Model Driven Development & Persistency

Rui Couto António Nestor Ribeiro

March 23, 2020

Contents

1 Introduction

The Game Management System (GMS) allows users to create a digital library of games. Users can create an account, and then associate with their account a videogame. The objective is to provide a tool for users to more easily manage large collections of videogames. As it is well known, a videogame has an associated platform. While each game belongs only to a platform, several games can have the same platform.

The proposal is to develop a system to support this digital library of video games, with the most common features:

1. Register a user;
2. Register a game;

3. Register a platform;
4. List user games;
5. List all games;
6. Search a game;
7. Delete a game.

Next follows a the description of the process to develop such system persistency layer in a model driven approach.

2 Creating the model

In order to adopt a model driven development methodology, the first step is to create the business model. The business models consists of the classes which define the system itself, in this case, as a *Class Diagram* model.

2.1 Classes

The proposed model to support the presented system is composed of a set of four classes, namely the `GMS`, `User`, `Game` and the `Platform`. This classes are the core of the GMS, and represent the minimal set of entities required to support the presented features.

2.2 Attributes/Methods

After the definition of the classes, their attributes should also be specified. So, for each class, follows a brief description of its attributes. In order to use the Hibernate framework, not only the attributes are required, but also the methods to access them (i.e. `get` and `set`).

System The `GMS` has no attributes other than the relationships with the classes it refers.

User Is the entity which represents a user of the system. So, at least we need its credentials and designation.

- Name: a string which represents the user designation (e.g. name, full name, or first name).

- Email: a contact is required, for instance for password recovery and confirmation features.
- Password: joint with the username, represent the credentials to access the system.

Game Is the core entity of the system, representing the items to be catalogued. The description of the main features of the game is required.

- Name: represents the title of the game, the easiest way to identify it.
- Year: the year of publication of the game.
- Price: the value of the game (e.g. the current price, or acquisition price).
- Description: a short description of the game (e.g. the box description).

Platform The platform represents the actual device (e.g. console) in which a game can be played.

- Name: the common name of the platform.
- Year: the year of release of the platform.
- Description: a brief description of the most relevant features of the console.
- Manufacturer: the name of the manufacturer of the device.

Tasks

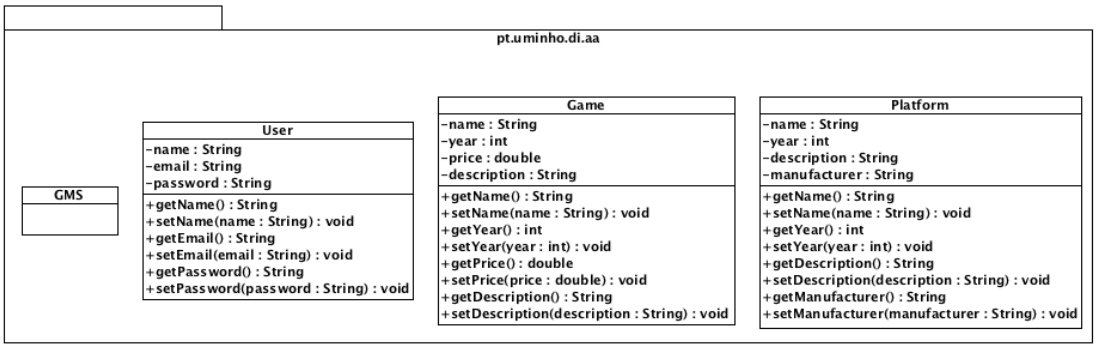
1. Create the the corresponding classes in Visual Paradigm (according to the provided information). Don't forget the corresponding getters and setters. The classes should be placed inside a package.

The diagram expected at this stage is as presented in Figure 1.

2.3 Relations

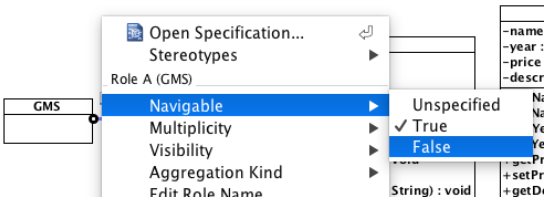
Properly defining the classes relationships is essential in order to generate the correct code. The associations will define not only their n-to-n relationships (concerning the source code), but also how the objects should be loaded (c.f. Hibernate). The process of specifying the associations requires three main components: **a)** the definition of navigability, **b)** the definition of the visibility, and, **c)** the definition of the multiplicities. The relation should also, as expected, have a name.

Figure 1: Class diagram with entities.



In Visual Paradigm, the navigability of a relationship can be set by right clicking in the end of the relation (see Figure 2). Next, define the desired multiplicity (i.e. true, false). In this case, we want to the `GMS` to be able to access the `User`, and the user to access the `Game`. The `Game` should be able to access the platform. The inverse associations are not required.

Figure 2: Defining navigability.



The visibility of the association should be set to `private`, in order to respect encapsulation. This can be set by right clicking an association, and define the visibility to private. Note that the indication - is not visible (contrary to the class attributes).

The multiplicities can be set, similarly to navigability, by right clicking the association end. In the system, the `GMS` should support several `User`, and a `User` has a collection of `Game`. A `Game` belongs only to a platform[1].

Tasks

1. For each class, define its relationships (as well as navigability, visibility and multiplicities properties).

[1]Although it is common for a game to exist in different platforms, we opt for define it as a different game

3 Code generation

Model specification tools usually have the capability to generate the code for the model. In Visual Paradigm, this feature is very useful in order to generate the foundation of the application itself. Furthermore, later in the tutorial will help the development by generating the persistency source code.

In order to generate source code, select `Tools -> Code Engineering -> Instant Generator -> Java...`. Select the model to generate, in the left side. Next, customise the parameters in the right side.

Note that there are several parameters available. An example are prefixes (`Attribute` and `Parameter`), which denote a character to be placed before the name of an attribute/ parameter. Another example is the definition of the associations kinds (`Vector`, `ArrayList`, etc.). The `Advanced Options...` allow to further customise the code generation process.

Tasks

1. Generate the code for the developed model. Analyse the code generation options available, and configure as desired[2].

2. Analyse the output in order to verify that everything is as expected (i.e. attributed, methods, relationships, etc.).

3. Modify the relationships from `Game` to `Platform` as follows, and analyse for each one the **source-code** and the **database-schema** generated by the Visual Paradigm:

   - 1-to-1 (`Game` → `Platform`)
   - 1-to-n (`Game` → `Platform`)
   - n-to-n (`Game` ↔ `Platform`)
   - n-to-1 (`Game` ↔ `Platform`)

   Identify the most appropriate association multiplicity and direction.

4. Update the model in order to correct any found error, or the code generation parameters to match the expected result.

[2]You can use the `Preview` feature before generating the source code

4 Persistency

The model created so far represents the business entities (i.e. classes). These entities correspond to the main components of the GMS system. However, and as seen by the generated code, they are POJOs (Pure Old Java Objects), which means that they do not have any associated feature (neither business logic, to be added by developers).

However, we want the capability to persist some of these classes. This can be done by specifying in the model such information. Then, tools will have the information regarding the entities which should be persistable. As a consequence, generating the source code, they will know for which entities to generate also persistency code.

The ORM persistency requires three main components:

1. The class diagram, with the specification of the entities to persist.
2. A database schema with the capability to support these classes.
3. The mapping information, with support for querying.

Since we already have a class diagram, next is presented how to automatically obtain the remaining artefacts.

4.1 Stereotypes

Stereotypes are the UML mechanism to add responsibilities (i.e. a special role) to classes. Such means that a class is expected to have more features rather than being just a POJO.

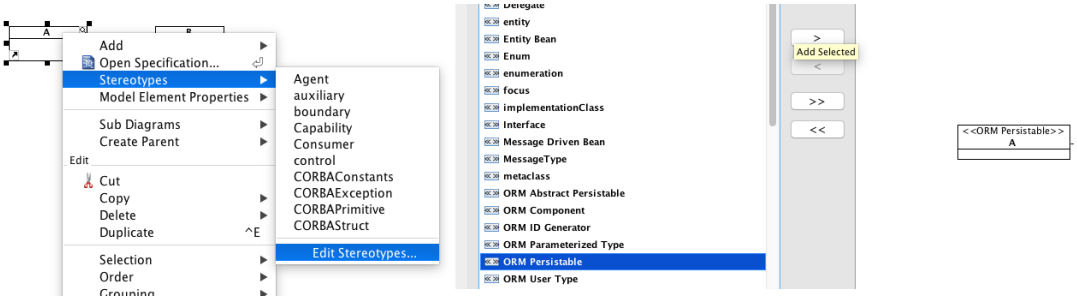
In the case of persistence capabilities, we are interested in the `ORM Persistable` stereotype. This stereotype denotes the classes as entities to be persisted by ORM.

The first step consists in identifying which classes need to be persisted.

- GMS: is a wrapper for all the entities. Thus there is no need to persist entities. It can however manage entities
- User: is a class to represent a user. Since it has individual information, we want to **persist** this entity.
- Game: the games are the entities which the users are interested in register, thus, it is required to **persist** them.
- Platform: although several games have the same platform, it is still required to **persist** it.

Adding a stereotype to a class in Visual Paradigm (see Figure 3) can be done by right clicking a class, `Stereotypes -> Edit Stereotypes...`. Next, select `ORM persistable` from the list, and the arrow button `>`. The class contains now a new visual element, `<<ORM Persistable>>`.

Figure 3: Setting stereotypes in Visual Paradigm.



Tasks

1. Specify the appropriate entities to be persisted.

4.2 Code generation

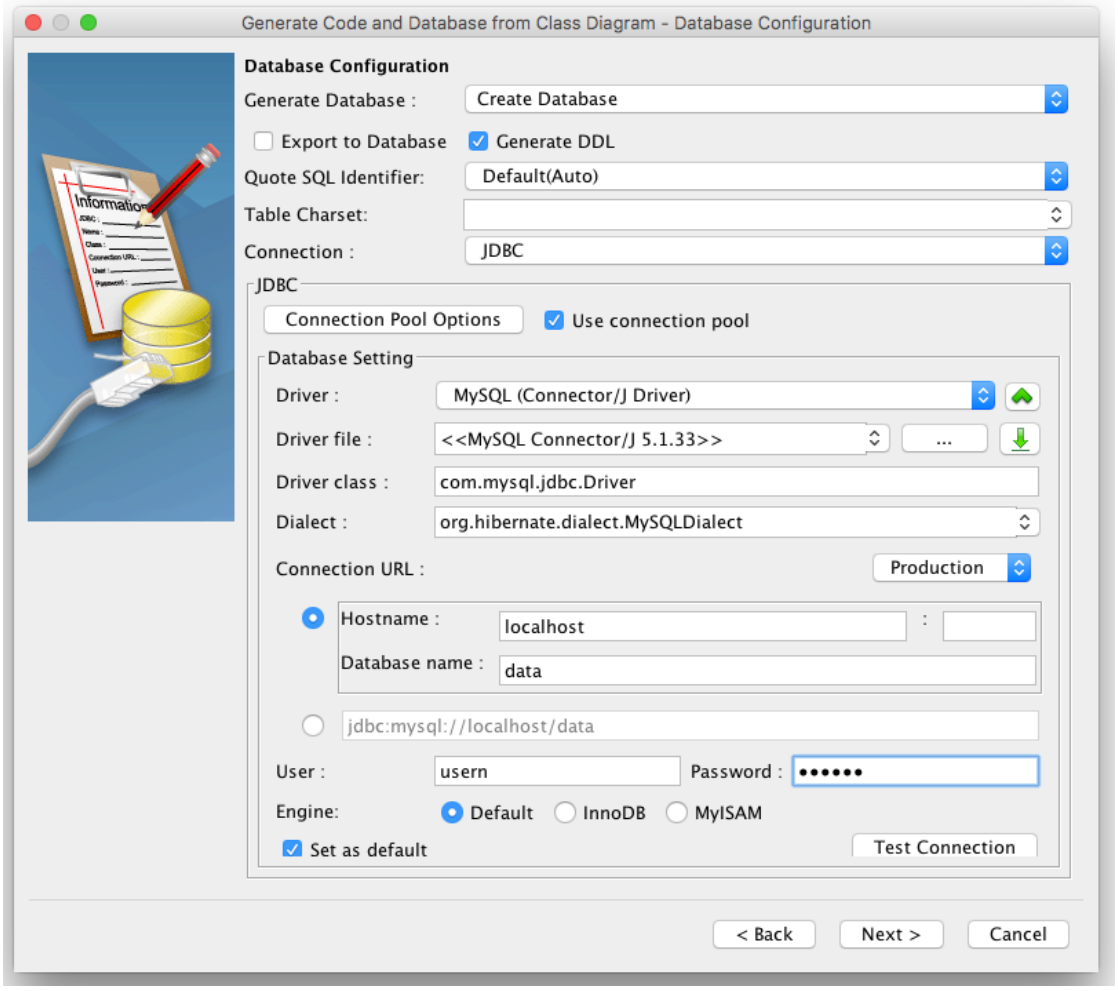
After setting the appropriate stereotypes, the modelling tools are able to interpret such information and generate the persistence code. In the case of Visual Paradigm, it is possible to automatically generate not only the database schema, but also the code to interact with that schema.

To generate the code, click in `Tools -> Hibernate -> Wizards...`. The wizard provides a step by step to guide the code generation process. In the first step select `Java`, and `Generate Code and Database from Class Diagram`. Make sure that the correct entities are marked in the next step. In the `Database configuration` step, provide the database details according to your setup. Some expected configurations are (see Figure 4):

- Generate Database: Create database;
- Export to database: Check - without this option, the database schema will not be generated;
- Connection: JDBC;
- Driver: According to your setup;

- Driver file: Select the green arrow to download automatically;
- Connection url: localhost;
- Database name: the name of the database previously created;
- User and Password: the database credentials.

Figure 4: Database configuration.



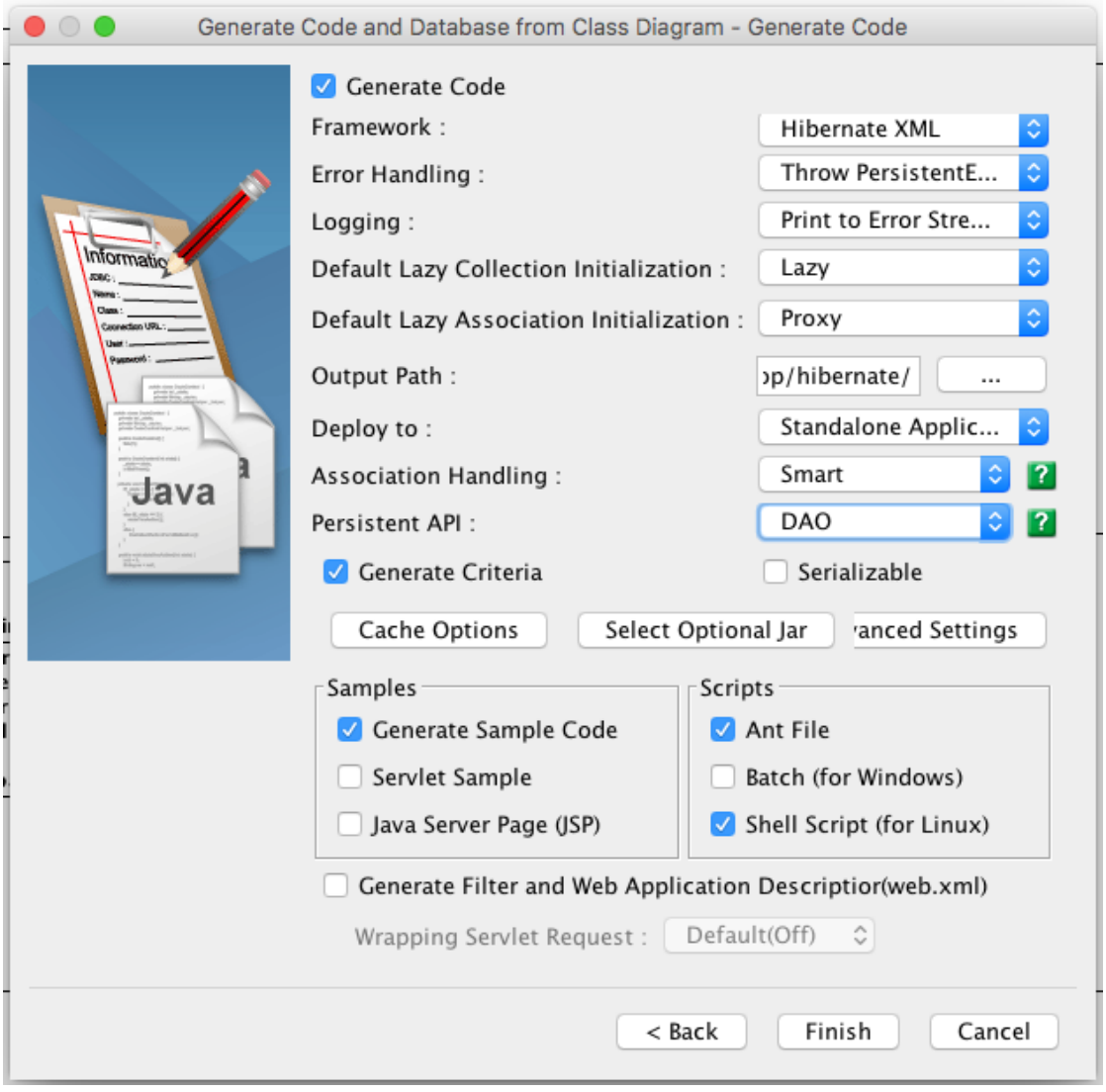
At the end, click `Test Connection`, and a `Connect Successful` success message should appear if everything is correct. Otherwise, check the provided data and try again.

By clicking in next, the Hibernate configuration window appears (see Figure 5 for an example). Several options are available, Special attention should be paid to:

1. The output path in order to select where the code will be generated;
2. The framework, since we are interested in Hibernate;
3. The `Persistency API`, in order to use the appropriate mechanism.

When ready, click in `Finish` to generate the source code. A set of files and classes are created in order to manage the entities. This project can now be compiled, or open in an IDE in order to be developed.

Figure 5: Code generation configuration.



Tasks

1. Generate the code for the developed model.
2. Open the source folder, and verify that the source code has been generated.
3. Open the database schema, and verify that the databases are being correctly generated.

5 Create a new project

In order to import the generated source to NetBeans, select `File -> New Project ... -> Java -> Java Application with Existing Sources`. Next, set a name for the project, and in the next step select the source folder with the generated classes. Click finish, and the project will be created.

Note that after creating the project, several errors will appear. Such is due to missing libraries. To solve them, right click the project, then `Properties`. In `Libraries`, select `Add Jar/Folder`, navigate to the project folder /lib, and select the `orm.jar` file. After the import, the errors should have disappear.

At this stage, the base project is set up. It is now possible to take advantage from the generated code. The DAO classes will have a set of methods to handle the persistency, as for instance `save`, `getByORMID` or `listByQuery`. Next are presented some examples on how to use the code. For instance, to save an entity:

```
A a = new A();
try {
    ADAO.save(a);
} catch (Exception e) {
    //handle errors...
}
```

To load an entity it is possible to use the following code.

```
try {
    A b = ADAO.getAByORMID(1);
    System.out.println("Loaded A with id " + b.getID());
    //...
} catch (Exception e) {
    //handle errors...
}
```

To list all the entities, it is possible to use following code.

```
try {
    A[] r = ADAO.listAByQuery("id>0", "ID");
    System.out.println("Retrieved "+r.length+" entries");
    //...
} catch (Exception e) {
    //handle errors...
}
```

Tasks

1. Create a new project, and import the generated code.
2. Solve possible project errors.
3. Run a minimal example: create, save, load, and list `User`.
4. Implement the remaining features of the GMS system (c.f. Section 1).