(new Cuerda("violín"));
    orquesta.addInstrumento(new Percusion("bombo"));
    for (Instrumento i: orquesta)
      System.out.println ( orquesta.afinar(i) );
    orquesta.tocar();
  }
}
```

Diagrama de clases



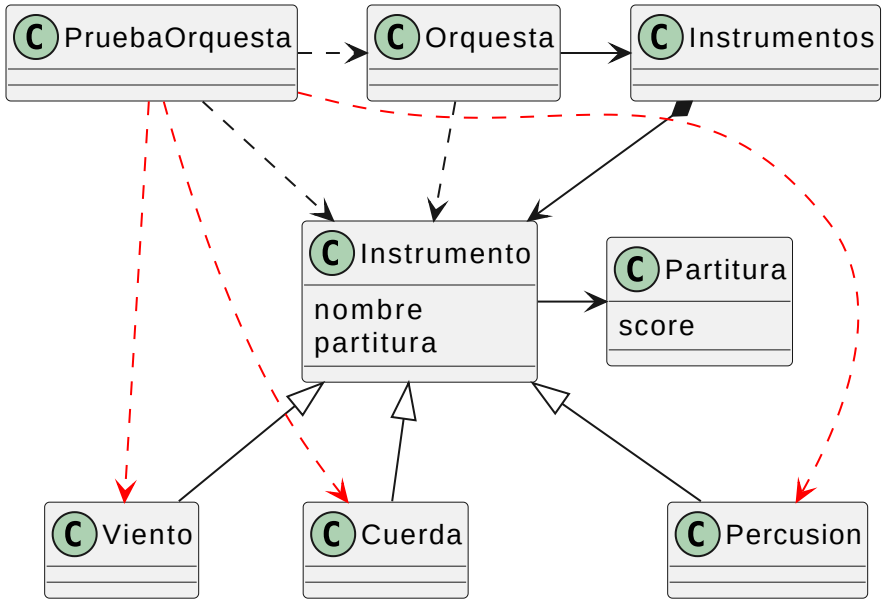
Dependencias de instrumento

```
public class PruebaOrquesta {
  public void main() {
    Orquesta orquesta = new Orquesta();
    orquesta.addInstrumento(new Viento("trompeta"));
    orquesta.addInstrumento(new Cuerda("violín"));
    orquesta.addInstrumento(new Percusion("bombo"));
    for (Instrumento i: orquesta)
        System.out.println ( orquesta.afinar(i) );
    orquesta.tocar();
  }
}
```

Los `new` de `PruebaOrquesta` siguen introduciendo dependencias de `PruebaOrquesta` con respecto a los tipos concretos de `Instrumento`.

Si quisiéramos probar la orquesta con otros instrumentos, tendríamos que modificar la clase cliente que utiliza la Orquesta.

Diagrama de clases - Dependencias



Por ejemplo, si programamos **casos de prueba unitaria** con *jUnit* (versión 3):

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class OrquestaTest extends junit.framework.TestCase {
  public void testTocar() {
      Orquesta orquesta = new Orquesta();
      orquesta.addInstrumento(new Viento("trompeta"));
      orquesta.addInstrumento(new Cuerda("guitarra"));
      orquesta.addInstrumento(new Percusion("tambor"));
      assertNotNull(orquesta.tocar());
      assertsEquals(orquesta.tocar(),"GD7CD7G");
  }
}
```

Problemas:

- Problema con la instanciación de instrumentos
- Cada vez que se prueba `Orquesta`, también se prueban las subclases de `Instrumento`.
- No se puede pedir a la orquesta que se comporte de otra forma (por ejemplo, un conjunto diferente de instrumentos)
- Tampoco se puede cambiar la partitura que queremos probar

Solución: inyección de dependencias

- Proporcionar a la `Orquesta` el conjunto de intrumentos de los que depende
- Proporcionar a cada `Instrumento` la partitura con la que debe ejecutar

Opciones:

Dependencia `Instrumento` `--→` `Partitura`:

- ¿Añadir un argumento `Partitura` a los constructores de las subclases de `Instrumento`?
- ¿Definir métodos `Instrumento::setPartitura(Partitura p)`?

Dependencia `Orquesta` `--→` `Instrumento`:

- ¿Añadir un argumento `Instrumento` al constructor de la `Orquesta`?
- ¿Definir métodos `Orquesta::addInstrumento(Instrumento i)`?
  - Esto ya lo estamos haciendo en `PruebaOrquesta` y `OrquestaTest`
- ¿Y por qué no un método `Orquesta::setPartitura(Partitura p)`?

¿Quién le añade los instrumentos a la orquesta?

¿Quién asigna la partitura? ¿A la orquesta o al instrumento?

Framework DI

La **inyección de dependencias** (DI) no suele hacerse de modo manual (programando una clase que lo haga), sino que se delega en una biblioteca especial (el framework DI) que lo hace, previa configuración.

El framework DI inyecta dependencias de forma universal, no de modo particular a un programa específico y a las clases que lo componen.

Algunos frameworks de inyección de dependencias

- Google guice
- Spring Framework
- Weld CDI
- Eclipse RCP

Inyección con Spring Framework

En un fichero de configuración `orquesta.xml` le indicamos los valores inyectables:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "..." ...>
<beans>
  <bean id="trompeta"
    class="Viento"/>
  <bean id="violin"
    class="Cuerda"/>
  <bean id="tambor"
    class="Percusion"/>
  <bean id="viola"
    class="Cuerda"/>
```

```xml
  <bean id="cuarteto"
    class="Orquesta">
    <property name="instrumento1">
      <ref bean="trompeta"/>
    </property>
    <property name="instrumento2">
      <ref bean="violin"/>
    </property>
    <property name="instrumento3">
      <ref bean="viola"/>
    </property>
    <property name="instrumento4">
      <ref bean="tambor"/>
    </property>
  </bean>
</beans>
```

La inyección de la dependencia concreta la hace el contenedor (*spring* en este ejemplo):

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

public class PruebaOrquesta {
  public static void main(String[] args) throws Exception {
    BeanFactory factory =
      new XmlBeanFactory(new FileInputStream("orquesta.xml"));
    Orquesta orquesta = (Orquesta)factory.getBean("cuarteto");
    for (Instrumento i: orquesta)
      orquesta.afinar(i);
    orquesta.tocar();
  }
}
```

Beans

Un *bean* es una clase/componente reutilizable en Java que tiene una interfaz bien definida, según una especificación estándar de Java, que permite a un contenedor gestionar su *ciclo de vida* (crearlos, cambiarles valores de sus propiedades, destruirlos, etc.)

Los *beans* son usados por muchos frameworks, entre otros Spring:

- Spring Bean
- Spring FactoryBean

Más info sobre Spring DI

Ejemplo: Logger

También se puede inyectar la dependencia en el constructor.

```
import java.util.logging.Logger;

public class MyClass {
  private final static Logger logger;
  public MyClass(Logger logger) {
      this.logger = logger;
      // write an info log message
      logger.info("This is a log message.")
  }
}
```

Un *contenedor* de dependencias en el framework debe responsabilizarse de crear las instancias de `Logger` e inyectarlas en su sitio (normalmente vía *reflexión* o *introspección*)

Anotaciones

Otra manera de inyectar dependencias

Anotaciones @ de Java en jUnit 4

```
import org.junit.*;
import static org.junit.Assert.*;

public class OrquestaTest { // no hace falta extends
  private Orquesta orquesta;
  @Before
  protected void setUp() {
      Orquesta orquesta = new Orquesta();
      orquesta.addInstrumento(new Viento("trompeta"));
      orquesta.addInstrumento(new Cuerda("guitarra"));
      orquesta.addInstrumento(new Percusion("tambor"));
  }
  @After
  protected void tearDown() {
      orquesta = null;
  }
  @Test
  public void testTocar() {
      assertNotNull(orquesta.tocar());
      assertsEquals(orquesta.tocar(),"GD7CD7G");
  }
}
```

¿Es necesario usar la inyección de dependencias para especificar las *partituras* con las que deben funcionar los instrumentos de la *orquesta*?

CASO PRÁCTICO: Implementación de comparadores (2)

Ejercicio: Identificador de BankAccount con inyección de dependencias

Supongamos que queremos obtener un listado ordenado por fecha de creación de todas las cuentas bancarias.

¿Cómo afecta este cambio a la versión de `BankAccount` ya implementada con JDK 1.5?

Resolvemos mediante inyección de dependencias...

Con herencia de interfaz y delegación

`BankAcccount.java`:

```
import java.util.*;
import java.io.*;
import java.time.*;
```

```
public final class BankAccount implements Comparable<BankAccount> {
  private final String id;
  private LocalDate creationDate;
  private Comparator comparator;

  public BankAccount(String number) {
    this.id = number;
    comparator = new BankAccountComparatorById();
  }
  public LocalDate getCreationDate() {
    return creationDate;
  }
  public void setCreationDate(LocalDate date) {
    this.creationDate = date;
  }
  public String getId() {
    return id;
  }
```

```
  public void setComparator(Comparator cmp) {
    comparator = cmp;
  }
  @Override
  public int compareTo(BankAccount other) {
    if (this == other)
      return 0;
    assert this.equals(other) : "compareTo inconsistent with equals.";
    return comparator.compare(this, other);
  }
  @Override
  public boolean equals(Object other) {
    if (this == other)
      return true;
    if (!(other instanceof BankAccount))
      return false;
    BankAccount that = (BankAccount) other;
    return this.id.equals(that.getId());
  }
  @Override
  public String toString() {
    return id.toString();
  }
}
```

`BankAcccountComparatorById.java`:

```
import java.util.Comparator;

class BankAccountComparatorById implements Comparator<BankAccount> {
    public int compare(BankAccount o1, BankAccount o2) {
        return o1.getId().compareTo(o2.getId());
    }
}
```

`BankAcccountComparatorByCreationDate.java`:

```
import java.util.Comparator;

class BankAccountComparatorByCreationDate implements Comparator<BankAccount> {
    public int compare(BankAccount o1, BankAccount o2) {
        return o1.getCreationDate().compareTo(o2.getCreationDate());
    }
}
```

Con inyección de dependencias

El motor de inyección de dependencias (por ejemplo, Spring) inyectaría la clase concreta `BankAccountComparatorBy...` de alguna de estas formas:

- Inyección a través del **constructor**: la clase inyectora suministra la dependencia a través del constructor de la clase dependiente.
- Inyección a través de **propiedades**: la clase inyectora suministra la dependenca a través de un método *setter* de la clase dependiente.
- Inyección a través de **métodos (API)**: la clase inyectora suministra la dependencia a través de una API determinada para la que está preparada (construida/configurada) la clase dependiente.

Creación de anotaciones

Ahora podría definirse una anotación del tipo `@comparator(BankAccountComparatorById.className)` o `@compareById` que inyecte a `BankAccount` una dependencia `BankAccountComparatorById` en `BankAccount.comparator`.

Inyección de dependencias con anotaciones

- Java: Ejemplo de cómo crear una anotación a medida en Java
- Typescript: las anotaciones se llaman **decorators** y son más sencillas de programar
- Python: No confundir con los decorators de Python, que son algo diferente