If you want to store things beyond the lifetime of a single instance of a program then you use a database. If you want to write programs nowadays then you use an object oriented language and work with objects. If you want to do both then you need an object database in which to store the objects. Most people do this with an object-relational mapping (ORM) which store the objects in a relational database but still maintains their connectivity and identity as objects. Hibernate is one such ORM technology.

Hibernate is an open source java framework designed to store java objects and their dependencies in a relational database. Another similar technology is Java Data Objects (JDO) - part of the J2EE collection of technologies. they are so similar in fact, that I didn't care which I used and based my selection on a survey of contracting job adverts in Germany, where Hibernate was vastly more in demand...

Hibernate can be done in two ways - either design your objects and let hibernate work out how to store them or write your database and let hibernate work out how to represent it in Java. (Obviously any realistic approach will require developer awareness and understanding of both ends of the mapping.) I used Hibernate Tools for Eclipse which has a reverse-engineering function to create the hibernate mapping and java class definitions from an existing database. i.e. I wrote my database first - but then, I know from previous experience what object mappings should look like.

Access to the MySQL database was via JDBC (Java Database Connectivity) and the MySQL Connector driver. I also used JNDI (Java Naming and Directory Interface) to identify the database, but in the final version I took that out because the shared tomcat server (mochahost) did not allow me to specify JNDI names without going through their tech support. On Tomcat, the JNDI resources need to be defined in context.xml (which can be in different places ontheserver, depending on it's scope) and a resource-ref needs to be defined in the application's web.xml.

Also, all text was stored in UTF-8 to allow for internationalisation. In fact, this was a policy I

used throughout the entire development (joomla is already built on utf-8).

So, working with the eclipse hibernate tools we have the following files:

- Hibernate.cfg.xml - this stores the general information, such as where to find the database and how to connect to it. It also references the individual class mapping files below. It defines a session factory object.
- For each table/java class there is a Myclass.hbm.xml which maps the columns in the database table to the java class members
- For each class there is a java file with the appropriate get/set methods for the fields and a separate data access object (DAO), also a java class and with a wrapping for some of the most useful methods, such as FindByID(...) etc. The DAO also wraps access to the session factory so if you have used a non-default name in your hibernate.cfg.xml then you will have to edit these files to match.

One complication is that the hibernate reverse engineering required a unique primary key for each table or it would try to create a compound key object - which I did not want.

Another MySQL complication is that databases can use different engines MyIsam and InnoDB - I was using InnoDB and the joomla tables were using MyISAM which meant that I was not able to create a foreign key in my tables refering to one of the joomla tables. Took me a while to find that out though.

On Eclipse, I also had to setup a javax.naming.InitialContext using org.mortbay.naming.InitialContextFactory. This required the jetty.jar. Tomcat has its' own naming context so this was not needed on the tomcat installations.

Getting all the required jars together was not that easy - especially as it was three different sets, depending on whether I was running in eclipse, my tomcat, or - later - the shared tomcat at

mochahost.

Here are the lists:

On All configurations:

- /CryptoLeague/war/WEB-INF/lib/cryptoleague.jar (My Application!)
- /CryptoLeague/war/WEB-INF/lib/commons-collections-3.2.1.jar
- /CryptoLeague/war/WEB-INF/lib/commons-logging-1.1.1.jar
- /CryptoLeague/war/WEB-INF/lib/dom4j-2.0.0-ALPHA-2.jar (document object model)
- /CryptoLeague/war/WEB-INF/lib/gwt-servlet.jar (the google web toolkit)
- /CryptoLeague/war/WEB-INF/lib/hibernate3.jar (the Hibernate code)
- /CryptoLeague/war/WEB-INF/lib/javassist.jar
- /CryptoLeague/war/WEB-INF/lib/jta-1.0.1B.jar (java transaction api)
- /CryptoLeague/war/WEB-INF/lib/log4j-1.2.16.jar (log4j logging utility)
- /CryptoLeague/war/WEB-INF/lib/slf4j-api-1.6.1.jar
- /CryptoLeague/war/WEB-INF/lib/slf4j-simple-1.6.1.jar

On Eclipse integrated debugging server:

- /CryptoLeague/war/WEB-INF/lib/hibernate-testing.jar
- /CryptoLeague/war/WEB-INF/lib/mysql-connector-java-5.1.12-bin.jar
- /CryptoLeague/war/WEB-INF/lib/jetty-naming-6.0.0.jar

On local tomact server:

- /CryptoLeague/war/WEB-INF/lib/mysql-connector-java-5.1.12-bin.jar

On my local tomcat I put the MySQL connector jar into the tomcat server-wide lib directory since

other applications will use this as well. Mochahost already had it installed in their server-wide configuration.