1

FRAMEWORK SPRING

1.1 INTRODUCCIÓN

Spring es un *framework* de desarrollo y contenedor de inversión de control *open source* para crear aplicaciones Java.

El creador de Spring es Rod Johnson, quien lo lanzó junto a la publicación de su libro *Expert One-on-One J2EE Design and Development* (Wrox Press, octubre 2002). En junio de 2003 vio la luz la primera versión del *framework* bajo la licencia Apache 2.0.

El primer gran lanzamiento fue la versión 1.0, que apareció en marzo de 2004. Al poco tiempo fue seguida por otros hitos en septiembre de 2004 y marzo de 2005. Los premios llegaron con la versión 1.2.6, cuando Spring obtuvo los reconocimientos *Jolt Awards* y *Jax Innovation Awards* en 2006. En ese mismo año fue lanzada la versión 2.0 del *framework*, posteriormente la versión 2.5 en noviembre de 2007, la 3.0 en diciembre de 2009, y dos años más tarde la 3.1. En enero de 2013 se anunció el inicio del desarrollo de la versión 4.0. Finalmente, en el momento de escribir este documento nos encontramos en la versión 4.2.0.

Una aplicación Java normalmente consiste en una serie de objetos que colaboran para conseguir el propósito de la aplicación. De ahí que los objetos de una aplicación tengan dependencia unos de los otros.

Aunque la plataforma Java proporciona una gran cantidad de funcionalidades de desarrollo de aplicaciones, tiene carencias en la manera de organizar las distintas partes de una aplicación en un todo, delegando esta tarea en arquitectos y desarrolladores. Los profesionales pueden apoyarse en los distintos patrones de

diseño existentes (por ejemplo *Factory*, *Abstract Factory*, *Builder*, *Decorator*, o *Service Locator*), sin embargo estos patrones no son más que simplemente buenas prácticas que uno debe implementar por sí mismo.

El componente de inversión de control del *framework* Spring se encarga de solucionar esta dificultad proporcionando una manera formalizada de componer y combinar los distintos componentes de una aplicación.

1.2 COMPONENTES DE UNA APLICACIÓN SPRING

El *framework* Spring comprende diversos módulos que proveen un amplio rango de servicios:

- **Contenedor de inversión de control**: permite la configuración de los componentes de la aplicación y la gestión de instancias de clases Java, que se lleva a cabo principalmente a través de la inyección de dependencias.
- **Programación orientada a aspectos**: facilita la implementación de código transversal.
- **Acceso a bases de datos**: permite interactuar con sistemas gestores de bases de datos con distintos paradigmas, como relacionales o NoSQL, utilizando tanto conectores con un nivel de abstracción menor como JDBC o superior como ORMs.
- **Gestión de transacciones**: unifica distintas APIs de gestión y coordina las transacciones para los objetos Java.
- **Modelo vista controlador** (MVC, *Model View Controller*): un *framework* basado en controladores, vistas (en JSP u otras tecnologías) y el modelo, además de en la extensión y personalización de aplicaciones web y servicios *web REST*.
- **Acceso remoto**: mediante tecnologías variadas como RMI, CORBA y protocolos basados en HTTP incluyendo servicios web (SOAP) facilita la importación y exportación del estilo RPC de objetos Java.
- **Convención sobre configuración**: el módulo Spring Roo fue introducido con el propósito de agilizar la puesta en marcha de proyectos en el *framework* Spring mediante un *shell* muy potente, donde prima la simplicidad sin renunciar en ningún momento a la flexibilidad.

- **Procesamiento por lotes**: un *framework* de procesamiento de tareas, con funcionalidades reutilizables como gestión de transacciones, estadísticas de procesamiento de trabajos, inicio de tareas, etc.
- **Autenticación y autorización**: herramientas para procesos de seguridad que abarcan un amplio abanico de estándares, protocolos, herramientas de seguridad y prácticas a través del subproyecto *Spring Security* (antiguamente *Acegi*).
- **Administración remota**: mediante JMX (*Java Management Extensions*), configuración de visibilidad y gestión de objetos Java para la configuración local o remota JMX.
- **Mensajes**: mediante JMS, registro configurable de objetos receptores de mensajes, para el consumo transparente; una mejora del envío de mensajes sobre las API JMS estándar.
- **Testing**: soporte de clases para desarrollo de unidades de prueba e integración.

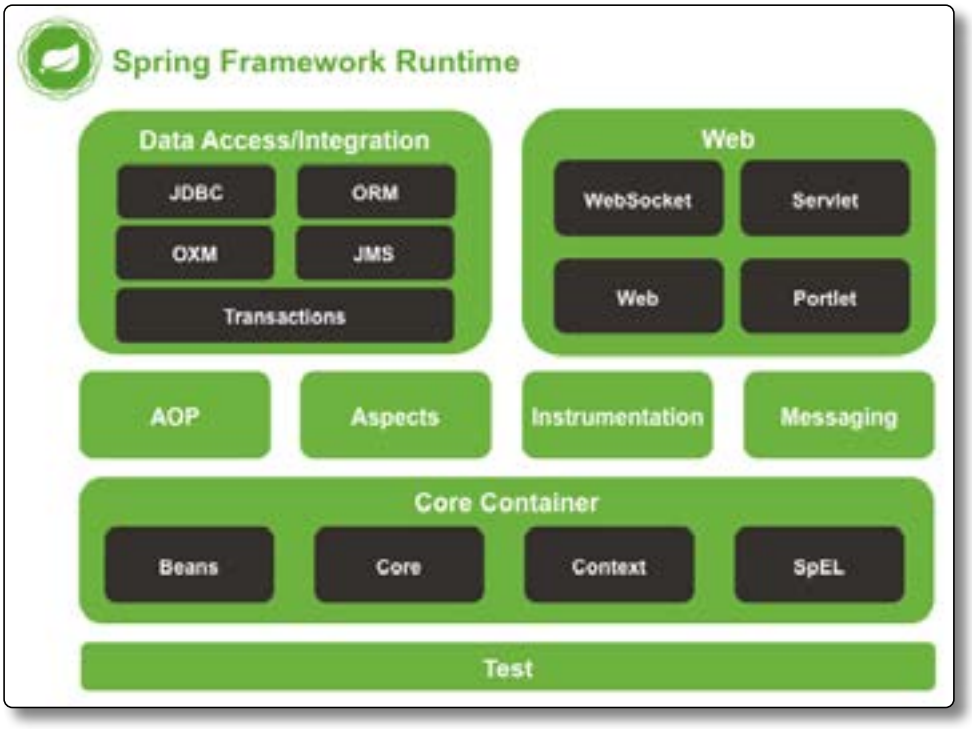


Figura 14.1. Módulos del entorno de ejecución de Spring

1.3 CARACTERÍSTICAS DE SPRING

Spring es el *framework* de desarrollo de aplicaciones Java más popular. Millones de desarrolladores en todo el mundo lo utilizan para crear aplicaciones de alto rendimiento, fáciles de probar y con código reutilizable.

Como ya se ha dicho, Spring es *open source*. Además, es un *framework* muy ligero, pesando la versión básica solo 2 MB.

Las características principales de Spring pueden ser utilizadas en el desarrollo de cualquier aplicación Java, pero hay extensiones para ir más allá en los desarrollos.

Los beneficios de utilizar Spring son:

- Permite a los desarrolladores el desarrollo de aplicaciones empresariales utilizando POJO. La ventaja de utilizar POJO es que no se necesita un contenedor de EJB como un servidor de aplicaciones, sino que existe la opción de utilizar simplemente un contenedor de *servlets* como puede ser Tomcat o cualquier otro producto comercial.

- Spring está organizado de manera modular. A pesar de que el número de clases y paquetes es elevado, solo tienes que preocuparte de los que necesitas, pudiendo ignorar el resto.

- Spring no trata de reinventar la rueda. En su lugar, utiliza algunas de las tecnologías ya existentes como ORM, *frameworks* de *logging*, etc.

- El testeo de aplicaciones es sencillo gracias a su mecanismo de inyección de dependencias.

- Consta de un *framework* MVC para el desarrollo de aplicaciones web.

- Proporciona una API para traducir excepciones específicas lanzadas por ejemplo por JDBC, Hibernate, etc., en excepciones más consistentes.

- Proporciona gestión de transacciones.

1.3.1 Entorno de desarrollo

En el momento de escribir esto la opción más cómoda para desarrollar proyectos Spring y poder incluir las dependencias de forma automática sería utilizando Eclipse J2EE Luna.

Esta versión de Eclipse trae soporte de serie para crear proyectos Maven. Para los proyectos Spring debemos incluir un *plugin*, el STS que está en *http://www.springsource.org/sts*. Con el *plugin* ya podemos crear proyectos de varios tipos. Nosotros utilizaremos aquel que se genera con la estructura que dictamina Maven y crea un *pom.xml* con las dependencias ya especificadas:

File > **New Spring Project** > **Simple Spring Maven**

Al crear este proyecto podrás distinguirlo de los anteriores (marcados con M de Maven) puesto que se marcan con la S de Spring. Ten en cuenta que todavía no estamos configurando un proyecto web sino un proyecto Spring ordinario para ejecución por consola.

Para la configuración de Spring se hará uso de un fichero *beans.xml* que, como todos los ficheros de configuración, debemos almacenar en *src/main/resources*. Para generarlo:

New > **Other** > **Spring** > **Spring Bean Configuration File**

Una vez que lo nombremos como *beans.xml* deberemos especificar los espacios de nombres que necesitamos incluir:

- *beans*: será utilizado en todos los proyectos puesto que nos permite manejar *beans* que representan los objetos.
- *aop*: para especificar aspectos.
- *context*: cuando utilicemos *autowiring*.
- *p*: propiedades abreviadas.
- *tx*: transacciones.
- Etc.

Ahora que tenemos el XML creado con la cabecera precisa procederemos a introducir los *beans* y las inyecciones.

1.3.2 Inyección de dependencias

Una de las mayores ventajas de las pruebas unitarias es la de garantizar la calidad del software desarrollado mediante métodos que permiten testear o probar una tarea concreta. No obstante, para realizar pruebas unitarias muchas veces surge la necesidad de falsear objetos en los tests unitarios (*mock objects*), lo cual exige desacoplamiento de clases y, por extensión, la DI (*Dependency Injection*).

Vamos a ver este caso a través de un sencillo ejemplo:

```
public class Driver {
  private Car vehicle = new Car();

  public Driver () {
  }

  public void drive () {
    vehicle.move();
  }
}
```

El *Driver* o conductor necesita obviamente un vehículo, y por tanto tiene un atributo que se instancia en la propia clase. La pega con la que nos encontramos es que el *Driver* se desplaza con un único coche y no puede usar otro vehículo. Además no podremos testear que realmente se llama al método *move()* del atributo *vehicle* ya que este es privado.

El objetivo final es que las clases estén cohesionadas, es decir, que colaboren entre ellas sin estar acopladas. Con el objeto de llegar a ese equilibrio aplicamos la inyección de dependencias para que las instancias de las que se depende se asignen desde fuera a la clase *Driver*.

Disponemos de dos vías simples para hacer la inyección a un atributo: o bien a través del método constructor o bien mediante los métodos *set*. Además en este caso no vamos a asignar una clase concreta sino una interfaz, lo que automáticamente habilita dos posibilidades: poder meter distintos tipos de vehículo (que deben implementar esa interfaz *Vehicle*) y simplificar el testeo del método desplazarse.

Esta sería la interfaz del vehículo:

```
public interface Vehicle {
  public void move();
}
```

Esta es una implementación de esa interfaz:

```
public class Car implements Vehicle {
  private int petrolTank;

  public void move() {
     if (petrolTank > 0)
      petrolTank--;
  }
}
```

Ahora cambiamos al *Driver* o conductor por lo siguiente:

```
public class Driver {
  private Vehicle vehicle;

  public Driver (Vehicle vehicle) {
    this.vehicle = vehicle;
  }

  public void drive () {
    vehicle.move();
  }
}
```

El vehículo, que será de cualquier tipo que implemente esa interfaz, se asignará desde fuera. Además no hacemos complejos los modelos sino que acabarán siendo POJO, es decir, objetos con métodos *getters/setters*.

Aplicando el *framework* Spring, mediante la configuración de un fichero XML podremos instanciar Vehículo (un objeto de la clase *Car*) e inyectarlo en una nueva instancia de *Driver*.

Además de conseguir desacoplar las clases, preparamos el terreno para facilitar el testeo del método *drive*, ya que ahora tenemos acceso al vehículo, que es la interfaz *Vehicle* y podemos falsearlo (o pasarle un *mock object*) en el momento de hacer *test*. Si lo que queremos es comprobar que realmente se llama al método *moverse()* haríamos algo así:

```
package org.sistema.spring.dibasicexample;

import static org.mockito.Mockito.*;
import org.junit.Test;
/**
* Testing drive method
```

```
*/
public class DriverTest {
@Test
public void driveCallsMove () {
   // Create a mock vehicle
   Vehiculo falseVehicle = mock(Vehicle.class);
   // Instance injection for the test
   Driver driver = new Driver(falseVehicle);
   driver.drive();
   // This is the actual TEST
   verify(falseVehicle, times(1)).move();
  }
}
```

Spring tiene como una de sus características más notables la inyección de dependencias. La inyección de dependencias permite instanciar objetos sin que tengamos que crear instancias dentro del código con *new*. Esto facilita en gran medida reducir el acoplamiento entre clases, haciendo que estas sean más sencillas (POJO) y también facilita las pruebas unitarias.

La creación de las instancias la lleva a cabo el *framework* Spring según la configuración que se le indica en un fichero XML. Veamos un ejemplo sencillo en el que tenemos una clase Persona y otra Dirección, con una relación 1:1 unidireccional en la que la Persona sabe su Dirección. Lo notable del ejemplo será que crearemos una Persona, le asociaremos una Dirección y la imprimiremos por pantalla sin que haya un solo *new* en nuestro código.

```
/**
 * Address class.
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Address {

  private long id;
  private String street;
  private String postCode;

  //Getters, setters & toString()
}
```

La clase Persona es la propietaria de la relación:

```
/**
 * Person class.
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Person {
   private long id;
   private String name;
   private Address address;

   //Getters, setters & toString()
}
```

Veamos ahora el fichero de Spring donde se crean las instancias de estas dos clases. Efectivamente es en el fichero *beans.xml* (aunque se puede cambiar) y se encuentra en *src/main/resources*, del que se había hablado en el apartado anterior.

```
<?xml version="1.0"?>
<beans xmlns="http://www.springframework.org/schema/
beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/
schema/beans
  http://www.springframework.org/schema/beans/spring-
beans-3.0.xsd ">

   <bean id="address" class="org.sistema.spring.
dependencyInjection.models.Address">
      <property name="id" value="1" />
      <property name="street" value="Avenida Barañáin"
/>
      <property name="postCode" value="31000" />
   </bean>

   <bean id="person" class="org.sistema.spring.
dependencyInjection.models.Person">
      <property name="id" value="1" />
      <property name="name" value="Eugenia" />
      <property name="address" ref="address" />
   </bean>
</beans>
```

Como se puede apreciar, se crea una instancia de cada clase, dando ya un valor a los atributos de cada una. Además, se indica que el atributo *address* va a apuntar al *bean address* que aparece justo encima de *Person*. Fíjate que el *id* de cada *bean* así como los *name* de cada propiedad deben coincidir con el nombre de cada atributo en las clases Java. En el ejemplo anterior los atributos se inyectan mediante los *set*.

También sería posible haciéndolo mediante los constructores de la clase:

```
  <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/
beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/
schema/beans http://www.springframework.org/schema/
beans/spring-beans-3.2.xsd">

  <bean id="address" class="com.sistema.spring.
Address">
    <constructor-arg name="id" value="1" />
    <constructor-arg name="street" value="Avenida
Barañáin" />
    <constructor-arg name="postCode" value="31000" />
  </bean>
  <bean id="person" class="com.sistema.spring.Person">
    <constructor-arg name="id" value="1" />
    <constructor-arg name="name" value="Eugenia" />
    <constructor-arg name="address" ref="address" />
  </bean>
</beans>
```

Comprobarás que vas a necesitar un constructor en cada clase que reciba todos esos parámetros y, adicionalmente, para que funcione, su constructor por defecto.

Veamos ahora la clase *Main*. Se limita a cargar el contexto de Spring a partir del fichero *beans.xml* y a obtener el *bean person* (*address* se inicializará en cadena). Es decir, se encargará de instanciar la *Person* y de inyectar e instanciar el resto. Por último la imprime:

```
/**
 * Main class.
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
```

```
 */
public class Main {

   private static ApplicationContext context;

   public static void main(String[] args) {
      context = new ClassPathXmlApplicationContext("beans.
xml");
      Person person = (Person) context.
getBean("person");
      System.out.print(person);
   }
}
```

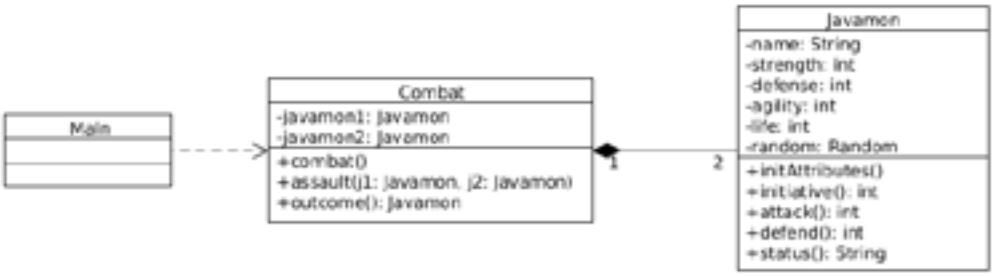
Y el resultado sería:

```
Person [id=1, name=Eugenia,
   address=Address [id=1, street=Avenida Barañáin,
postCode=31000]]
```

1.3.3 Autowiring

Conforme los proyectos se complican puede que gestionar el fichero XML donde se hilan las instancias se convierta en una tarea demencial. Una forma de paliar eso es utilizar el *autowiring*, lo cual nos permite mediante varios tipos de convenciones hacer que las instancias se asignen solas de forma automática.

Aquí vamos a ver cómo se podría hacer el *autowiring* a través del nombre de *bean* en el fichero XML. El proyecto *javamon* consiste en una clase de combate que carga dos instancias de la clase *Javamon* y las hace pelear. Todo ello se inicia desde una clase principal.



Esta sería la clase *Javamon*, que representa una especie de criatura con una serie de atributos (velocidad, fuerza) que ataca y se defiende:

```
package org.sistema.spring.autowiring.models;

import java.util.Random;

/**
 * Represents a Javamon creature
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Javamon {

   private String name;
   private int strength;
   private int defence;
   private int agility;
   private int life;
   private Random random = new Random();

   /**
    * constructor
    *
    * @param nombre
    */
   public Javamon(String nombre) {
      this.name = nombre;
      initAttributes();
   }

   /**
    * intis javamon attributes randomly
    */
   private void initAttributes() {
      int pointsToDeal = 18;

      // to what attrib we give the points in each loop
      int whatAttrib = 0;

      // points to give at most en each loop
      int max = 0;
```

```
    // first they are 0
    strength = defence = agility = 0;
    while (pointsToDeal > 0) {
      whatAttrib = random.nextInt(3);
      switch (whatAttrib) {
      case 0:
        strength++;
        break;
      case 1:
        defence++;
        break;
      case 2:
        agility++;
        break;
      default:
        break;
      }

      pointsToDeal--;
    }

    life = strength + defence + 6;
  }

  /**
   * an initiative roll
   *
   * @return
   */
  public int initiative() {
    return agility + random.nextInt(6);
  }

  /**
   * an attacking roll
   *
   * @return
   */
  public int attack() {
    return strength + random.nextInt(6);
  }

  /**
   * a defending roll
   *
```

```
     * @return
     */
    public int defend() {
      return ((agility + defence) / 2) + random.
nextInt(6);
    }

    /**
     * javamon status description
     *
     * @return
     */
    public String status() {
      return name + "(" + life + ")| S:" + strength + "|
D:" + defence
          + "| A:" + agility;
    }

    public String getName() {
      return name;
    }

    public void setName(String name) {
      this.name = name;
    }

    public int getStrength() {
      return strength;
    }

    public void setStrength(int strength) {
      this.strength = strength;
    }

    public int getDefence() {
      return defence;
    }

    public void setDefence(int defence) {
      this.defence = defence;
    }

    public int getAgility() {
      return agility;
    }
```

```
    public void setAgility(int agility) {
       this.agility = agility;
    }

    public int getLife() {
       return life;
    }

    public void setLife(int life) {
       this.life = life;
    }

    public Random getRandom() {
       return random;
    }

    public void setRandom(Random random) {
       this.random = random;
    }

}
```

La clase Combate es la que se encarga de crear dos instancias de *Javamon* y las hace combatir.

```
package org.sistema.spring.autowiring.models;

/**
 * Runs a combat between Javamon creatures
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Combat {

   private Javamon javamon1;
   private Javamon javamon2;
   private int assaults = 0;

   /**
    * Class constructor
    */
   public Combat() {
   }
```

```
   /**
    * this method performs combat
    */
   public void combat() {
     Javamon first, second;
     do {
       assaults++;

       // Depending on initiative roll
       // one attacks and the other defends
       if (javamon1.initiative() < javamon2.initiative()) {
         first = javamon2;
         second = javamon1;
       } else {
         first = javamon1;
         second = javamon2;
       }
       System.out.println(first.status() + " -> attacks -> "
            + second.status());
       assault(first, second);
       // while both are alive they keep on fighting
     } while (first.getLife() > 0 && second.getLife() > 0);

   }

   /**
    * represents one assault of the combar the first
parameter is the attacker
    *
    * @param jm1
    * @param jm2
    */
   private void assault(Javamon jm1, Javamon jm2) {
     // Assault damage will be one attack minus
defender defense roll
     int damage = (jm1.attack() - jm2.defend());
     // In case of damage we decrease defender life points
     if (damage > 0) {
       System.out.println(jm1.getName() + " -> makes " + damage
            + " damage to -> " + jm2.getName());
       jm2.setLife(jm2.getLife() - damage);
     } else {
       System.out.println(jm2.getName() + " stops the
attack!!");
     }

   }
```

```
    /**
     * outcome of the combat
     *
     * @return the winner javamon
     */
    public Javamon outcome() {
      System.out.println("Total " + assaults + "
assaults");
      if (javamon1.getLife() > 0) {
        return javamon1;
      } else {
        return javamon2;
      }
    }

    /**
     * @return the javamon1
     */
    public Javamon getJavamon1() {
      return javamon1;
    }

    /**
     * @param javamon1
     *       the javamon1 to set
     */
    public void setJavamon1(Javamon javamon1) {
      this.javamon1 = javamon1;
    }

    /**
     * @return the javamon2
     */
    public Javamon getJavamon2() {
      return javamon2;
    }

    /**
     * @param javamon2
     *       the javamon2 to set
     */
    public void setJavamon2(Javamon javamon2) {
      this.javamon2 = javamon2;
    }

}
```

La clase principal lo único que hace es sacar una instancia de Combate desde el contexto Spring y llama al método para poner dos *Javamon* en combate.

```
package org.sistema.spring.autowiring;

import org.sistema.spring.autowiring.models.Combat;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

/**
 * Main class.
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Main {

   public static void main(String[] args) {
      // public static void main (String args[]) {
      ApplicationContext context = new
ClassPathXmlApplicationContext(
            "beans.xml");
      Combat combat = (Combat) context.
getBean("combat");
      combat.combat();
      System.out.println("And the winner is: " + combat.
outcome().getName());
      System.out.println("Thanks for playing javamon");
   }

}
```

Es en el fichero de Spring donde se hace el *autowiring* por nombre. Creamos dos *beans* llamados estratégicamente *javamon1* y *javamon2* que deben coincidir con los nombres de los atributos (y por tanto los *set* y *get*) de la clase *Combat*. En la instancia de *Combat* debemos decir que hacemos *autowiring*, y Spring hilará el resto.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/
beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
   xsi:schemaLocation="http://www.springframework.org/
schema/beans
```

```
http://www.springframework.org/schema/beans/spring-
beans-3.2.xsd">

   <!-- We instantiate two javamons -->
   <bean id="javamon1" class="org.sistema.spring.
autowiring.models.Javamon">
      <constructor-arg value="Pikachu" />
   </bean>
   <bean id="javamon2" class="org.sistema.spring.
autowiring.models.Javamon">
      <constructor-arg value="Bulbasur" />
      <!-- Si el constructor tuviera más de 1 parámetro
se pondrían tantos
      elementos constructor-arg como parámetros tuviese
-->
   </bean>

   <!-- We create a combat autowiring previous javamon
by name - Combat class
      has two attributes called javamon1 and javamon2
With autowiring they will
      be set automagically -->
   <bean id="combat" class="org.sistema.spring.
autowiring.models.Combat"
      autowire="byName">
   </bean>
</beans>
```

Hay cuatro tipos de *autowiring*:

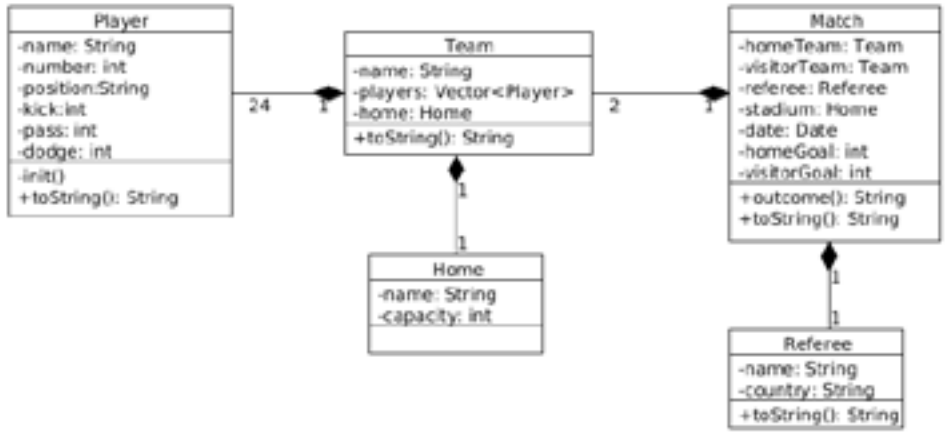
- **Por nombre (byName)**: es el utilizado en el ejemplo anterior.
- **Por tipo (byType)**: Spring tratará de encontrar una clase cuyo tipo se ajuste a lo que necesitamos.
- **Por constructor**: se buscan los *beans* que se ajusten al constructor. En este caso no fue posible su uso, ya que existían dos parámetros del mismo tipo.
- **autodetect**: en primera instancia se aplica *autowiring* por constructor y si no se consigue lo hace por tipo.

1.3.3.1 AUTOWIRING CON ANOTACIONES

Nos hemos ahorrado algo de trabajo pero seguimos teniendo que hacer la configuración vía XML. Si el único propósito para utilizar XML es crear instancias, entonces podríamos utilizar anotaciones en las propias clases. En concreto, vamos a ver la inyección de dependencias mediante anotaciones en dichas clases.

El proyecto consiste en una serie de clases que representan las clases de un partido de fútbol, con unas relaciones obvias.

- Clase *Player*: representa un jugador.
- Clase *Home*: representa un estadio.
- Clase *Team*: representa un equipo, contiene un *Hashtable* de jugadores y un estadio.
- Clase *Referee*: representa un árbitro.
- Clase *Match*: el partido, contiene un estadio, un árbitro y dos equipos.



Para hacer la inyección por anotaciones necesitaremos básicamente dos cosas:

- Se debe incluir la siguiente etiqueta *<qualifier name="..." />* en cada instancia desde el fichero XML. Será utilizada más adelante.
- Delante de la propiedad a inyectar o bien en su método *set* se deben indicar las siguientes anotaciones:

```
@Autowired
@Qualifier("el-name-del-xml")
```

Ahora veamos las distintas clases. Comenzaremos por la clase *Player*:

```
package org.sistema.spring.autowiring.annotations.
models;

import java.util.Random;

/**
 * Represents a football player
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Player {
   private String name;
   private int number;
   private String position;
   private int kick;
   private int pass;
   private int dodge; // dribbling
   private Random random = new Random();

   /**
    * default constructor
    */
   public Player() {
      init();
   }

   /**
    * Constructor using some fields
    *
    * @param name
    * @param number
    * @param position
    */
   public Player(String name, int number, String
position) {
      this.name = name;
      this.number = number;
      this.position = position;
   }
```

```
    /**
     * inits player playing atributes
     */
    private void init() {
        this.kick = random.nextInt(6) + 1;
        this.pass = random.nextInt(6) + 1;
        this.dodge = random.nextInt(6) + 1;
    }

    /*
     * (non-Javadoc)
     *
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() {
        return "Player [name=" + name + ", number=" +
number + ", position="
                + position + ", kick=" + kick + ", pass=" +
pass + ", dodge="
                + dodge + ", random=" + random + "]";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }
```

```
    public int getKick() {
       return kick;
    }

    public void setKick(int kick) {
       this.kick = kick;
    }

    public int getPass() {
       return pass;
    }

    public void setPass(int pass) {
       this.pass = pass;
    }

    public int getDodge() {
       return dodge;
    }

    public void setDodge(int dodge) {
       this.dodge = dodge;
    }

    public Random getRandom() {
       return random;
    }

    public void setRandom(Random random) {
       this.random = random;
    }

}
```

Clase *Home* (estadio):

```
package org.sistema.spring.autowiring.annotations.
models;

/**
 * Represents a football player
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
```

```
public class Home {
   private String name;
   private int capacity;

   /**
    * default constructor
    */
   public Home() {
   }

   /**
    * @param name
    * @param capacity
    */
   public Home(String name, int capacity) {
      this.name = name;
      this.capacity = capacity;
   }

   /*
    * (non-Javadoc)
    *
    * @see java.lang.Object#toString()
    */
   @Override
   public String toString() {
      return "Home [name=" + name + ", capacity=" +
capacity + "]";
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getCapacity() {
      return capacity;
   }

   public void setCapacity(int capacity) {
      this.capacity = capacity;
   }

}
```

Clase *Referee* (árbitro):

```
package org.sistema.spring.autowiring.annotations.
models;

/**
 * The referee of the match
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Referee {
   private String name;
   private String country;

   /**
    * default constructor
    */
   public Referee() {
   }

   /*
    * (non-Javadoc)
    *
    * @see java.lang.Object#toString()
    */
   @Override
   public String toString() {
      return "Referee [" + (name != null ? "name=" +
name + ", " : "")
            + (country != null ? "country=" + country + ",
" : "")
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getCountry() {
      return country;
   }
```

```
   public void setCountry(String country) {
      this.country = country;
   }
}
```

Clase *Team* (equipo):

```
package org.sistema.spring.autowiring.annotations.
models;

import java.util.Hashtable;
import org.springframework.beans.factory.annotation.
Autowired;
import org.springframework.beans.factory.annotation.
Qualifier;

/**
 * Represents a football team
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Team {
   private String name;
   private Hashtable<Integer, Player> players;
   @Autowired
   @Qualifier("sadar")
   private Home homeStadium;

   /**
    * default constructor
    */
   public Team() {
   }

   /**
    * @param name
    * @param players
    * @param homeStadium
    */
   public Team(String name, Hashtable<Integer, Player>
players,
         Home homeStadium) {
      super();
      this.name = name;
```

```
        this.players = players;
        this.homeStadium = homeStadium;
    }

    /*
     * (non-Javadoc)
     *
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() {
        return "Team [name=" + name + ", players=" +
players.toString()
                + ",homeStadium=" + homeStadium + "]";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Hashtable<Integer, Player> getPlayers() {
        return players;
    }

    public void setPlayers(Hashtable<Integer, Player>
players) {
        this.players = players;
    }

    public Home getHomeStadium() {
        return homeStadium;
    }

    public void setHomeStadium(Home homeStadium) {
        this.homeStadium = homeStadium;
    }

}
```

Clase *Match* (partido):

```
package org.sistema.spring.autowiring.annotations.
models;

import java.util.Date;
import org.springframework.beans.factory.annotation.
Autowired;
import org.springframework.beans.factory.annotation.
Qualifier;

/**
 * Represents a football match between two teams
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Match {
   @Autowired
   @Qualifier("osasuna")
   private Team homeTeam;
   @Autowired
   @Qualifier("erreala")
   private Team visitorTeam;
   @Autowired
   @Qualifier("undiano")
   private Referee referee;
   @Autowired
   private Home stadium; // There is only one Home in
xml file. Spring
   // will inject that.
   private Date date;
   private int homeGoal;
   private int visitorGoal;

   /**
    * default constructor
    */
   public Match() {
   }

   /**
    * gives result of the match
    *
    * @return
```

```
     */
    public String outcome() {
      return homeTeam.getName() + " " + homeGoal + " - "
          + visitorTeam.getName() + " " + visitorGoal;
    }

    /*
     * (non-Javadoc)
     *
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() {
      return "Match [homeTeam=" + homeTeam.toString() +
"\n, visitorTeam="
          + visitorTeam.toString()
          + "\n, referee=" + referee.toString()
          + "\n, stadium=" + stadium.toString() + "\n,
date=" + date
          + ", homeGoal=" + homeGoal + ", visitorGoal="
+ visitorGoal
          + "]";
    }

    public Team getHomeTeam() {
      return homeTeam;
    }

    public void setHomeTeam(Team homeTeam) {
      this.homeTeam = homeTeam;
    }

    public Team getVisitorTeam() {
      return visitorTeam;
    }

    public void setVisitorTeam(Team visitorTeam) {
      this.visitorTeam = visitorTeam;
    }

    public Referee getReferee() {
      return referee;
    }
```

```
    public void setReferee(Referee referee) {
        this.referee = referee;
    }

    public Home getStadium() {
        return stadium;
    }

    public void setStadium(Home stadium) {
        this.stadium = stadium;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public int getHomeGoal() {
        return homeGoal;
    }

    public void setHomeGoal(int homeGoal) {
        this.homeGoal = homeGoal;
    }

    public int getVisitorGoal() {
        return visitorGoal;
    }

    public void setVisitorGoal(int visitorGoal) {
        this.visitorGoal = visitorGoal;
    }

}
```

Y esta es la clase principal que lanza la ejecución:

```
package org.sistema.spring.autowiring.annotations;

import org.sistema.spring.autowiring.annotations.
models.Match;
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;
/**
 * Main class.
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Main {

  public static void main(String[] args) {

    ApplicationContext context = new
ClassPathXmlApplicationContext(
         "football.xml");
    Match match = (Match) context.getBean("match");
    System.out.println("Match data: " + match.
toString());
  }

}
```

Por último, necesitamos crear el fichero XML de Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/
beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/
context"
  xsi:schemaLocation="http://www.springframework.org/
schema/beans
http://www.springframework.org/schema/beans/spring-
beans-3.2.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-
context-3.0.xsd">
  <!-- To enable autowiring through annotations -->
  <!-- Don't forget to add context-related xsd lines
above -->
  <context:annotation-config />
  <bean id="match" class="org.sistema.spring.
autowiring.annotations.models.Match">
  </bean>
```

```
  <bean id="team1" class="org.sistema.spring.
autowiring.annotations.models.Team">
     <qualifier value="osasuna" /> <!-- Used in
autowiring with qualifier -->
     <property name="name" value="Osasuna" />
     <!-- This is the way to wire a Hashtable -->
     <property name="players">
       <map>
         <entry key="24" value-ref="player1" />
         <entry key="1" value-ref="player2" />
       </map>
     </property>
  </bean>
  <bean id="team2" class="org.sistema.spring.
autowiring.annotations.models.Team">
     <qualifier value="erreala" />
     <property name="homeStadium" ref="home1" />
     <property name="name" value="Real Sociedad" />
     <property name="players">
       <map>
         <entry key="4" value-ref="player3" />
         <entry key="17" value-ref="player4" />
       </map>
     </property>
  </bean>
  <bean id="home1" class="org.sistema.spring.
autowiring.annotations.models.Home">
     <qualifier value="sadar" />
     <property name="name" value="Sadar" />
  </bean>
  <bean id="referee"
     class="org.sistema.spring.autowiring.annotations.
models.Referee">
     <qualifier value="undiano" />
     <property name="name" value="Undiano Mallenco" />
  </bean>
<bean id="referee2"
     class="org.sistema.spring.autowiring.annotations.
models.Referee">
     <qualifier value="mejuto" />
     <property name="name" value="Mejuto González" />
        <!--if the referee of the match is required to
be changed in the future,
     changing the qualifier attribute of the Match class
from 'undiano'
        to 'mejuto' will be enough -->
```

```
<bean id="player1" class="org.sistema.spring.
autowiring.annotations.models.Player">
     <property name="name" value="Ilarra" />
     <property name="number" value="24" />
     <property name="position" value="Midfielder" />
  </bean>
  <bean id="player2" class="org.sistema.spring.
autowiring.annotations.models.Player">
     <property name="name" value="Casillas" />
     <property name="number" value="1" />
     <property name="position" value="Keeper" />
  </bean>
  <bean id="player3" class="org.sistema.spring.
autowiring.annotations.models.Player">
     <property name="name" value="Pepe" />
     <property name="number" value="4" />
     <property name="position" value="Defender" />
  </bean>
  <bean id="player4"
     class="org.sistema.spring.autowiring.annotations.
models.Player">
     <property name="name" value="Azpilicueta" />
     <property name="number" value="17" />
     <property name="position" value="Fullback" />
  </bean>
</beans>
```

Si ejecutamos el programa vemos que todo funciona correctamente:

```
Match data: Match [homeTeam=Team [name=Osasuna,
players={24=Player [name=Ilarra, number=24,
position=Midfielder, kick=6, pass=3, dodge=2,
random=java.util.Random@ac44e3], 1=Player
[name=Casillas, number=1, position=Keeper, kick=3,
pass=6, dodge=2, random=java.util.Random@1dea2d0]},home
Stadium=Home [name=Sadar, capacity=0]]
, visitorTeam=Team [name=Real Sociedad,
players={17=Player [name=Azpilicueta, number=17,
position=Fullback, kick=5, pass=6, dodge=2,
random=java.util.Random@1932672], 4=Player [name=Pepe,
number=4, position=Defender, kick=1, pass=6, dodge=6,
random=java.util.Random@1c11fcb]},homeStadium=Home
[name=Sadar, capacity=0]]
, referee=Referee [name=Undiano Mallenco, ]
, stadium=Home [name=Sadar, capacity=0]
, date=null, homeGoal=0, visitorGoal=0]
```

1.3.4 SpEL

SpEL o *Spring Expression Language* es un lenguaje introducido desde la versión 3 de Spring que nos permite asignar valores complejos calculados en tiempo de ejecución.

SpEL nos permite por ejemplo hacer operaciones aritméticas, condicionales y hacer referencia a valores de otros *beans* existentes. Vamos a ver los distintos tipos de expresiones que podemos meter, desde las más simples a las más complejas.

1.3.4.1 LITERALES

La expresión más simple de SpEL es un valor dentro de *#{}*, por ejemplo:

```
<property name="speed" value="#{140}" />
<property name="euro" value="#{166.386}" />
```

Que también puede ser compuesto, dentro de una cadena:

```
<property name="description" value="Your age is #{84}" />
```

Pueden ser cadenas:

```
<property name="name" value="#{'Gandalf'}" />
```

O *booleanos*:

```
<property name="examPassed" value="#{false}" />
```

1.3.4.2 REFERENCIAS

En los ficheros también podemos asignar referencias para asignar una instancia entera a una propiedad. Pero si lo que nos interesa es una propiedad de un *bean*:

```
<property name="speed" value="#{'javamon1.speed'}" />
```

Spring llevará a cabo la operación *getSpeed()*.

También podemos invocar el método de un *bean* para sacar un valor.

```
<property name="strength" value="#{dice1.roll()}" />
```

1.3.4.3 TIPOS

Mediante el operador *T()* tenemos acceso a atributos y métodos estáticos de las clases, algo que puede ser realmente útil:

```
<property name="strength" value="#{T(java.lang.Math).PI}" />
<property name="strength" value="#{T(java.lang.Math).
random()}" />
```

También se puede hacer de esta otra forma usando una especie de *pseudojava* y operaciones aritméticas como se ve en el ejemplo anterior:

```
#{new java.util.Random().nextInt(6) + 1}
```

1.3.4.4 OPERACIONES

Podemos meter expresiones más o menos similares a las operaciones básicas que tenemos en el lenguaje Java, con alguna variante:

Aritméticas

+, -, *, /, %, ^

A diferencia de Java, en SpEL tenemos el operador de potencia ^. Un ejemplo simple:

```
<property name="beast" value=#{600 + 66} />
```

Cálculo de la circunferencia dado un radio 40:

```
<property name="circumference" value="#{2 * T(java.
lang.Math).PI * 40}" />
```

Y el área:

```
<property name="area" value="#{T(java.lang.Math).PI *
40 ^2}" />
```

Comparación

<, >, ==, <=, >=, lt, gt, eq, le, ge

Las tenemos en dos formatos, al estilo Java o con expresiones *lt*, *gt*… las cuales tienen el mismo efecto. Nos pueden servir para establecer valores *booleanos* o aplicarlas en expresiones más complejas.

```
<property name="isAllowed" value="#{customer1.age > 17}" />
```

Y esto sería lo mismo:

```
<property name="isAllowed" value="#{customer1.age gt 17}" />
```

Lógicas

and, *or*, *not*, *!*

Un ejemplo sencillo:

```
<property name="isHero" value="#{player1.speed > 100
and player1.level >19}" />
```

Condicionales

?: operador ternario (o *Elvis operator*)

Podemos aplicar esta abreviatura de *if-else* para, por ejemplo, establecer valores:

```
<property name="weapon" value="#{player1.
isDwarf()?axe:sword}" />
```

Expresiones regulares

También podemos usar un operador llamado *match* y expresiones regulares con la sintaxis habitual en la mayoría de lenguajes:

```
<property name="isElvenName" value="#{player1.name
matches '[a-z]+as$'}" />
```

Colecciones

Y por último, si tenemos alguna *bean* que contenga una colección podemos acceder a esos elementos como si se tratara de un *array*, y lo mismo podríamos hacer si fuera un *Hash* o una estructura de ese tipo.

```
<property name="badBull" value="#{bulls[2]}">
```

1.3.5 Spring JDBC

Al trabajar con bases de datos utilizando JDBC, es común tener que escribir código repetitivo para manejar excepciones, abrir y cerrar la conexión con la base de datos, etc. Spring JDBC se ocupa de todos esos detalles por nosotros, con lo que nos ahorramos ese código *boilerplate* tan pesado que se requiere cuando se hace a mano.

Por tanto, lo que debemos hacer nosotros es definir los parámetros de la conexión y especificar la sentencia SQL a ejecutar. A continuación veremos un ejemplo de cómo se suele hacer esto en Spring.

Comenzaremos creando una base de datos *jdbc_test* con la siguiente tabla dentro de MySQL:

```
CREATE DATABASE jdbc_test;
use jdbc_test;
CREATE TABLE Client (
  ID INT NOT NULL AUTO_INCREMENT,
  NAME VARCHAR(20) NOT NULL,
  COUNTRY VARCHAR(15) NOT NULL,
  PHONE VARCHAR(9) NOT NULL,
  AGE INT NOT NULL,
  PRIMARY KEY(ID)
);
```

A continuación creamos un proyecto vacío en Eclipse.

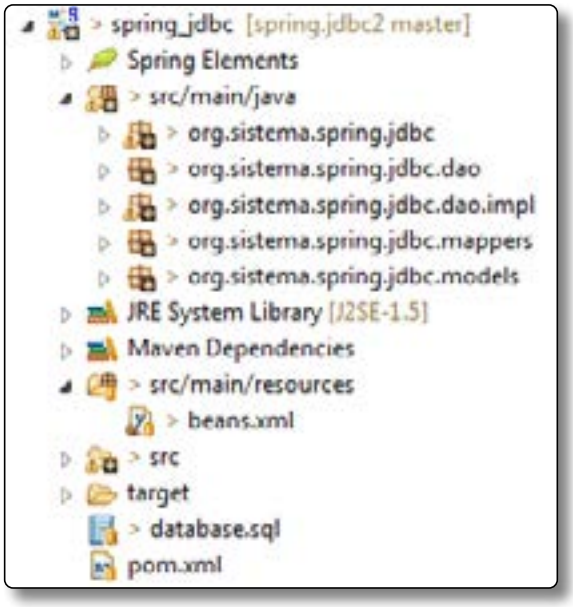


Figura 14.2. Estructura de un proyecto Spring

Ahora comenzaremos a añadir las dependencias. En principio, para este ejemplo necesitaremos incluir los módulos de *context*, *jdbc* y *transaction*. Añadimos

también la referencia al conector de *mysql* al igual que en los proyectos realizados en la parte de Hibernate.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sistema.spring</groupId>
  <artifactId>spring.jdbc</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>spring.jdbc</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.
build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>4.1.3.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>4.1.3.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>4.1.3.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.10</version>
    </dependency>
  </dependencies>
</project>
```

A continuación creamos la clase *Client* de nuestro modelo:

```
package org.sistema.spring.jdbc.models;
/**
 * Client representation
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Client {

   private Integer id;
   private String name;
   private String country;
   private String phone;
   private Integer age;

   public void setPhone(String phone) {
      this.phone = phone;
   }

   public String getPhone() {
      return phone;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setId(Integer id) {
      this.id = id;
   }

   public Integer getId() {
      return id;
   }

   public String getCountry() {
      return country;
   }
```

```
    public void setCountry(String country) {
        this.country = country;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

Para la capa de persistencia utilizaremos el patrón DAO igual que en la parte de Hibernate. Una práctica habitual es definir una interfaz por cada DAO, ya que esto es requerido para poder utilizar inyección de dependencias.

```
package org.sistema.spring.jdbc.dao;

import java.util.List;

import javax.sql.DataSource;

import org.sistema.spring.jdbc.models.Client;
/**
 * DAO class for Client entity
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public interface ClientDAO {
    /**
     * This is the method to be used to initialize
database resources ie.
     * connection.
     */
    public void setDataSource(DataSource ds);

    /**
     * This is the method to be used to create a record
in the Client table.
     */
    public void insert(String name, String country,
String phone, Integer age);
```

```
    /**
     * This is the method to be used to list down a
record from the Client
     * table corresponding to a passed client id.
     */
    public Client selectById(Integer id);

    /**
     * This is the method to be used to list down all the
records from the
     * Client table.
     */
    public List<Client> selectAll();

    /**
     * This is the method to be used to delete a record
from the Client table
     * corresponding to a passed client id.
     */
    public void delete(Integer id);

    /**
     * This is the method to be used to update a record
into the Client table.
     */
    public void update(Integer id, String name);
}
```

Ahora crearemos una clase que será la encargada de hacer el mapeo de cada tupla o registro que proviene de la base de datos a objetos de la clase *Client*. Este tipo de clases se suelen conocer como mapeadores.

```
package org.sistema.spring.jdbc.mappers;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;
import org.sistema.spring.jdbc.models.Client;
/**
 * Relational-object mapper for Client
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
```

```
public class ClientMapper implements RowMapper<Client>
{
  public Client mapRow(ResultSet rs, int rowNum) throws
SQLException {
     Client client = new Client();
     client.setId(rs.getInt("id"));
     client.setName(rs.getString("name"));
     client.setCountry(rs.getString("country"));
     client.setPhone(rs.getString("phone"));
     client.setAge(rs.getInt("age"));
     return client;
  }
}
```

Seguidamente veremos la implementación de la interfaz *ClientDAO* mediante JDBC. Esta manera de organizar los DAO facilitará en el futuro añadir una nueva implementación de este DAO mediante otro sistema de persistencia, como puede ser Hibernate:

```
package org.sistema.spring.jdbc.dao.impl;

import java.util.List;
import javax.sql.DataSource;
import org.sistema.spring.jdbc.dao.ClientDAO;
import org.sistema.spring.jdbc.mappers.ClientMapper;
import org.sistema.spring.jdbc.models.Client;
import org.springframework.jdbc.core.JdbcTemplate;

/**
 * JDBC Implementation of ClientDAO
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class JdbcClientDAO implements ClientDAO {
  private DataSource dataSource;
  private JdbcTemplate jdbcTemplateObject;

  public void setDataSource(DataSource dataSource) {
     this.dataSource = dataSource;
     this.jdbcTemplateObject = new JdbcTemplate(dataSource);
  }
```

```
  public void insert(String name, String country,
String phone, Integer age) {
     String SQL = "insert into Client (name, country,
phone, age) values (?, ?, ?, ?)";

     jdbcTemplateObject.update(SQL, name, country,
phone, age);
     System.out.println("Created Record Name: " + name
+ " Country: "
          + country + " Phone: " + phone + " Age: " +
age);
  }

  public Client selectById(Integer id) {
     String SQL = "select * from Client where id = ?";
     Client client = jdbcTemplateObject.
queryForObject(SQL,
          new Object[] { id }, new ClientMapper());
     return client;
  }

  public List<Client> selectAll() {
     String SQL = "select * from Client";
     List<Client> clients = jdbcTemplateObject.
query(SQL,
          new ClientMapper());
     return clients;
  }

  public void delete(Integer id) {
     String SQL = "delete from Client where id = ?";
     jdbcTemplateObject.update(SQL, id);
     System.out.println("Deleted Record with ID = " +
id);
  }

  public void update(Integer id, String name) {
     String SQL = "update Client set name = ? where id
= ?";
     jdbcTemplateObject.update(SQL, name, id);
     System.out.println("Updated Record with ID = " +
id);
  }
}
```

Por último, creamos una clase *Main* para probar todo el código anterior:

```
package org.sistema.spring.jdbc;

import java.util.List;

import org.sistema.spring.jdbc.dao.impl.JdbcClientDAO;
import org.sistema.spring.jdbc.models.Client;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

/**
 * Main class.
 *
 * @author Eugenia Pérez Martínez
 * @email eugenia_perez@cuatrovientos.org
 */
public class Main {

  public static void main(String[] args) {
    ApplicationContext context = new
ClassPathXmlApplicationContext(
        "beans.xml");
    JdbcClientDAO clientDAO = (JdbcClientDAO) context
        .getBean("clientJDBCTemplate");

    System.out.println("Creating clients
.....................");
    clientDAO.insert("César", "Spain", "676123456", 32);
    clientDAO.insert("John", "UK", "098765433", 45);
    clientDAO.insert("Mauro", "Italy", "11223344", 15);
    clientDAO.insert("Eugenia", "Spain", "985123456", 30);
    System.out.println();

    selectAllClients(clientDAO);

    System.out.println("Updating Client with ID = 3
............");
    clientDAO.update(3, "Peter");
    System.out.println();

    System.out.println("Listing Client with ID = 3
............");
    Client client = clientDAO.selectById(3);
```

```
        System.out.print("ID : " + client.getId());
        System.out.println(", Name : " + client.getName());
        System.out.println();

        System.out.println("Deleting Client with ID=1
..............");
        clientDAO.delete(1);
        selectAllClients(clientDAO);
    }

    private static void selectAllClients(JdbcClientDAO
clientDAO) {
        System.out.println("Listing clients
.....................");
        List<Client> clients = clientDAO.selectAll();
        for (Client record : clients) {
            System.out.print("ID : " + record.getId());
            System.out.print(", Name : " + record.
getName());
            System.out.print(", Country : " + record.
getCountry());
            System.out.print(", Phone : " + record.
getPhone());
            System.out.println(", Age : " + record.
getAge());
        }
        System.out.println();
    }
}
```

Antes de lanzar la aplicación, debemos crear el archivo mediante el cual indicamos los parámetros de configuración que Spring necesita. Este es el archivo *beans.xml*, que creamos en el directorio *src/main/resources*.

```
<?xml version="1.0"?>
<beans xmlns="http://www.springframework.org/schema/
beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/
schema/beans
    http://www.springframework.org/schema/beans/spring-
beans-3.0.xsd ">
    <!-- Initialization for data source -->
    <bean id="dataSource"
```

```
     class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
     <property name="driverClassName" value="com.mysql.
jdbc.Driver" />
     <property name="url" value="jdbc:mysql://
localhost:3306/jdbc_test" />
     <property name="username" value="root" />
     <property name="password" value="" />
  </bean>

  <!-- Definition for clientJDBCTemplate bean -->
  <bean id="clientJDBCTemplate" class="org.sistema.
spring.jdbc.dao.impl.JdbcClientDAO">
     <property name="dataSource" ref="dataSource" />
  </bean>
</beans>
```

Como se puede ver, indicamos los parámetros de conexión a la base de datos, pero también creamos una instancia de la implementación del *ClientDAO* con JDBC. Esta luego es cargada desde la clase *Main* de la siguiente forma (en lugar de instanciándola con *new*):

```
JdbcClientDAO clientJDBCTemplate = (JdbcClientDAO)
context
          .getBean("clientJDBCTemplate");
```

El resultado final debe ser:

```
Creating clients .....................
Created Record Name: César Country: Spain Phone:
676123456 Age: 32
Created Record Name: John Country: UK Phone: 098765433
Age: 45
Created Record Name: Mauro Country: Italy Phone:
11223344 Age: 15
Created Record Name: Eugenia Country: Spain Phone:
985123456 Age: 30

Listing clients ......................
ID : 1, Name : César, Country : Spain, Phone :
676123456, Age : 32
ID : 2, Name : John, Country : UK, Phone : 098765433,
Age : 45
ID : 3, Name : Mauro, Country : Italy, Phone :
11223344, Age : 15
```

```
ID : 4, Name : Eugenia, Country : Spain, Phone :
985123456, Age : 30

Updating Client with ID = 3 ............
Updated Record with ID = 3

Listing Client with ID = 3 ............
ID : 3, Name : Peter

Deleting Client with ID=1 ..............
Deleted Record with ID = 1

Listing clients ......................
ID : 2, Name : John, Country : UK, Phone : 098765433,
Age : 45
ID : 3, Name : Peter, Country : Italy, Phone :
11223344, Age : 15
ID : 4, Name : Eugenia, Country : Spain, Phone :
985123456, Age : 30
```