Guía Técnica para las Prácticas de Diseño de Sistemas Software

Daniel Molina Cabrera <daniel.molina@uca.es>

Curso 2011/2012

Contents

1 Descripción del documento

Este documento se plantea como una guía técnica para la realización de la práctica del documento. La idea de este documento es el de ofrecer una guía que permita a los alumnos/as de Diseño de Sistemas Software, para reducir los problemas técnicos, y que puedan centrarse en la resolución de las prácticas.

La idea del documento es de orientar a los/as alumnos/as, si algo no queda suficientemente claro, sólo hay que comunicarlo, y estaré encantado de resolver las dudas.

2 Lenguaje Java

La práctica se realizará en el lenguaje de Programación Java. Se ha elegido este lenguaje ya que es un lenguaje muy extendido (el más usado en los servidores) y en distinto tipo de sistemas (por ejemplo, Android se programa usando dicha sintáxis).

Otro punto a favor es su cercanía al lenguaje (C++) ya explicado en asignaturas anteriores, y a la existencia de excelentes IDEs, tanto de pago como gratuitos y libres, que simplifican mucho el desarrollo.

Tal y como se describe en el apartado de IDEs, se propone usar un IDE, para facilitar el trabajo. Además, se debe de crear el proyecto con una estructura estándar, independiente del IDE, denominado proyecto Maven (descrito en el apartado de Maven).

2.1 Aprendizaje de Java

A pesar de que pueda parecer otra cosa, en su sintaxis el lenguaje Java es muy parecido al C++. Se recomienda leer el documento Introducción al Java para programadores C++.

2.2 Librerías Java

Java, a diferencia de C++, presenta una completa librería de clases que ofrecen mucha funcionalidad. Para la práctica que nos ocupa, las clases que necesitaremos son muy pocas, y nos centraremos en las siguientes:

2.2.1 Clases contenedoras

El manejo de clases contenedoras está ofrecido por C++ mediante la increíblemente útil STL, sin embargo, presenta problemas en su uso, debido a dos grandes problemas: El confuso uso de los templates, y la poca utilidad de los mensajes de error.

La librería Java, por su parte, contiene en el paquete estándar **java.util.** un conjunto de contenedoras más fácil de utilizar.

Java ofrece las siguientes clases muy útiles: **List** (listas) implementado como *ArrayList* (*arrays* en C++), *LinkedList* (Listas enlazadas); **Map**, implementado como *HashMap* (*tabla hash*); o **Set** (colecciones sin repetición).

En mi experiencia, se suele usar bastante las listas (*List*) pero no lo suficiente las contenedoras *Map*, que en ciertos casos son mucho más apropiados y permiten simplificar bastante el código.

2.2.2 Librería de Fechas

Es ampliamente reconocido el mal manejo de las fechas por parte de la librería estándar de Java, por tener un API poco intuitiva y cómoda.

El hecho de que una funcionalidad esté en un API estándar no impide que pueda utilizarse otra librería. En ese caso, recomiendo el uso de la librería Joda-Time, que posiblemente acabe reemplazando la librería estándar.

En el apartado añadiendo librerías se introduce cómo instalar las librerías. en una sesión de prácticas se hará un ejemplo práctico.

2.2.3 Librerías de Validación

Es importante siempre validar los parámetros introducidos por el usuario, y especialmente cuando haya que realizar conversiones, como por ejemplo, una fecha en formato de cadena.

Para ello es muy útil instalar la librería **commons** de la fundación Apache. En dicha librería, existe una clase, denominada Validate, que permite métodos que facilitan la comprobación de parámetros (devuelve una excepción si no se cumple), la idea es similar al uso de *assert* en C/C++.

2.3 Implementación de la funcionalidad

Para poder ofrecer libertad en el diseño, cada grupo debe de elegir la estructura de clases que crea más conveniente. Para poder probar el código de forma homogénea, se ofrecerá unos interfaces. Dentro del código deberá de existir clases que implementen dichos interfaces y que delegen las funcionalidades pedidas a las clases que implementen dichas funcionalidades. Se ofrecerá asímismo pruebas automáticas que comprueben la funcionalidad utilizando dicho interfaz.

3 IDE

Una gran ventaja es que existen para java entornos de desarrollo (*IDE*) que permiten trabajar de forma mucho más cómoda. entre los distintos entornos, destaco dos que son gratuitos y libres. ambos son multiplataforma, existen para linux y windows (gracias a la portabilidad de java).

Los entornos que recomiendo son: el Eclipse, que es el IDE gratuito más popular, y el Netbeans, IDE oficial de sun/oracle, y el Eclipse. Ambos están fácilmente disponibles en la web. A su vez, en el campus virtual existe una transparencia sobre el entorno de desarrollo Eclipse, *Introduccion_ eclipse.pdf*, introduciéndolo.

3.1 Netbeans y Eclipse

Ambos entornos son muy completos, y la elección de uno u otro depende en gran parte de preferencias personales. Si tuviese que resumirlo, en mi opinión se distinguen en que el Eclipse posee una estructura más modular (existen módulos para casi todo), y el Netbeans es más compacto. Eso presenta ventajas y desventajas.

Gracias al componente *maven*, que describimos un poco a continuación, la estructura de los proyectos creados, independientemente del entorno utilizado, será el mismo. Por tanto, la compatilidad está totalmente garantizada, aunque es recomendable que los miembros del mismo grupo usen el mismo IDE.

Yo tengo experiencia en ambos. Para aquel que tenga que aprender alguno de los dos, recomendaría eclipse, ya que es el más implantado (además de ser un entorno ya utilizado en otras asignaturas). Asímismo, la existencia de Mylyn lo hace el más recomendable para usar.

3.2 Características Comunes

Aunque no tengan mucha experiencia con IDEs suficientemente adecuados, supongo que conocerán las funcionalidades más usuales que suelen aportar: Editar código fuente con resaltado de sintaxis, abrir fácilmente múltiples ficheros del proyecto, encargarse de compilar y ejecutar los programas (sin necesidad de crear un *Makefile* o algo similar *a mano*),....

Otra opción que ambos IDEs admiten es la inclusión de un depurador de código. Para poder terminar las prácticas correctamente a tiempo **es necesario saber utilizar el depurador**. Es una herramienta básica de todo desarrollador, sin la cual el tiempo de desarrollo se llega a extender demasiado.

Sin embargo, es posible, que no sepan que ambos entornos ofrecen ciertas características avanzadas.

3.2.1 Autocompletado

A menudo es pesado tener que recordar el nombre de las variables y/o métodos, y esa repetición es proclive a errores. Esto es especialmente molesto en Java, ya que es un lenguaje bastante explícito. Una funcionalidad que se vuelve necesario es un autocompletado inteligente. En ambos IDEs, al empezar a escribir una variable y pulsar **CTRL+SPACE**, se muestran los posibles nombres, a elegir. Si es únicamente uno, se completa.

Con esta característica no hay excusas para no usar variables descriptivas, aunque sean más largas de escribir :-).

Otra opción muy interesante del autocompletado, es que cuando se desea llamar a un método de un objeto, con pulsar su nombre seguido del punto, **objeto.**, y pulsar **CTRL+SPACE**, muestra los distintos métodos posibles (métodos públicos de esa clase), incluyendo los parámetros necesarios. Es muy útil a la hora de utilizar las clases.

3.2.2 Detección y resaltado dinámico de errores

Es una característica muy útil que evita muchas compilaciones. Conforme se va escribiendo, va detectando posibles errores sintácticos, y los marca de color rojo. Así, una vez terminado un trozo de código (sentencia, función), la existencia de marca rojas indica errores a arreglar. De esta manera, se pueden arreglar errores sin esperar a compilar el proyecto.

3.2.3 Refactorización

A menudo es necesario hacer cambios en el código, no para aumentar la escalabilidad, sino para mejorar el diseño. Ha estos cambios se les denomina refactorización. Es una técnica consistente en una serie de cambios pequeños, para mejorar el mantenimiento y legibilidad del código.

Dado que introducir un pequeño cambio (como cambiar el nombre de un método o clase) puede ser muy laborioso, los IDEs actuales ofrecen utilidades para ello (por ejemplo, renombrar una clase o método cambiando automáticamente todas sus referencias del proyecto.

3.2.4 Pruebas automáticas

Como vemos en el apartado de pruebas automáticas, para comprobar el correcto funcionamiento de la funcionalidad implementada, es necesario que el proyecto pase determinados casos de tests automáticos.

Los IDEs permiten facilitar la creación de los casos de tests, creando el esqueleto, pudiendo centrarse en los métodos de tests. Tambien permiten facilitar su ejecución con una simple tecla, y comprobar el número de tests pasados (y en qué ejemplos no se han pasado los tests).

3.2.5 Gestión de librerías

Tal y como se ha indicado, se usará Maven para instalar las librerías. Ambos IDEs permiten un uso muy sencillo.

Antes de poder trabajar y/o crear el proyecto bajo Eclipse, es necesario instalar el plugin M2Eclipse, dicho plugin, al igual que múltiples plugins, pueden instalarse directamente desde el propio entorno de Eclipse, sin problemas.

En primer lugar, es necesario crear el nuevo proyecto como un proyecto Maven, para poder usarlo. Si se crea el proyecto como otro tipo de proyecto no podrá usarse, habrá que crear el proyecto de nuevo.

Un vez creado el proyecto, se puede incluir dependencias de las librerías que queremos, indicando incluso el número de versión mínimo requeridad. Ambos IDEs ofrecen buscar las librerías.

Por defecto, ambos poseen una serie de repositorios, pero puede aumentarse esa lista para poder instalar librerías adicionales (todas las librerías recomendadas en esta guía poseen un repositorio maven). El concepto es muy similar a añadir un nuevo repositorio Ubuntu usando la aplicación *Origenes del Software*, o directamente usando *apt-add-repository*. Para acceder al repositorio sólo hay que seguir los enlaces de las librerías de la documentación.

En los enlaces se puede ver cómo se puede añadir librerías en Eclipse y añadir librerías en Netbeans usando Maven. De todas maneras, se explicará la instalación de las librerías en prácticas.

3.2.6 Herramientas de trabajo en grupo

Los entornos permiten facilidades de trabajo en grupo, como el soporte de los sistemas de control de versiones comentados en la documentación. En el siguiente apartado se detalla cómo se usa.

4 JUnit: Pruebas automáticas

Como veremos en clase, es necesario para tener un cierto nivel de confianza en el código realizar un conjunto de pruebas. Aunque inicialmente se suele hacer *a mano* (es decir, invirtiendo horas haciendo pruebas con valores introducidos a mano y comprobados los resultados). Esa forma de hacer pruebas es

intuitiva, pero es demasiado laboriosa, y no se suelen repetir. Por tanto, cuando al modificar el código e introducir errores, éstos no suelen ser detectados.

Otra forma de abordar las pruebas es mediante el empleo de programas automáticos que realicen tests. Esto permite probar de forma periódica la funcionalidad implementada, y tener ejemplos directos cuando no funciona de forma correcta (es más fácil de depurar al tener ona región pequeña de código que no funciona correctamente).

En Java, esta funcionalidad se implementa por medio de la librería JUnit, que está muy documentada en la Red, y con distintos ejemplos. De todas formas, invertiremos una sesión práctica con las pruebas automáticas.

La idea es el de crear clases adicionales con distintos métodos encargados de probar el código. **JUnit** se diseñó para crear pruebas de unidad, por lo que es común tener por cada clase que ofrece una clara funcionalidad, una clase de Tests, asegurando que el resultado sea el esperado, para distintas situaciones.

5 Maven: Añadiendo Librerías

Un primer problema a la hora de trabajar con un IDE y/o con un *framework* (como los *frameworks webs* para Java) es que cada uno de ellos implica una determinada estructura de ficheros y directorios, haciendo difícil compilar y/o desarrollar sin tener dicho entorno.

Otro problema que se presenta es el de la instalación de librerías, problema muy dependiente de las librerías.

Ambos motivos obligaban a tener que usar el mismo IDE con el que se desarrolló para compilar el proyecto. Para resolverlo surgió un estándar de compilación, denominado **ant** (equivalente al uso de **make** para C/C++). Dicho programa, parte de una definición en un fichero *xml* denominado **build.xml** que permite establecer las dependencias.

Con el uso de **ant**, mejoró el proceso, pero no se resolvió el problema de una estructura común ni facilitar la instalación de librerías.

Maven surgió como una solución a este problema. Maven es un sistema que ofrece las mismas funcionalidades que **ant** (lo llama internamente) pero presenta grandes ventajas.

Explicar el uso de **Maven** excede las pretensiones de esta guía, por lo que paso simplemente a comentar dos aspectos importantes para el desarrollo de la práctica.

5.1 Definición de la estructura

Al crear el proyecto como un proyecto Maven, el propio sistema se encarga de crear una estructura de ficheros y directorios que es estándar. Por tanto, una vez creado, es posible compilarlo sin necesidad de tener el IDE instalado, únicamente con Maven.

A su vez, aunque no lo veremos en la práctica, el Maven también permite definir fácilmente estructuras necesarias para desarrollar en determinados *frameworks*, facilitando mucho su desarrollo (puede generar el esqueleto de la casi totalidad de *frameworks web*).

Ejemplo en video usando m2eclipse.

5.2 Instalar librerías

A cualquier usuario de Linux agradece la facilidad de instalar aplicaciones directamente por medio de un conjunto de repositorios. Esto permite no sólo un lugar en donde encontrar la aplicación, si no que también permite, a la hora de desarrollar una aplicación que dependa de una librería, poder indicar dicha dependencia. De esta forma, cuando se solicita la instalación de la aplicación, automáticamente se descarga e instala también las librerías de las que depende, haciendo el proceso de instalación muy sencillo tanto para el usuario como para el/la desarrollador/a.

Maven permite realizar esta misma tarea. Por medio del sistema Maven (se puede hacer desde el propio IDE) se puede indicar que una aplicación requiere una determinada librería. Así, a la hora de compilar y/o ejecutar el programa, maven instalará las librerías si no están ya instaladas, por medio de repositorios externo.

Esto permite que a la hora de distribuir un proyecto *maven* sólo sea necesario distribuir vuestro código, sin preocuparse de las librerías.

Además, permite indicar la versión de las librerías, evitando cualquier tipo de problema de versiones.

Esto, aunque algo más complejo desde un punto de vista técnico, con el uso de IDEs que soportan este sistema (se puede ver su uso, muy similar, tanto en Eclipse como en Netbeans), acaba siendo más sencillo que instalar las librerías a mano.

En prácticas vamos a ver cómo crear un proyecto maven e instalar dependencias. Para un primer vistazo, podemos ver en el siguiente video de adición de dependencias en ma2eclipse.

6 Persistencia

No serviría de nada una aplicación que no almacenase las tareas/citas para que estuviesen disponibles en las siguientes ejecuciones.

Un enfoque directo sería el uso de una Base de Datos (BD), y guardar los datos en ella. Afortunadamente, en el API de Java se ofrecía un interfaz común para las distintas bases de datos, por lo que toda librería que funcione bajo Java es independiente de la Base de Datos que tengamos instalada, pudiendo cambiar una por otra simplemente cambiando la configuración. Es decir, no existen dependencias innecesarias entre la Base de Datos concreta y el código de la aplicación.

Existen varios tipos de bases de datos, para distinto tipo de aplicaciones.

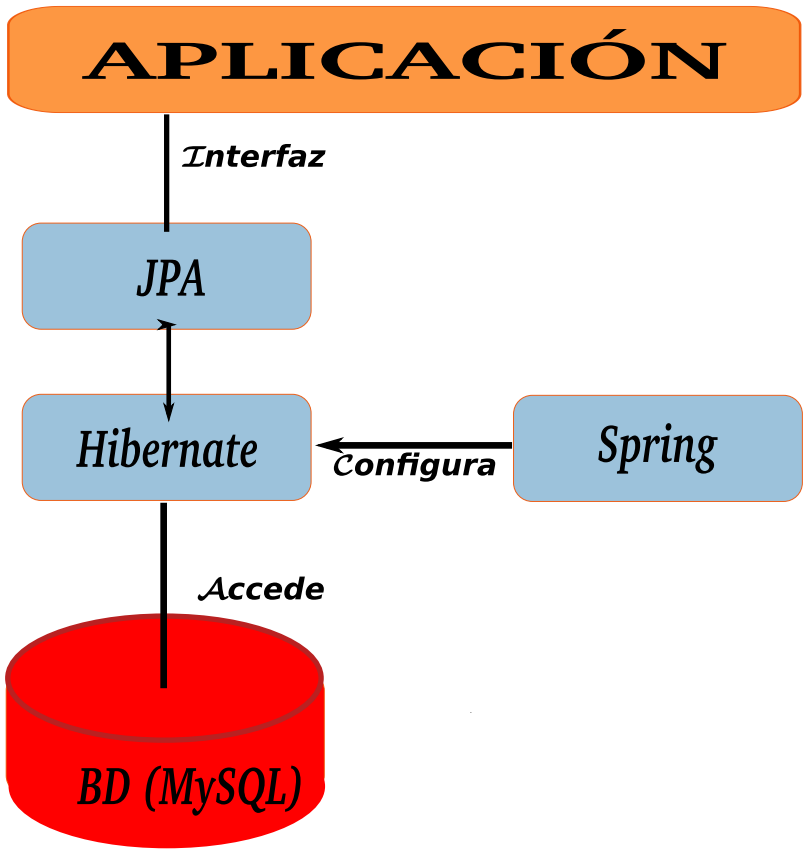
- Por un lado, las aplicaciones simples pueden usar bases de datos empotradas (en la que es la propia librería de Java la encargada de gestionar los datos, sin necesidad de instalar un proceso de Base de Datos).
- Por el otro, las aplicaciones más complejas, o web, requieren un sistema de BD relacional externo, que permite el compartir información entre aplicaciones.

En el caso de BD relacionales, se puede acceder directamente a la Base de Datos haciendo uso del SQL, o por medio de una herramienta ORM, que permita asociar una clase con una tabla de la Base de Datos, y permite recuperar y almacenar los objetos en la base de datos. Un ejemplo de un ORM muy popular bajo Java es el Hibernate.

6.1 Conceptos generales

Aunque existen muchos tutoriales libremente disponibles, no está de más introducir algunos conceptos asociados para el hibernate.

En primer lugar, la idea de hibernate es permitir almacenar y recuperar una clase Java en una base de datos relacional. Permite guardarlas en distintos tipos de Bases de Datos, aunque para las pruebas usaremos el MySQL para trabajar (aunque para depurar podemos una Base de Datos empotrada, ya que es más rápida).



- Como se ve, la aplicación/librería utiliza un interfaz estándar JPA (Java Persistence API) que es interpretada por Hibernate, que es la que se conectará finalmente con la Base de Datos.
- La aplicación será configurada por otra librería denominada Spring, ya que ofrece un método de configuración más flexible y cómodo.

6.2 Definición de clases ORM

Para poder guardar una clase en una Base de Datos, es necesario establecer la correspondencia entre los atributos de dicha clase y las columnas de las tablas en donde se almacenan. Existen dos formas de proceder. Por un lado, el formato clásico era definir el objeto sin ningún tipo de información y en un fichero xml aparte (persistence.xml) almacenar dicha relación. Sin embargo, la tendencia actual (y nuevo estándar) es añadir una serie de términos (empezados por @) en el propio fichero Java que permite indicar dicha correspondencia.

La clase no tiene por qué heredar de ninguna clase particular, y debe poseer los atributos a guardar con sus métodos de acceso correspondientes. Para que el sistema almacene la clase hay que añadir los siguientes atributos:

@Entity Define que la clase correspondiente se almacenará en la Base de Datos.

@Table(name="*tablename*") Define el nombre de la tabla asociada a dicha entidad (si no se indica será el nombre de la clase).

Luego, antes de la declaración de cada atributo se pueden indicar propiedades.

@Id @GeneratedValue Permiten definir el atributo siguiente como clave primaria de la tabla, se autogenerará su valor con cada nuevo objeto.

@Column(name="*name*", nullable=true/false) Permite indicar el nombre de la columna que guarda el valor, permitiendo indicar si se admite valor nulo o no en la Base de Datos.

@Basic(optional=true/false) Permite indicar al ORM si el atributo puede ser nulo o no.

Un par de ejemplos:

```
@Entity
@Table(name="author")
public class Author {
        @Id @GeneratedValue long id;
        @Column(name="name", nullable=false) @Basic(optional=false)
        private String name;
        @Column(name="country", nullable=false) @Basic(optional=false)
        private String country;
        ...
}
```

¬øCómo se puede reflejar la relación entre tablas? Evidentemente, por medio de referencias cruzadas. Para indicar que un atributo debe de obtenerse a partir de otra tabla es necesario añadir otro tipo de información. Los atributos son los siguientes:

OneToMany(cascade=ALL) Permite definir que el atributo siguiente (contenedora) es una referencia. El parámetro *cascade* permite hacer borrados/modificaciones en cascada.

ManyToOne(cascade=ALL, mappedBy="otherclase") Permite indicar que el atributo indicado (anónimo) referencia a otra clase. mappedBy es opcional permite indicar el atributo de la otra clase (si la relación es biyectiva).

Se puede consultar más información en el Wikibooks sobre relaciones usando JPA. Un ejemplo sencillo sería indicar que un libro puede tener un (único) autor se reflejaría de la siguiente forma.

```
@Entity
@Table(name="book")
public class Book {
        ...
        @ManyToOne(optional=false, cascade=CascadeType.ALL)
        private Author author;
        ...
}
```

6.3 Uso de la Base de Datos

Para acceder a la Base de Datos, una vez definidas las clases que vamos a relacionar en la Base de Datos, necesitamos un objeto **session** de tipo SessionFactory (en la configuración vemos cómo obtenerlo).

6.3.1 Recuperar un objeto

Si tenemos el *id* de un objeto, podemos recuperarlo simplemente con *session.load(Class, id)*. ejemplo: session.load(Author.class, id).

En el caso más frecuente no tenemos el *id*, pero sí un conjunto de restricciones, y podemos obtenerlas mediante la instrucción **createQuery** que posee la siguiente sintaxis:

```
session.createQuery(from XXX as XX where XXX.yy ...)
```

Esto permite definir una consulta. A la hora de poner la condición no se debe de crear una cadena con los valores concretos, ya que puede generarse todo tipos de problemas. La opción es uso de parámetros (empezados por el símbolo ':').

Es decir, en vez de hacer

```
Query q = sess.createQuery("from DomesticCat cat where cat.name = " +name);
```

Se debe de hacer de la siguiente forma:

```
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", name);
```

Los métodos para asignarle valores a los parámetros son: *setString*, *setDate*, ... Se puede consultar la documentación.

Una vez asignado valores a los parámetros, es necesario indicar que se devuelva los resultados. Se puede indicar que se devuelva un único resultado, con uniqueResult (si existe sólo un elemento, devuelve null si no lo encuentra).

```
...
q.setString("name", name);
Cat cat = q.uniqueResult();
```

Otra opción es indicar que se desea un grupo de objetos, mediante el método **list**.

```
...
q.setString("race", race);
List<Cat> cats = q.list();
```

6.3.2 Modificar y guardar un objeto

Se modifica recuperando el objeto de la Base de Datos, modificando los atributos que queramos, y volviéndolos a almacenar en la Base de datos.

Un ejemplo tonto, que cambia para todos los gatos de raza 'angola' por 'Angola':

```
List<Cat> cats = session.createQuery("from DomesticCat cat where race = :race").setString(
    "race", "angola").list();


for (Cat cat : cats) {
    cat.setRace("Angola");
}


session.save(cat);
```

Para borrar, existe el método delete().

6.4 Configuración

Dado que se ofrece un ejemplo en el campus virtual, nos centramos en explicar los componentes que deben o pueden personalizarse.

Como en todo proyecto Maven la descarga instalación de las librerías Hibernate y Spring se realiza de forma automática por medio del fichero *pom.xml*.

La configuración de la configuración de la Base de Datos se hace por medio del fichero **beans.xml** en el directorio *resources*.

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/databasename"/>
    <property name="username" value="usuario"/>
    <property name="password" value="clave"/>
  </bean>
```

Para adaptarlo a otra configuración sólo hay que cambiar las siguientes propiedades:

driverClassName Contiene el nombre de la clase que conecta con la Base de Datos. No hay que cambiarla si se usa el mysql, sí su se desea usar otra Base de Datos (consultar la documentación de dicha BD).

url La url contiene la información de conexión, puede depender de la base de datos. En este caso sólo se cambiaría el nombre de la base de datos (*database*). Evidentemente, *localhost* es el servidor que contiene la Base de Datos.

username Usuario con el que se va a acceder a la base de datos (un usuario *normal*, no *root*, pero con todos los permisos para la base de datos indicada).

password Contraseña del usuario de la base de datos.

Adicionalmente, hay que indicar las clases que se relacionarán con la Base de Datos (y que deberán de poseer los decoradores @Entity, ...). Esto se realiza añadiendo el nombre completo de las clases en la lista de la propiedad **annotatedClasses** del fichero *beans.xml*. Un posible ejemplo sería el siguiente:

```
<property name="annotatedClasses">
        <list>
                <value>org.uca.dss.example.data.Author</value>
                <value>org.uca.dss.example.data.Book</value>
                ...
        </list>
</property>
```

Por último, necesitamos que las clases que nos permiten operar con la Base de Datos contengan un objeto de tipo **SessionFactory** para poder comunicarse con la Base de Datos (usando el interfaz JPA). ¬øY cómo se puede iniciar ese objeto? Spring es capaz de crear objetos con dicho atributo inicializado. Para ello, es necesario indicarlo en el fichero *beans.xml* con una notación como la siguiente:

```
<bean id="autores" class="org.uca.dss.example.dao.RealAuthors">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
  <bean id="libros" class="org.uca.dss.example.dao.RealBooks">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
```

De esta manera, el atributo **sessionFactory** estará adecuadamente iniciado para los objetos creados autores y libros de las clases correspondientes. Para acceder a dichos objetos dentro del código sólo es necesario hacer:

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
Authors autores = (Authors) ctx.getBean("autores");
Books libros = (Books) ctx.getBean("libros");
```

en donde Authors y Books son interfaces (que las clases RealAuthors y RealBooks cumplen).

7 Trabajando en equipo, usando una forja

Para poder trabajar en equipo es necesario el uso de una serie de herramientas que permita trabajar sobre el mismo proyecto, y evitar tener posibles conflictos.

Lo primero que hay que hacer es asegurarse que se trabaja sobre el mismo código, para lo cual es necesario un Sistema de Control de versiones. Otras opciones que nos permiten es mantener un **wiki** en el que documentar y un sistema de tickets. Este tipo de herramientas son especialmente importantes ya que permiten trabajar de forma físicamente separada.

7.1 Configurando un proyecto

Para permitir trabajar físicamente separado es necesario que toda la información del proyecto esté disponible en un servidor conectado a internet para permitir acceder a sus miembros. Afortunadamente, existen distintos servidores, forjas, libremente disponibles. Un ejemplo sería la forja universitaria de red iris. También existen distintas forjas para otros sistemas de control de versiones, como Github o BitBucket.

En nuestro caso, dado que vamos a usar el Subversion, y queremos usar un sistema de tickets, vamos a usar la página web assembla que nos permite usar ambos, con un interfaz relativamente sencilla.

El proceso es el siguiente:

1. Acceder a la web de assembla: http://www.assembla.com
2. Elegir crear un espacio: Cree un Espacio -> Cree un Proyecto Público -> Hosting de Subversion con Tickets Integrados.
3. Inscribirse como usuario de Assembla.
   (a) Darse de alta como usuario.
   (b) Elegir como id dss2012-grupoX-nombre, identificándose como grupo, y un nombre corto del proyecto.
4. Acceder a la url del proyecto.

7.2 Compartiendo código: Sistema de Control de versiones

Como podrán notar, no es lo mismo desarrollar código en separado (como están más acostumbrados) a hacerlo en equipo. Además de las múltiples dificultades comunicativas y organizativas, se añade la dificultad de trabajar sobre el mismo código.

Si no se trabaja sobre el mismo código, es difícil trabajar, ya que hay que estar enviando los ficheros cambiados, con el riesgo de olvidar enviar clases cambiadas, o hacer cambios incompatibles entre sí, con lo que se puede generar situaciones muy difícil.

Sin embargo, desde hace muchos años existe una solución, utilizar un servidor que guarde vuestro código, y sincronizar vuestro código con dicho servidor. Existen sistemas que permiten esto, denominados **Sistemas de Control de Versiones** (estoy simplificando mucho como detectará aquel ó aquella que tenga conocimiento previo de estos sistemas).

De esta manera, se puede subir siempre todos los cambios realizados, y tener la confianza de trabajar con el mismo código.

El uso de un Sistema de Control de Versiones permite:

- Mantener centralizado el código, evitando el riesgo de que cada desarrollador tenga una versión diferente.
- Que los desarrolladores puedan acceder siempre al código actualizado.
- Mantener todas las versiones, y no sólo la última versión. De esta manera se permite *volver atrás* si hay un error.

- Los desarrolladores pueden bajarse el código actual, aplicar cambios, y subir sus cambios al repositorio.

La idea de trabajo es la siguiente:

- Primero, se descarga el código del servidor.
- Se hacen cambios locales, hasta tener una versión más avanzada, que al menos compile.
- Se sube el código cambiado al servidor.

Además, en todo momento, se puede descargar la nueva versión del servidor, detectando posibles cambios incompatibles (colisiones), en cuyo caso avisa de los cambios incompatibles.

Como se trabaja con los cambios, varios desarrolladores pueden, por ejemplo, añadir distintos métodos a la misma clase, sin que se produzca ningún problema, lo cual simplifica mucho el trabajo de integración. Un cambio incompatible sería que dos desarrolladores cambiasen simultáneamente la misma sentencia, o bloque.

En caso de detectar cambios incompatibles, el seguno desarrollador recibiría un aviso al intentar *subir* su código, y deberá de adaptar su cambio al cambio anterior.

Hay que recordar que un sistema de versiones espera un cierto grado de organización entre los desarrolladores, no es una herramienta de sincronización, simplemente garantiza coherencia en el código.

Otra opción que permite un sistema de control de versiones, es que guarda todas las versiones, no sólo la última versión, con lo que siempre se puede volver a una versión anterior en el caso de cometer un error en el desarrollo, quitando el *miedo al cambio*. Además, esto implica que no hay que desactivar código poniéndolo entre comentarios, si un código no se utiliza se debe de borrar, ya que siempre se podria recuperar si es necesario (aunque no suele ser necesario).

7.2.1 Acceder al repositorio del proyecto

Dentro de la web del proyecto se indica una URL que permite indicar al subversion dónde se encuentra el repositorio remoto. Un ejemplo sería la URL http://subversion.assembla.com/svn/dss2012-hibernate-ejemplo/

Dentro de esa URL existe un fichero README y un directorio *trunk*. El código fuente de nuestra aplicación debe estar en dicho directorio. En nuestro caso, no se guardará únicamente el código fuente, si no también los ficheros de configuración (*pom.xml*, *beans.xml*, ...). El propio entorno IDE es capaz de trabajar usando el SCV, por lo que se encargará de añadir dichos ficheros al repositorio.

7.2.2 Soporte del IDE de los SCV

Existen múltiples sistemas de control de versiones, pero los más utilizados son Subversion (modelo centralizado) y el **Git** (modelo de desarrollo distribuido). En este trabajo vamos a hacer uso del **Subversion**.

Ambos están soportados por los IDEs recomendados, pero hay que recordar instalar el plugin Subeclipse para poder usarlo (se instala directamente desde la configuración del eclipse, al igual que con el otro plugin).

Para entender su uso, mejor que mirar directamente un tutorial de Subversion (que explicará cómo usarlo bajo linea de comandos), es mejor aprender cómo se puede usar en los IDEs (obteniendo una visión más directa). En los enlaces siguientes hay un tutorial de Subversion bajo Eclipse y otro tutorial de Subversion bajo Netbeans.

7.3 Documentando con Wiki

Dado que la documentación de una aplicación (incluyendo la documentación de diseño) es algo vivo, ¿qué mejor forma que usar la wiki del proyecto para documentarlo?

En el wiki se va a copiar la documentación entregada en la primera entrega, y se modificará los cambios sobre él (incluso los diagramas), permitiendo de ese modo tener un histórico de la documentación.

7.4 Sistema de Tickets

Por hacer.