DeciBell: A novel approach to the O.R.M. software for Java™

Chomenides Ch., Sopasakis P., Sarimveis H.

May 2010

Minimalist Architecture

DeciBell is an *open source* and *free* tool developed to tackle in a uniform and structured way the problem of Java and SQL cooperation (available at `http://github.com/hampos/DeciBell`). In DeciBell, Java classes are related to relational database entities automatically and in a transparent way as far as the background operations are concerned. So, on one hand, non-expert users can work on Java code exclusively while expert ones are able to focus on more algorithmic aspects of the problem they try to solve rather than be wasted with trivial database management issues. In contrast to the existing O.R.M. programs, DeciBell does not require any configuration files or composite query structures, but only a proper annotation of certain fields of the classes. This annotation is carried out by means of the *Java Annotations* which is a modern trend in Java programming. Among its supported facilities, DeciBell supports primary keys (single and multiple), foreign keys, constraints, one-to-one, one-to-many, and many-to-many relations and all these using pure Java predicates and no SQL or other Query Languages.

DeciBell is developed to be easy to use for all Java developers as it assumes no expert knowledge. It is also meant to be efficient and secure as it makes use of solely *Prepared Statements*. In many cases it would be more efficient to develop a database management system which is application oriented and tailored to the needs of a specific software. However, the implementation and maintenance of such a system would be time consuming for the developer distracting him/her from the scope of the application itself. So DeciBell is more a convenient tool for DB management rather than an optimum solution.

DeciBell's main goal is to strike a balance between configurability, efficiency and complexity, putting the most of its effort to suppress the last one. DeciBell struggles to resemble more to an one-button washing machine rather to a state-ot-the-art Boing 747 cockpit.

Everything is a Component

In DeciBell every class one needs to map to a relational entity, is called a *Component*. Any class that needs to be persistent in the database using DeciBell has to subclass (directly or indirectly) the class `Component`. This is an abstract class that contains methods which enable the user to directly read from and write to

the database, as well as delete and update one or more database entries. The user does not need to interact directly or by any means with the database itself or be aware of anything about the database structure, SQL queries and Java-to-db-to-Java data transactions whatsoever. The class `Component` contains the following four method implementations:

- `public void` **register**`(DeciBell db) throws DuplicateKeyException, ImproperRegistration`: Register the object into the database. There is no need to specify the table(s) in which the data are stored or the way this registration is carried out.
- `public void` **delete**`(DeciBell db) throws NoUniqueFieldException`: Delete the specified object or an object that is uniquely identified by the object on which the method is applied. A primary key value or a unique field has to be provided thus pointing to some specific target object (entry).
- `public ArrayList<T>` **search**`(DeciBell db)`: Apply this method to some object to search for *objects* in the database that resemble to the given one. Fields that you wish to be ignored during the search should be set to `null`. No additional parameters need to be specified such as the SQL query. The search is based on one or more prepared statements.
- `public void` **update**`(DeciBell db) throws NoUniqueFieldException, DuplicateKeyException`: Update an object in the database with the given object. The object on which the method is applied uniquely idenifies some object in the database (using the given value for its primary key(s) or some unique field value). Then all other entries are updated with the corresponding data in the given component.

For increased security and performance, `SELECT`, `INSERT`, `UPDATE` and `DELETE` queries are prepared on startup. The prepared statements are supplied with parameters using the Java reflection API (`java.lang.reflect`) and the Java library Reflections (see `http://code.google.com/p/reflections`). However all reflective elements (Classes and Fields) are collected *on startup*, so no reflective lookups take place during runtime.

DeciBell Annotations

@PrimaryKey

A Primary Key, in ER-terms, is a characteristic *attribute* (entry) of an entity which is unique and not nullable. The primary key value of a table entry is an identifier for this entry. Every component in DeciBell should possess a primary key. One or more fields of the component (a Java class) must be annotated with the `@PrimaryKey` annotation.

@Entry

A simple annotation to declare that the underlying field should be mapped to a table column (or a set of table columns) in some table. No need to worry about the table structure when using this annotation. Database creation takes

into account all fields annotated with the `@Entry` annotation to create a proper structure that allows this information to be registered in and/or be read from the database. No knowledge of the database structure is needed, since you understand the relations between the classes you have defined, which is the normal Java (possibly hierarchical) structure. The following methods are contained in this annotation, allowing the user to specify certain meta-information about the annotated field. These are:

1. `boolean unique() default false`: The entry is unique among all entities. Every field annotated with the `@Entry` annotation with the extra characterization `unique=true` should have unique values over all instances of that class, at least when database registration operations are performed.
2. `boolean notNull() default false`: The entry cannot accept `null` values.
3. `String defaultValue() default ""`: Default value to be used when one attempts to register a `null` value.
4. `boolean autoGenerated() default false`: Applies only to `integer` or `long` fields. Auto-generated fields acquire a value automatically upon registration. Note that at most one field can be auto-generated in every class.

@ForeignKey

In DeciBell, entities are allowed to point one another. So a `User` may point(have) to a `UserGroup`, or a `Product` may point to its `TechnicalSpecifications`. So, an attribute or a set of attributes in some entity *references* an attribute or a set of attributes in another entity by a `foreign key` constraint. The purpose of the foreign key is to ensure referential integrity of the data i.e. only values that are supposed to appear in an attribute of some other entity are permitted. Foreign keys are defined in the ANSI SQL Standard. Let us now give an example of how you can use `@ForeignKey` in your source code. Suppose you have two classes, namely `User` and `UserGroup` as follows:

```
class UserGroup {
   public String groupName;
  public String authorizationLevel;
}


class User {
   public String userName;
  public String password;
  public UserGroup group;
}
```

Then you can use the annotation `@ForeignKey` to declare that the field `public UserGroup group` points to the entity `UserGroup`. We note again here that every entity must be endowed with a `@PrimaryKey`. So modifying slightly the

above code, yields:

```
class UserGroup extends Component{
   @PrimaryKey public String groupName;
  @Entry public String authorizationLevel;
}


class User extends Component{
   @PrimaryKey public String userName;
  @Entry public String password;
  @ForeignKey public UserGroup group;
}
```

The following methods of the annotation enable the user to configure some properties of the foreign key such as its behavior `on delete` and `on update`. These methods are:

1. `OnModification onDelete() default OnModification.NO_ACTION`.
2. `OnModification onUpdate() default OnModification.NO_ACTION`.

@Constraint

In order to enhance data integrity in your database, you may add a constraint on an attribute. If the attribute is numeric (`int, long, short, double, float`) the constraint consists of a lower and a higher bound (numeric constraints), that is $x_{low} \leq x \leq x_{max}$. On the other hand, constraints on String attributes impose a finite range of values (nominal constraints), i.e. $x \in \{x_1, x_2, \ldots, x_n\}$. This Java Annotation Type defines the necessary methods for establishing such constraints on attributes (fields) of components. These are:

1. `String low() default ""`: Lower bound imposed on an attribute as a constraint.
2. `String high() default ""`: Upper bound imposed on an attribute as a constraint.
3. `String[] domain() default {""}`: Nominal constraint imposed on String fields.

@NumericNull

This DeciBell annotation is used to provide a *numeric null* value for database searching as the JVM does not allow for `null` values in numeric variables. The number provided using this interface, stands for the `null` value of the annotated variable.

The idea is simple: When a search command is executed on a DeciBell Component, DeciBell checks that component's Fields and when a Field has a `null` value it gets replaced by an indefinite value (i.e. a value used to inform the database that this specific field is to be ignored in the search, in the sense

to perform the search operation irrespective of the value of this field) In that way DeciBell keeps using the same PreparedStatements and greatly increasing performance even though some of the statements values have to be ignored.

The problem comes in when Java automatically initializes all primitive numerics with a 0 value (There is no meaning of `null` for primitives). This makes us unable to see if the field was initialized by the user or by the compiler. A new annotation then became clear. NumericNull is used to define the value on which a numeric primitive is considered null. The default value is -1. One can set a numeric null value other than that on each DeciBell Entry. If one is to use this annotation, they must provide a numeric value that they will not (or at least not expect to) use ever in their database for that field; otherwise this will lead to a considerable comfusion and wrong results.

@TableName

This annotation should concern only those users who need to directly access the automatically created database to inspect the data in it and/or understand how DeciBell works. So it accomplishes mainly educational purposes. Using `@TableName`, users can specify some arbitrary name for the generated table themselves, overriding the standard naming procedure of DeciBell. In general DeciBell creates too long table names which would be a considerable burden for someone who would need to directly access the database. This is a short example of use:

```
@TableName("YSER") class User extends Component{
  @PrimaryKey public String userName;
  @Entry public String password;
  @ForeignKey public UserGroup group;
}
```

This tells DeciBell that the table produced to host the attributes of the class `User` should be called "YSER".

Database creation

Users of DeciBell find it very easy to generate and maintain a database structure. The class `org.kinkydesign.decibell.DeciBell` is a user friendly database manager. It takes just a couple of steps (at most) to configure the manager. First, you have to specify which components will be used by DeciBell. You can either provide a target package which contains your components, i.e.

```
DeciBell db = new DeciBell();
db.attachFromPackages("com.my.components");
```

Or you can specify your components one by one, i.e.

```
DeciBell db = new DeciBell();
db.attach(User.class);
```

```
db.attach(UserGroup.class);
etc...
```

However this step is optional since if you skip it, DeciBell will search in all packages in your classpath and will attach every component therein.

Second, you have to set some database name of your choice and trigger the database creation engine. Here is an example of use:

```
db.setDbName("myDataBase");
db.start();
```

This will start the database server (DeciBell currently works only with Derby but we plan to add support for other database servers as well, such as MySQL) and creates the tables needed to store your data. Some statements are prepared and cached in a *pool*. If the database already exists and contains all necessary tables, no modification of its structure will take place. Once all transactions are over, call it a day using the following statement:

```
db.stop();
```

This will disconnect you from the database server and also send a stop signal to Derby.

Quering

Complex queries can be carried out using DeciBell. Let us give an example including three entities, namely `Master`, `Slave` and `Other`, having the following structure described by their Java classes:

```
class Master {
  @Primarykey @NumericNull("-20") long uid;
  @ForeignKey Other other;
}

class Other {
  @PrimaryKey String uid;
  @ForeignKey Slave slave;
}

class Slave {
@PrimaryKey URI uri;
@Entry String x;
}
```

DeciBell provides a way to get all `Master` objects having an `Other` which has some `Slave` with a value for `x` that contains the letter `d`. This is as simple as the following piece of code. First we construct a *prototype* for our search operation:

```
Slave slave = new Slave();
```

```
slave.x = "%k%";

Other other = new Other();
other.slave = slave;

Master prototype = new Master();
prototype.uid = -20; prototype.other = other;
```

Then we can search for all `Master` objects which reseble the given prototype. This is:

```
ArrayList<Master> componentsFound = prototype.search(db);
```

Component Relations

We encounter the problem of one-to-many and many-to-many component relations when a component contains some `Collection<? extends Component>` of other components. In many cases, the user denote that this collections is a `Set` or a `List`. This generates extra relational tables in the database used to store these data. Registration and Search are performed as mentioned before. The information contained in a Java Collection are stored in the database and can be also retrieved. We need to note that the retrieval of a Collection, brings back Java the Collection cast as the same datatype it was used when the data where registered. Here is an example of use.

Suppose we have two entities, namely `Master` and `Slave`, and there is a many-to-many relation between Masters and Slaves which is reflected in the Java code by means of a Collection-type field. Here is the structure of the two classes:

```
class Master {
  @Primarykey @NumericNull("-20") long uid;
  @ForeignKey Collection<Slave> collection;
}
```

And here is an example of registration and search operations using DeciBell:

```
Collection<Slave> coll_1 = new HashSet<Slave>();
coll_1.add(slave1);
coll_1.add(slave2);

Master master = new Master();
master.uid = 100;
master.collection = coll_1;
master.register(db);

Master prototype = new Master();
prototype.uid = -20;
ArrayList<Master> masters = prototype.search(db);
```

```
Master foundMaster = masters.get(0);
```

Then `foundMaster.collection` is a `HashSet`. That is, DeciBell also stores the collection implementation used when storing the data.

Implementation Details

DataTypes Mapping

Java basic datatypes (`int, long, short, double, float` and `String`) do not have an exact counterpart in SQL. It is intuitively implied that the Java datatype `int` is mapped to the SQL datatype `INTEGER` but those types differ both in number of allocated bytes and range. Specificaly, in Java integers accept values from -2^{31} up to $2^{31}-1$, while in SQL (and in particular in Derby) the lowest integer is -2147483648 and the highest is 2147483647. Analogously there are some datatype inconsistencies between `short` and `SMALLINT` and so on. However, as every ORM has to establish some correspondnce between datatypes in Decibell we adopt by convention the following mapping:

Table 1: Java-SQL datatypes mapping

Basic Java datatype	SQL datatype
`int`	`INTEGER`
`short`	`SMALLINT`
`long`	`BIGINT`
`double`	`DOUBLE PRECISION`
`float`	`REAL`
`String`	`VARCHAR(32672)`
`BigDecimal`	`DECIMAL`
`boolean`	`BIT`

Dependencies

DeciBell makes use of the following open source, free Java libraries:

1. **Reflections:** Reflections is an opensource/free java metadata analysis software distributed under the GNU Lesser GPL license. Project web page @ `http://code.google.com/p/reflections`. Reflections offers some advanced metadata analysis facilities and mainly configurable reflective lookup in packages.

2. **Derby JDBC server:** The Derby JDBC server is freely distributed under the Apache license. DeciBell makes use of the transitional alpha release of Derby 10.6.0. but will soon upgrade to the latest release 10.6.1 (see `http://wiki.apache.org/db-derby/DerbyTenSixOneRelease`)

3. **XStream:** Library for the (de)serialization of objects in Java to and from XML. It is used in DeciBell for registering in the database non-Component objects of unknown structure.It is an open source software

distributed under a BSD license. XStream is available for download at `http://xstream.codehaus.org`.

Contribute!

The quintessence of an open source project is that other developers can study the source code and contribute their ideas and proposals or even fork the project thus adjusting it to their specific needs. This is the reason why DeciBell is open source and needs the contribution of the open source community. One can either contribute in terms of optimizing the source code, adding new features or fixing existing bugs or write documentation for the existing code. Being in its infancy, DeciBell also needs lots of tests. Contact one of the developers for more information.

Does it make coffee?

Well, that's a tough one! It doesn't! But this is still it's alpha/testing version (0.1.1.0). All JUnit tests carried out till today are quite encouraging and we'll do our best so that it will make coffee (due for the beta version).

Found a Bug?

Did you find a bug? Please report it at `github.com/hampos/DeciBell/issues` (issue tracker) or send us an email.