Development of Web Applications using the Fénix Framework and Layer Architecture

Introduction

The goal of this document is to help students to develop the project of the Software Engineering course. This document mainly describes the following:

- The Fénix Framework
- The DML language
- The maven project to be used to develop an application in this course that uses the Fénix Framework
- The layered architecture to be applied in the project

Lesson 1: Welcome to the Fénix Framework project

Fénix Framework allows the development of Java-based applications that need a **transactional and persistent domain model**. Even though it was originally created to support the development of web applications, it may be used to develop any other kind of applications that need to keep a persistent set of data.

One of the major design goals underlying the development of the Fénix Framework is that it should provide a **natural programming model** to programmers used to develop plain Java programs without persistence. Often, the addition of persistence, typically backed up by a relational database management system, interferes with the normal coding patterns of object-oriented programming, because the need to access the database efficiently precludes the use of many of the powerful mechanisms available in the object-oriented paradigm.

Other frameworks and tools, often called Object-Relational Mapping (O/RM) frameworks, address this problem by giving the programmer ways to specify how objects are mapped into the relational database and by doing most of the heavy lifting needed to access the data in the database. Yet, they fail to completely hide the presence of the database, meaning that the problem remains. With the Fénix Framework, on the other hand, the relational concerns with database are completely hidden from the programmer.

This has two consequences:

- The programmer no longer can control the mapping of objects to the database;
- There is no way to take advantage of database facilities, such as joins or aggregate functions.

In return, programmers may, and are encouraged to, use all of the normal object-oriented programming coding patterns.

Since version 2.x, the Fénix Framework provides several backends to support transactions and the persistency of data. The one we are going to apply in the project for Software Engineering is based on a Software Transactional Memory library and the data is stored in a relational database system.

Programmers using the Fénix Framework specify the structure of the application's domain model using a new domain-specific language created for that purpose, the Domain Modeling Language (DML), and then develop the rest of the application in plain Java. No configuration files, XML files, or property files are needed.

In a nutshell, to use the Fénix Framework programmers have to do the following:

1. Define the structure of the domain model in DML;
2. Write the rest of the program in plain Java;
3. Use the *@Atomic* annotation on the methods that should execute atomically;
4. Create an empty database;
5. Build the project and run the application.

Lesson 2: What is this DML thing anyway?

In every software application, there is a domain model that describes the entities and their relations that the software application intends to abstract and manipulate.

Using the Object-Oriented Programming (OOP) paradigm, programmers may easily specify the domain model of a software application by structuring those entities as classes from which new instances can be created. OO programming languages, like Java, offer a set of constructs (e.g. classes, composition and inheritance) that programmers can use to implement a domain model. However, these general-purpose languages do not provide specific declarative primitive constructs to specify domain models that would help programmers to easily specify the domain model of their applications, such as the storage in a database or the access by multiple concurrent threads in a consistent way. A language, specifically designed for this purpose, is called a **Domain Modeling Language (DML)**.

The Fénix Framework includes one of such languages. It provides a persistent and transactional infrastructure that eases the development of Java applications, and it also provides a new language — called the Domain Modeling Language (DML) — which complements and integrates with the Java programming language.

The reason for using this framework is that Java programmers must manually implement all the relations between classes in an ad-hoc manner, which is a time-consuming and error-prone activity. Also, Java does not provide means for creating new syntactic structures to extend Java constructs in order to support these mechanisms.

The Fénix Framework automatically translates the DML specification into the necessary classes that fully implement relations as described by the developers, liberating them from the repetitive burden that would encompass the translation of the domain model into Java source code, which would be responsible for actually enforcing and delivering the desired structural relations. Traditionally, to provide the required structural aspects, code conventions are defined. However, these conventions need to be applied manually by the developers and must be consistently used across the entire project to avoid errors. This is especially time-consuming when persistent and transactional behavior is needed. The generated classes, derived from the DML specification, can then be extended through the inheritance mechanism already available in an OO language like Java.

The Fénix Framework and its DML do not replace Java. Instead, they integrate with Java providing constructs to better represent a domain model's structure, including its structural aspects like relations and leaving all the rest (like behavior) to Java (and the developer). In fact, a domain model is not completed without its behavior and that is not supported by the DML and needs to be later implemented in Java. This includes business logic and domain restrictions.

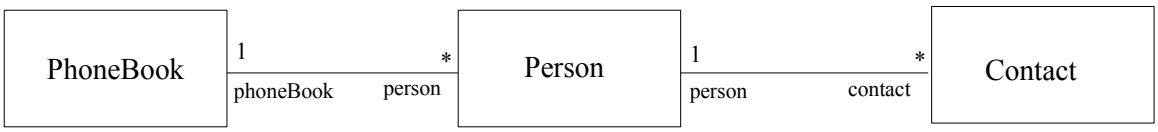
The DML language is considered a micro-language designed specifically to implement the structure of a domain model. This language has constructs for specifying both entity types and relations between them. Based on the DML specification, a compiler can automatically generate code to be integrated with a 'regular' programming language, such as Java. The DML allows Java programmers to specify the domain model using a syntax that is very similar to Java itself. The DML compiler later reads the set of DML specifications that encode the structure of the domain model and translates it into a set of Java source files.

Lesson 3: The PhoneBook Example

Let us imagine that we want to build a simple PhoneBook application, capable of storing a set of contacts for each registered person. A person has a name and each contact is represented by a name and a phone number.

Lesson 3.1: Describing the Domain

The domain model of the PhoneBook application consists only of three entities: `PhoneBook`, `Person` and `Contact`. The `PhoneBook` entity represents the class that knows all registered people. Each person maintains his/her own set of contacts. The relationship between these entities is depicted in the following class diagram:



Lesson 3.2: Describing entities using the DML

Using the DML, we could textually describe the domain model of our PhoneBook application through the following DML code:

```
class PhoneBook;


class Person {
  String name;
}


class Contact {
  String name;
  String phoneNumber;
}
```

The class of the main object of our PhoneBook application is defined in line 1. Since the PhoneBook class has no fields, we may omit the empty bracket block. On the other hand, when the class to be defined has one or more fields, then the fields must be specified inside a bracket block as exemplified in the definition of the Contact class (lines 7 to 10). The built-in value types (those you can use without declaring them) that the DML supports are:

- The Java primitive types: boolean, byte, char, short, int, float, long, and double.
- The wrapper types for the respective Java primitive types: `Boolean, Byte, Character, Short, Integer, Float, Long, and Double`. These types are typically used when an entity's field is optional, i.e., whether null value is allowed.
- The `String` type.
- The `bytearray` type (it will map into the Java type byte[]).
- The `Serializable` type.
- To represent dates and their related types, there are the following built-in value-types, from the Joda Time (Java date and time API): `DateTime`, `LocalDate`, `LocalTime`, and `Partial`. See the documentation for these at http://joda-time.sourceforge.net/.

- Finally, the GSON's `JsonElement` to represent unstructured data.

Lesson 3.3: Describing relations using the DML

The DML also allows the programmer to specify the relations between the several entities of the domain model of the application. Usually this is made after the description of the structure of the classes of the domain model since the classes need to be declared (with 'class') before they are referred in the specification of the existing relations. Considering the PhoneBook application, now that we have described the structure of each class, we can describe the two bi-directional relations existing in this example. Using the DML, we can easily describe the relation between PhoneBook and Person using the following code:

```
relation PhoneBookContainsPersons {
  PhoneBook playsRole phoneBook;
  Person playsRole person {
    multiplicity *;
  }
}
```

In DML, a relation between two classes is specified using the ***relation*** construct. By default, all relations in DML are bi-directional. After naming the relation, we have to define the two roles of the participating entities in the bi-directional relation (through the ***playsRole*** construct), and their respective cardinality. By default, an empty multiplicity block implies a *0 or 1* multiplicity.

This means that line 12 could also be written as:

```
12.  PhoneBook playsRole phonebook { multiplicity 0..1; };
```

Finally, the whole DML specification describing the domain model of the PhoneBook application must be included in a file (e.g., `phonebook.dml`) and it should be stored in the specific directory `src/main/dml` of the Maven project:

```
1. package pt.tecnico.phonebook.domain;
2.
3. class PhoneBook;
4.
5. class Person {
6.   String name;
7. }
8.
9. class Contact {
10.   String name;
11.   Integer phoneNumber;
12. }
13.
14. relation PhoneBookContainsPersons {
15.   PhoneBook playsRole phoneBook;
```

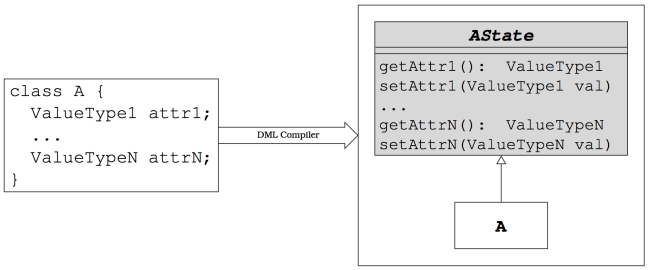
```
16.   Person playsRole person {
17.     multiplicity *;
18.   }
19. }
20.
21. relation PersonContainsContacts {
22.   Person playsRole person;
23.   Contact playsRole contact {
24.     multiplicity *;
25.   }
26. }
```

Note the package definition in line 1. This definition allows the programmer to set the package for all the classes described in the DML specification file. Alternatively, one could provide fully-qualified names for all the specified classes, but in such situations the relations also require the declaration of the fully-qualified name of the classes involved.

```
1. class pt.ist.phonebook.domain.PhoneBook;
2.
3. class pt.ist.phonebook.domain.Person {
4.   String name;
5. }
6.
7. class pt.ist.phonebook.domain.Contact {
8.   String name;
9.   Integer phoneNumber;
10. }
11.
12. relation PhoneBookContainsPersons {
13.   pt.ist.phonebook.domain.PhoneBook playsRole phoneBook;
14.   pt.ist.phonebook.domain.Person playsRole person {
15.     multiplicity *;
16.   }
17. }
18.
19. relation PersonContainsContacts {
20.   pt.ist.phonebook.domain.Person playsRole person;
21.   pt.ist.phonebook.domain.Contact playsRole contact {
22.     multiplicity *;
23.   }
24. }
```

The Fénix Framework compiler will then read our DML file, and automatically generate the classes and the relations described therein.

The class that the DML compiler generates (containing only the entity's state) from an entity type specification in DML has both a getter method and a setter method for each of the entity type's fields. The DML compiler generates the names of these methods by concatenating the prefixes *get* and *set*, respectively, with the name of the field, after capitalizing the first letter of the attribute. So, given an entity object `obj`, the method call `obj.getAttrName()` returns the `obj`'s value for the field named `attrName`, whereas the call `obj.setAttrName(newVal)` sets the value of the field to the value `newVal`. No other methods can access the field.



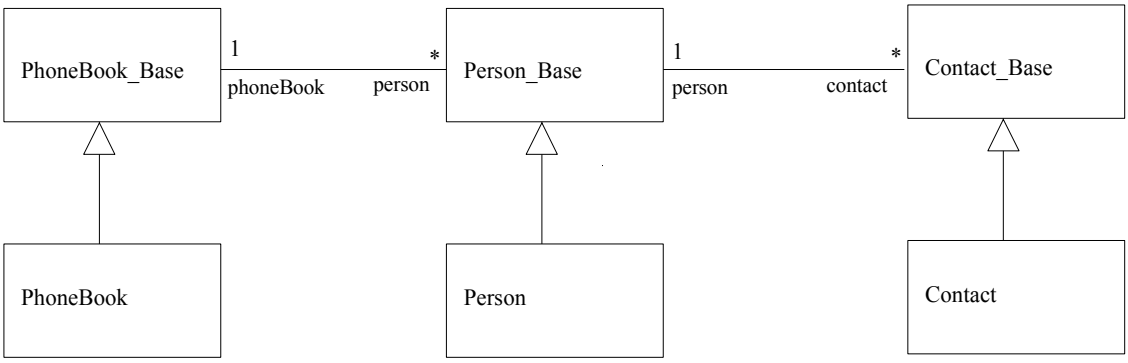
In order to separate the domain business logic concerns from the data and its getters and setters, the Fénix Framework generates two different types of classes for each entity specified in the DML file:

- The ***base*** class (also called the *state* class) **is abstract** and implements the domain entity's structural aspects. The DML compiler automatically generates this class from the domain specification. Thus, programmers should not edit this class manually. The base class is provided with methods that allow clients of this class to access the fields and relations of this class specified in the domain model.

- The ***behavior*** class **extends** the *base* class and should implement the domain entity's behavioral aspects. The DML compiler cannot automatically generate this class, unlike the state class, because the domain specification does not have any behavior specification. Instead, the implementation of this class is the responsibility of the developers: this is the class where programmers implement the entity's behavior. If this class does not exist yet, the DML compiler automatically generates a file that contains an empty class that represents the behavior class.

Even though a state class has all the fields of an entity type and no abstract method, it must be abstract, because it does not represent an entity. An entity contains both state and behavior, but the instances of a state class only contain the state. Therefore, the state class is abstract to prevent the creation of instances of an incomplete custom-made type representing the entity.

Using the PhoneBook example, the following class diagram illustrates the inheritance pattern that the Fénix Framework uses in the definition of domain classes.



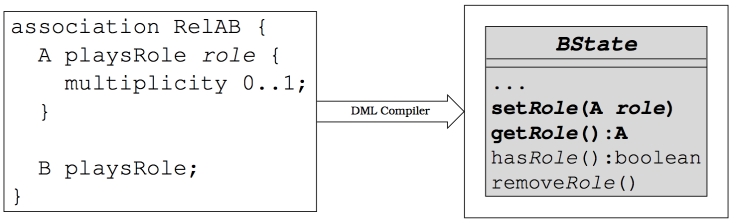
Regardless of the multiplicities involved, the methods that implement a relation must allow us to do each one of the following tasks:

- create a new link between two objects;
- remove an existing link between two objects
- traverse from one object in one end of a link to the object on the other end.
-

The specific signature of the methods that allow us to do this depends on the multiplicity of each role. Let us see each case separately.

3.3.1: Roles with a multiplicity upper-bound of one

If the multiplicity of a role declaration has an upper-bound of one, then an object of the opposite type may refer to at most one object of the role's type. This is the simplest case of a relation, which is typically implemented with a getter and a setter method.



The method `setRole` is the setter method. It allows us to create or to delete a link between an instance of class B and an instance of class A. The call to the setter method with a null argument eliminates any current link that might exist.

The method `getRole` is the getter method. This method is the method that allows us to traverse the relation.

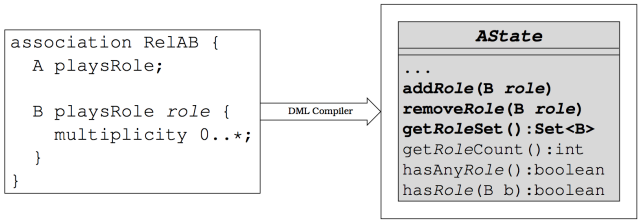
The method `hasRole` returns `true` if there is a link between the receiver of the method call and an instance of class A. Otherwise, it returns `false`. It is equivalent to `obj.getRole() != null`.

The method `removeRole` removes an existing link that might exist for the receiver of the method call. It is equivalent to call the setter method with a `null` argument.

These last two methods, `hasRole` and `removeRole`, are no longer generated in the current version of Fénix Framework.

3.3.2: Roles with a multiplicity upper-bound greater than one

The difference between this case and the previous one is that the object of the opposite type may have multiple links, at the same time, with objects of the role's type. So, when we create a new link, we do not replace any existing link, as in the previous case. The existing links must be removed explicitly, by specifying which object is on the other end of the link that should be removed. Furthermore, when we traverse the relation, we may reach multiple objects. Thus, the getter method must return a collection of objects, in this case.



The method `addRole` creates a new link between the receiver of the method and an instance of the class B passed as argument. If the argument of the method is `null`, the method does nothing. The method `removeRole` deletes the link between the receiver of the method and the instance of the class B passed as argument. If no such link exists, or if the argument is `null`, the method call has no effect.

The method `getRoleSet` returns the set of instances of the class B that have a link with the receiver of the method. The value returned by this method is always an instance of a class that implements the `java.util.Set` interface.

The remaining three methods are equivalent to the following expressions:

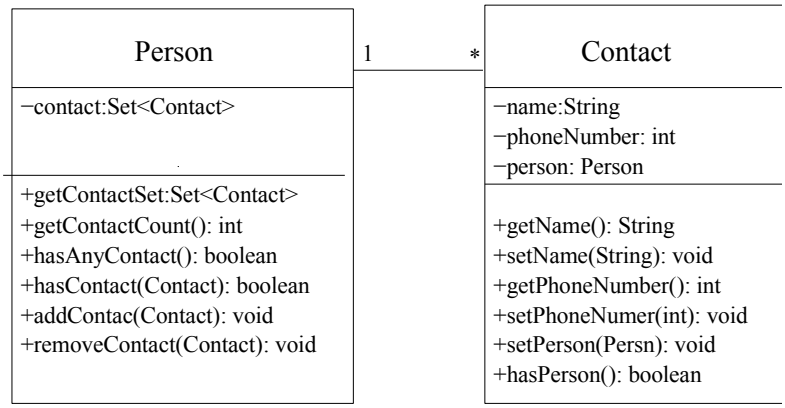
`obj.getRoleCount()` is equivalent to the expression `obj.getRoleSet().size()`

`obj.hasAnyRole()` is equivalent to the expression `(! obj.getRoleSet().isEmpty())`

`obj.hasRole(b)` is equivalent to the expression `obj.getRoleSet().contains(b)`

In the current version of the Fénix Framework, the `hasRole()` and `hasAnyRole()` methods are no longer generated and the `getRoleCount()` is deprecated and therefore it should not be used.

Including all this in our example, we get the following class diagram, concerning just the `Person` and `Contact` classes:



You must read the documentation concerning the DML, available in https://fenix-framework.github.io/DML.html

You may find basic documentation in this excerpt[1] of the PhD thesis that introduced the DML. More advanced (and newly) aspects of the DML language are described in this doc.

Lesson 3.3: Defining the root object

The applications developed using the Fénix Framework must have a special object called the ***root*** object. The root object provides a way to access the persistent state of the application. It is the entry point of the application persistent state. In the Fénix framework, the root object is always the single instance of the *DomainRoot* class. This class is already provided by the Fénix Framework. Then, you just need to connect the domain model of your application with the *DomainRoot* class. In this example, we just need to connect the *PhoneBook* class with the *DomainRoot* class. Since there should be a single PhoneBook instance in the PhoneBook application, we need to establish a one-to-one relation between the PhoneBook and DomainRoot classes. Therefore, we need to add the following declaration to the DML file of this application:

```
1. relation PersonContainsContacts {
2.   PhoneBook playsRole phonebook { multiplicity 0..1 };
3.   .pt.ist.fenixframework.DomainRoot playsRole root { multiplicity 0..1; }
4. }
```

Lesson 3.4: Generating the Fénix Framework related code

The process of building an application that uses the Fénix Framework is relatively complex due to both code generation tasks and the required post-processing steps applied to the compiled classes. However, this complex building process can be eased and simplified by using a build tool such as Maven where you can use specific plugins defined by the Fénix team for executing the Fénix Framework specific tasks.

Consider the following POM file for the *PhoneBook* application:

```
1. <project                                          xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.                                       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.   <modelVersion>4.0.0</modelVersion>
4.
5.   <groupId>pt.tecnico</groupId>
6.   <artifactId>phonebook</artifactId>
7.   <version>1.0-SNAPSHOT</version>
8.   <packaging>jar</packaging>
9.
10.  <name>phonebook</name>
11.
12.  <properties>
13.        <code.generator.class>pt.ist.fenixframework.backend.jvstmojb.codeGenerator.FenixCo
   deGeneratorOneBoxPerObject</code.generator.class>
14.
15.    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
16.    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
```

```
17.     <version.fenixframework>2.6.0</version.fenixframework>
18.     <version.junit>4.11</version.junit>
19.     <version.maven.build-helper-plugin>1.7</version.maven.build-helper-plugin>
20.     <version.slf4j.api>1.7.2</version.slf4j.api>
21.     <version.org.jdom.jdom>2.0.5</version.org.jdom.jdom>
22.   </properties>
23.
24.   <build>
25.     <plugins>
26.       <plugin>
27.         <groupId>pt.ist</groupId>
28.         <artifactId>ff-maven-plugin</artifactId>
29.         <version>${version.fenixframework}</version>
30.         <configuration>
31.           <codeGeneratorClassName>${code.generator.class}</codeGeneratorClassName>
32.           <params>
33.             <ptIstTxIntrospectorEnable>false</ptIstTxIntrospectorEnable>
34.           </params>
35.         </configuration>
36.         <executions>
37.           <execution>
38.             <goals>
39.               <goal>ff-generate-domain</goal>
40.               <goal>ff-post-compile</goal>
41.               <goal>ff-process-atomic-annotations</goal>
42.               <goal>ff-test-post-compile</goal>
43.               <goal>ff-test-process-atomic-annotations</goal>
44.             </goals>
45.           </execution>
46.         </executions>
47.         <dependencies>
48.           <dependency>
49.             <groupId>pt.ist</groupId>
50.             <artifactId>fenix-framework-backend-jvstm-ojb-code-generator</artifactId>
51.             <version>${version.fenixframework}</version>
52.           </dependency>
53.         </dependencies>
54.       </plugin>
55.
56.       <plugin>
57.         <groupId>org.apache.maven.plugins</groupId>
```

```
58.       <artifactId>maven-compiler-plugin</artifactId>
59.       <version>3.1</version>
60.       <configuration>
61.         <source>1.7</source>
62.         <target>1.7</target>
63.       </configuration>
64.     </plugin>
65.
66.     <plugin>
67.       <groupId>org.codehaus.mojo</groupId>
68.       <artifactId>exec-maven-plugin</artifactId>
69.       <version>1.3.2</version>
70.       <executions>
71.         <execution>
72.           <goals>
73.             <goal>java</goal>
74.           </goals>
75.         </execution>
76.       </executions>
77.       <configuration>
78.         <mainClass>pt.ulisboa.phonebook.PhoneBookApplication</mainClass>
79.         <killAfter>-1</killAfter>
80.       </configuration>
81.     </plugin>
82.   </plugins>
83. </build>
84.
85. <dependencies>
86.   <dependency>
87.     <groupId>pt.ist</groupId>
88.     <artifactId>fenix-framework-backend-jvstm-ojb-runtime</artifactId>
89.     <version>${version.fenixframework}</version>
90.   </dependency>
91.   <dependency>
92.     <groupId>junit</groupId>
93.     <artifactId>junit</artifactId>
94.     <version>${version.junit}</version>
95.     <scope>test</scope>
96.   </dependency>
97.   <dependency>
98.     <groupId>org.slf4j</groupId>
```

```
99.       <artifactId>slf4j-log4j12</artifactId>
100.      <version>${version.slf4j.api}</version>
101.    </dependency>
102.  </dependencies>
103.
104.  <repositories>
105.    <repository>
106.      <id>fenix-ashes-maven-repository</id>
107.      <url>https://fenix-ashes.ist.utl.pt/maven-public</url>
108.    </repository>
109.  </repositories>
110.
111.  <pluginRepositories>
112.    <pluginRepository>
113.      <id>fenix-ashes-maven-repository</id>
114.      <url>https://fenix-ashes.ist.utl.pt/maven-public</url>
115.    </pluginRepository>
116.  </pluginRepositories>
117.
118.  </project>
```

This POM file can be used for any Maven project that should use the Fénix Framework. You only need to change the values defined for the *groupId* and *artifactId* elements (lines 5 and 6 of `pom.xml`) of the new project. You also need to change the Java class to execute (see line 78) when the goal `exec:java` is executed. This file also specifies the execution of three goals that are specific to the Fénix Framework code:

- **`ff-generate-domain`** corresponds to the taks that processes the DML file and generates the *base* classes, and the empty *behavior* classes (if they do not already exist).
- **`ff- post-compile`** post processes the generated classes to inject Java bytecode relevant to the Fénix Framework.
- **`ff-process-atomic-annotations`** adds transactional behavior to methods marked with the `@Atomic` annotation.

You also have to define the directory structure of your Maven project. Considering the *PhoneBook* application, the first thing to do is to create the home directory of this Maven project and then copy this POM file to this directory. Then, you need to create the standard directory structure of a Maven project. In this case, it is necessary to create `src/main/java/pt/tecnico/phonebook` and `src/test/java/pt/tecnico/phonebook`. You also have to create the directory `src/main/dml`. This directory will hold the *dml* file of this project. Finally, you will have to create `src/main/resources` directory. This directory contains the file `fenix-framework-jvstm-ojb.properties`. This file specifies the information needed by the Fénix Framework to connect to the database system that stores the persistent data of the application. You need to specify the values for three properties that specify the parameters needed to connect to the database:

- *dbAlias* represents the name of the database that holds the persistent state of the application.
- *dbUsername* is the username of the user used to access to the database.
- *dbPassword* is the password of the user specified in "dbUsername".

For example, this file could have the following (using the *root* user and *rootroot* as the password for this user):

```
1. # using special INFER_APP_NAME to do just that
2. appName=INFER_APP_NAME
3.
4. dbAlias=//localhost:3306/phonebook?useUnicode=true&amp;characterEncoding=UTF-
   8&amp;clobCharacterEncoding=UTF-8&amp;zeroDateTimeBehavior=convertToNull
5. dbUsername=root
6. dbPassword=rootroot
```

Notice that you need to create the *phonebook* database in your database system before starting the execution of the PhoneBook application. If you use the *mysql* client to access to the database system, you should execute the following sql command: "*create database phonebook;*".

Lesson 3.5: Implementing the Business Logic

To implement the business logic, first you need to generate the *base* classes that represent the structural aspects of the application domain model. This can be achieved by executing `'maven compile'`. This Maven lifecycle phase was customized and now this phase also processes the given DML file that describes the domain model of our application.

As we saw, the name of a generated base class is equal to the name of the entity of the application domain model it represents with the *_Base* suffix. The name of the behavior classes is equal to the name of the entities they represent in the application domain model. These classes are generated in the *target* directory of the Maven project since they correspond to code automatically generated during the build of the project.

The first time this phase is executed, it also generates the empty behavior classes that should hold the domain logic of our application. These classes (`PhoneBook.java`, `Person.java` and `Contact.java`) are located at the `src/main/java/pt/tecnico/phonebook/domain` directory since the package of the application domain specified in the DML file was `pt.tecnico.phonebook.domain`.

Given that we only defined three classes in our domain model, we will make a simple example by just editing the empty `Contact` class:

```
1. public class Contact extends Contact_Base {
2.
3.   public Contact() {
4.     super();
5.   }
6.
7. }
```

The previous source code exemplifies the empty class generated by the DML compiler. It will only be created if this file does not exist when the *base* classes are generated. That way, any implemented business logic that we have already implemented meanwhile is not deleted.

In order to create new contacts and associate them values for their name and their phone number fields, we must modify the empty generated constructor of our `Contact` class to accept such parameters and use the inherited setters from the `Contact_Base` class to set their value:

```
1. public class Contact extends Contact_Base {
2.
3.   public Contact(String name, String phoneNumber) {
4.     this.setName(name);
5.     this.setPhoneNumber(phoneNumber);
6.   }
7.
8. }
```

Apart from the inherited data field accessors (setters and getters) defined in the DML file, the DML compiler also generates the source code for the relations defined in the same file.

In this case, we can call the `getPerson()` method on a `Contact` instance that will return the `Person` instance holding that `Contact` instance.

On the `Person` class, we can call the `addContact(...)` method, inherited from the `Person_base` class, to add a new contact to the `PhoneBook` instance.

The Fénix Framework implements these methods considering the bi-directionality of the relation. Therefore, we only need to call either the `setPerson(...)` method in a `Contact` instance, or the `addContact(...)` in a `Person` instance to establish a relation between a these two instances, because the result is the same.

See the Fénix Framework documentation for a better understanding on the rules used to generate the *accessor* and *relation* methods made available for each *behavior* class via inheritance of the respective *base* class.

As mentioned before, the business logic of the application should be implemented as methods of the *behavior* classes. For instance, suppose that there was a business rule in the `PhoneBook` application that specified that

> "a `Person` instance can not have two or more `Contact` instances with the same name".

Since the Fénix Framework generates methods for managing the association between the `Person` and `Contact` instances, we should override these methods to ensure this business rule. Therefore, in this case, to implement this business rule, we must override the *`addContact`* method in the `Person` class as follows:

```
@Override
public void addContact(Contact newContact) {
  for (Contact c : getContactSet()) {
    if (c.getName().equals(newContact.getName())) {
      throw new IllegalArgumentException();
    }
  }
  super.addContact(newContact);
}
```

and, since we also need to ensure this business rule if the `setPerson` if invoked on a Contact this method also needs to be overridden. In order to avoid duplication of code, the best solution is to invoke the `addContact` method. This way the business rule is implemented in a single method:

```
1. @Override
public void setPerson(Person person) {
  person.addContact(this);
}
```

There are other business rules that should be implemented. For instance, consider that (1) it should be possible to remove a `Contact` given its name, and (2) getting all `Contact` instances whose name contains a given substring. These would correspond to the implementation of methods in the `Person` class with the desired functionality.

Lesson 4: Developing a Fénix Framework Application

Developing an application that uses the Fénix Framework requires several steps:

1. Define the domain model in a DML file;
2. Automatically generate the domain classes from the domain model (it creates the *base* classes and empty *behavior* classes, for those not found at compile time); Execute the following Maven command "*mvn compile*" once the DML file is defined and stored in the right place. If there is an error during the execution of this command, then usually this means that the DML file has an error;
3. Implement application logic;
4. Compile all application classes using again the same command "*mvn compile*";
5. Instrument the generated application classes and process the `@Atomic` annotation, as required (**otherwise, there will be run-time errors**). This is automatically made when the *test* phase is executed. Therefore, you need to run the following Maven command: "*mvn test*";
6. Run the application. In this example, running the application corresponds to the invocation of the `main` method of the `PhoneBookApplication` class. The *java* goal of the *exec* plugin was customized in the POM file of this project to invoke this class. Therefore, to run the application you just need to give the command "*mvn exec:java*". Notice that this command should only be executed after the invocation of the test phase, otherwise some error will occur since the code of the application was not instrumented as required.

As we saw before, the Fénix Framework provides tools for the class generation and instrumentation tasks and they are automatically invoked during the build process of this Maven project.

Lesson 5: Populating Initial Data

Some applications may require the preexistence of initial data before they can run for the first time. For such cases, we need to initialize the persistent state of the application with the initial data, as required. In our `PhoneBook` application, we can populate the SOS contact by creating a `SetupDomain` class for that purpose:

```
1. package pt.ist.phonebook;
2.
3. import pt.ist.fenixframework.Config;
4. import pt.ist.fenixframework.FenixFramework;
5. import jvstm.Atomic;
6.
7. import pt.ist.phonebook.domain.PhoneBook;
8. import pt.ist.phonebook.domain.Contact;
9.
10. public class SetupDomain {
11.
27.   @Atomic
28.   public static void main(String[] args) {
29.     populateDomain();
30.   }
31.
32.   static void populateDomain() {
33.     PhoneBook pb = Phonebook.getInstance();
34.     Person person = new Person("Manel");
35.     pb.addPerson(person);
36.     person.addContact(new Contact("SOS", 112));
37.     person.addContact(new Contact("IST", 214315112));
38.   }
39. }
```

This class is responsible for setting up the initial persistent state of the PhoneBook application.

Notice that it is necessary to have a relational database system running (mysql for instance) to persistently store the application data, The Fénix Framework accesses to this database system using the given database, user and password specified in the `fenix-framework-jvstm-ojb.properties` file present in the resources directory of this Maven project. This user must have access to the database in the database system (the last word in specified in the dbAlias parameter).

Transactional support in the Fénix Framework

An important requirement of applications developed using the Fénix Framework is that all accesses to the persistent state of the application (i.e., every read and/or write operation to a *base* class belonging to the application domain model) must be done in the context of a transaction. The Fénix Framework provides transactional support through three mechanisms:

- Standard *begin*, *commit* and *rollback* operations common to any transactional system;
- Atomic invocation of a *Callable*;
- Atomic invocation of methods that use the @Atomic annotation.

Low-level mechanism for transaction management

The `pt.ist.fenixframework.TransactionManager` interface class provides the low-level API for managing a transaction. Using this low-level API, programmers have a fine-grained control over the transactional code. However, this transactional mechanism adds complexity to the code. This interface extends the `javax.transaction.TransactionManager` interface, which means that any application already using the Java Transactions API can be easily ported to use the Fénix Framework.

This interface provides the following functionality. Starting a new transaction requires the invocation of the `begin()` method. This transaction will be active, within the current thread, until either the transaction commits (corresponding to the invocation of `commit()`method) or aborts (corresponding to the invocation of the `rollback()`method).

When a programmer wants to finish the current transaction, making any changes made in the context of the current transaction available to others, he must invoke the `commit()` method on the `TransactionManager` of the Fénix Framework. This method will commit any changes performed on the active transaction, within the current thread, and close the transaction.

If the programmer sees fit to abnormally terminate the current transaction, then he must invoke the `rollback()` method. This method will close the transaction, discarding any changes that the current transaction would have made to the persistent state of the application.

Here is an excerpt of code that exemplifies the use of this low-level mechanism. You can also look at the `main()` method of `PhoneBookApplication` to see another example.

```
1.  import pt.ist.fenixframework.FenixFramework;
2.  import pt.ist.fenixframework.TransactionManager;
3.
4.  public class SomeClass {
5.    // ...
6.
7.    void someMethod() {
8.      // non-transactional code
9.
10.     TransactionManager tm = FenixFramework.getTransactionManager();
11.     tm.begin();
12.     boolean txCommited = false;
13.     try {
```

```
14.       // transactional code
15.       tm.commit();
16.       txCommited = true;
17.     } catch(SystemException | NotSupportedException | RollbackException|
18.             HeuristicMixedException | HeuristicRollbackException ex) {
19.             System.err.println("Error in execution of transaction: " + ex);
20.             // exception handling code here
21.     } finally {
22.       if (! txCommited) {
23.         try {
24.           tm.rollback();
25.         } catch (SystemException ex) {
26.           System.err.println("Error in roll back of transaction: " + ex);
27.           // exception handling code here
28.         }
29.       }
30.     }
31.     // non-transactional code
32.   }
33.
34.   // ...
35. }
```

Transactional commands

Another way to provide transactional support is programming the behavior that needs to be transactional within the `call()` method of a `java.util.concurrent.Callable` subclass (typically as an inner class) and give a instance of this class to the `withTransaction()` method of `TransactionManager` as we can see in the next example.

```
1.  import java.util.concurrent.Callable;
2.  import pt.ist.fenixframework.TransactionManager;
3.
4.  public class SomeClass {
5.    // ...
6.
7.    void someMethod() {
8.      // non-transactional code
9.
10.     FenixFramework.getTransactionManager().withTransaction(new Callable() {
11.       @Override
12.       public Object call() {
```

```
13.         // transactional code
14.       }
15.     });
16.
17.     // non-transactional code
18.   }
19.
20.   // ...
21. }
```

The code specified inside the `call()` method will always be executed in the context of a transaction that is automatically created and managed by the Fénix Framework when the `withTransaction()` method is invoked.

@Atomic anotation

Finally, The easiest way to execute a method in its own transaction is to simply annotate it with the `@Atomic` annotation. During the *post-compile* phase, the annotated methods are automatically surrounded by transaction management code that uses the lower-level APIs transactional mechanism. Every method marked with this annotation is automatically executed in the context of a transaction.

```
1. import pt.ist.fenixframework.Atomic;
2.
3. public class SomeClass {
4.   //...
5.
6.   @Atomic
7.   void atomicMethod() {
8.     // transactional code
9.   }
10.
11.  // ...
12. }
```

Notice that the Fénix Framework **does not support nested transactions** (transactions occurring inside other transactions). This means that a transactional method must not invoke another transactional method.

Lesson 5.1: Accessing the Root Object

The root object of a Fénix Framework application is the entry point in the persistent state of the application. This is always the first persistent object to be accessed in the application. All other valid objects of the persistent state are accessed (directly or indirectly) through the root object. The *FenixFramework.getRoot()* static method returns the *root* object of a Fénix Framework application.

The first time the application is executed there are some initializations that are automatically made. One of these initializations concerns the *root* object. The *root* object is initialized with an instance of the domain class specified in the *rootClass* parameter of the `Config` object.

In the `PhoneBook` application, the *main* class is the `PhoneBook` class. There is a single `PhoneBook` instance that knows all `Person` instance. Each person has his own set of contacts. According to the DML file of this application, the single instance of `PhoneBook` is connect to the root object. Thus, given the root object, we can access to the `PhoneBook` instance by invoking the get`PhoneBook()` method.

Lesson 6: Running the Application

After we have defined and implemented the business logic of our application, and populated some initial data, we must find a way to run it.

To run our `PhoneBook` application, we can define a class called `PhoneBookApplication`, which is very similar to the `SetupDomain` class. The `PhoneBookApplication` class defines the *main* method that will be responsible for starting the application. In this example, it will simply initialize the application persistent state if needed, using the `SetupDomain` class defined previously, and print the existing contacts:

```
1.  package pt.ist.phonebook;
2.
3.  import pt.ist.fenixframework.Config;
4.  import pt.ist.fenixframework.FenixFramework;
5.  import jvstm.Atomic;
6.
7.  import pt.ist.phonebook.domain.PhoneBook;
8.  import pt.ist.phonebook.domain.Contact;
9.
10. public class PhoneBookApplication {
11.
40.   @Atomic
41.   public static void main(String[] args) {
42.     PhoneBook pb = PhoneBook.getInstance();
43.     setupIfNeed(pb);
44.     printPhoneBook(pb)
45.   }
46.
47.   private static void printPhoneBook(PhoneBook pb) {
48.     if (pb.getPersonSet().isEmpty())
49.           SetupDomain.populateDomain();
50.   }
51.
52.   public static void printPhoneBook(PhoneBook pb) {
53.     for (Person person : pb.getPersonSet()) {
54.       System.out.println("The Contact book of " + person.getName() + " :");
55.       for(Contact c : person.getContactSet()) {
56.         System.out.println("\t Name: "+ c.getName() + " phone: " +
57.                            c.getPhoneNumber());
58.       }
59.     }
60. }
```

In the PhoneBook application, the access to the root object is encapsulated inside the getInstance() method of the PhoneBook class. Recall, that in the DML we specified that the root object (that is an instance of the DomainRoot class) is connected to 0 or 1 PhoneBook instances. Since there should be a single PhoneBook instance (that holds the all registered Person objects), we apply the Singleton design pattern. The code concerning this aspect of the PhoneBook class is the following:

```
1.  package pt.ist.phonebook.domain;
2.
3.  import pt.ist.fenixframework.FenixFramework;
4.
5.  public class PhoneBook extends PhoneBook_Base {
6.
7.    public static PhoneBook getInstance() {
8.      PhoneBook pb = FenixFramework.getDomainRoot().getPhonebook();
9.        if (pb == null)
10.         pb = new PhoneBook();
11.
12.       return pb;
13.     }
14.
15.     private PhoneBook() {
16.       FenixFramework.getDomainRoot().setPhonebook(this);
17.     }
18.     // …
19. }
20.
```

Access to the root object is achieved through the invocation of `getDomainRoot()` static method of `FenixFramework` class. If the root object does not have a `PhoneBook` instance associated with it (line 10), then we create a `PhoneBook` instance. The constructor of the `PhoneBook` class accesses to the root object and creates an association between the new `PhoneBook` instance and the root object when the `setPhonebook()` method is invoked (line 16).

Once this class is defined and the project is compiled and tested, we can execute the `PhoneBook` application. This can be achieved by executing the *Maven* command '`mvn test exec:java`'. This command first executes the test phase and then executes the goal

`exec:java`. This goal was configured in the POM file to execute the `PhoneBookApplication` class.

Lesson 7: Understanding Layers

In the last step, we have learned how to directly manipulate the domain entities of our PhoneBook application. However, the best practices foster a separation of concerns strategy, which allows us to build software applications that are better structured and maintainable.

One way to achieve such separation of concerns is structuring the architecture of our application into different layers, i.e., building the application using the layered architecture.

In this architectural model, the application is structured into several layers where each layer has a well-defined interface and responsibility. Each layer only needs to communicate with its adjacent layers. This architectural model has some advantages. Modifications done in one layer do not have impact in the other layers, if the interface is preserved. Even if the interface is modified, those changes will only have direct impact to the layers immediately adjacent to it. Usually, given the different concerns of a web application, four different logical layers are defined:

- *Presentation Layer* This layer concerns user interaction and the form how information is presented to the user. For instance, when a user wants to see all existing contacts stored in the PhoneBook application, the presentation layer is responsible for displaying that information to the user with a given format.
- *Service Layer* This layer separates the *Presentation* layer from the *Business Logic* layer. It is responsible for implementing the features provided to the *Presentation* layer that are supported by rules and semantic defined in the domain model of the application. For instance, when a user wants to list all the contacts of a given person stored in the Phonebook application, there is a functionality (called *service*) implemented in the *Service* layer that is responsible for getting the respective information (from the *Business Logic* layer) so that the *Presentation* layer may display it to the user.
- *Business Logic Layer* This layer concerns the implementation of the application business logic. All the application's semantics and business rules should be implemented in the context of this layer. For instance, the business rule that states that "a `person` cannot have two contacts with the same name and phone number" should be implemented in the *addContact* method of the `Person` class (that belongs to the Business Logic layer) and not in the service that is responsible for adding a contact. Using this approach, it is easy to know all the implemented business rules that concern a domain entity. We just need to see the code of the class that implements the concerned entity and we do not need to see the code of all services. Moreover, by placing the business rules in the domain entities, we do not need to replicate them in several services. For instance, if we consider the *add contact* and *rename contact* services, we would need to replicate the unique contact business rule described above in both services.
- *Data Access Layer* This layer supports the persistence and retrieval of data of a software application. For instance, persisting the information relative to a new contact created in the PhooneBook application.

The Fénix Framework concerns the last two layers: the business logic and data access layers. The model of the application state is specified using the DML offered by the Fénix Framework. The programmers then enrich the generated entities from the DML specification in order to implement the specific business rules of the application to develop. The Fénix Framework does not offer any support for the remaining layers.

Next, we will show the proposed solution in the Software Engineering course to implement the service and presentation layers of a Web application using the *PhoneBook* application to illustrate the most important aspects of this solution.

Lesson 8: Defining a Service

If we implement the PhooneBook application as a console application, whenever the user wants to list the existing contacts of a Person, it can execute the command `list`. Understanding the meaning of this command is a concern of the Presentation layer, which we described in the previous section as the layer that deals with the user interaction.

The Presentation layer knows that when the user sends the `list` command, it needs to display information about all the contacts existing of the concerned person in the PhooneBook application. Hence, this listing feature will have to ask the Service Layer to retrieve the required information.

The Service layer corresponds to the Service Layer Pattern presented in the book **Patterns of Enterprise Application Architecture**. The Service layer defines the application boundary. It acts as the entry point of the application for the several interfaces (e.g Web, mobile) that are available to use the application. Usually, this layer controls the transactions and offers the set of functionalities supported by the application core as a set of *services*. Each service represents a functionality of the application and is implemented based on the functionality of the Business Logic layer.

In the architecture proposed by us, each distinct service is implemented as a subclass of the *PhoneBookService* class:

```
1. package pt.ist.phonebook.service;
2.
3. import pt.ist.phonebook.exception.PhoneBookException;
4. import jvstm.Atomic;
5.
6. public abstract class PhoneBookService {
7.
8.         @Atomic
9.         public void execute() throws PhoneBookException {
10.           dispatch();
11.        }
12.
13.        protected abstract void dispatch() throws PhoneBookException;
14. }
```

In the proposed approach, to execute a service you need to invoke the execute method over an instance of a subclass of *PhoneBookService*. This method will execute the code corresponding to the called service in the context of a transaction. This specific code to each type of service supported by the application must be implemented in the *dispatch* method in each *PhoneBookService* subclass. For instance, consider the functionality referred above: list all persons. This functionality should be implemented in a subclass of PhoneBookService, called ListPersonsService, The specific behavior of this service should be implemented in the dispatch method of *ListPersonsService*.

```
1. package pt.ist.phonebook.service;
2.
```

```
3. import java.util.List;
4. import java.util.ArrayList;
5.
6. import pt.ist.fenixframework.FenixFramework;
7. import pt.ist.phonebook.domain.PhoneBook;
8. import pt.ist.phonebook.domain.Person;
9. import pt.ist.phonebook.exception.PhoneBookException;
10. import pt.ist.phonebook.service.dto.PersonSimpleDto;
11. import pt.ist.phonebook.service.dto.PhoneBookDto;
12.
13. public class ListPersonsService extends PhoneBookService {
14.
15.   private PhoneBookDto result;
16.
17.   public final void dispatch() throws PhoneBookException {
18.     PhoneBook pb = FenixFramework.getRoot();
19.     List<PersonSimpleDto> personList = new ArrayList<PersonSimpleDto>();
20.
21.     for(Person p : pb.getPersonSet()) {
22.         PersonSimpleDto view = new PersonSimpleDto(p.getName());
23.         personList.add(view);
24.     }
25.
26.     result = new PhoneBookDto(personList);
27.   }
28.
29.   public PhoneBookDto getResult() {
30.      return result;
31.   }
32.
33. }
```

In this example, it is noteworthy that instead of returning the PhoneBook object as the result of the invocation of this service, the result (obtained through the *getResult* method of *ListPersonsService*) is an instance of *PhoneBookDto*, and not *PhoneBook*. The Presentation layer has to manage data that represents entities from the domain model. The question is how to represent domain entities in the Presentation layer? We cannot use the domain entities in the Presentation layer for several reasons:

- First, this would impose a dependency between the Presentation and the Business Logic layers and that should not happen since these layers are not adjacent.
- Second, usually the classes that implement the application domain model are all related somehow and a domain object may have references (directly or indirectly) to a large number of other domain objects. Since it is not mandatory that the *Service* and *Presentation* layers are in the same address space (and usually they are not), sending a *PhoneBook* instance from the *Service* layer to the *Presentation* layer could imply the transfer of a large amount of data, being most of it not needed by the *Presentation* layer for answering a user request.
- Third, in our specific solution, the domain entities can only be accessed in the context of a transaction and it is not possible to have transactions in the Presentation layer since the code of the Presentation layer is usually executed in a different process (and most probably in a different computer too) than the process that runs the Service and Business Logic layers.

A possible solution for this problem is to apply the Data Transfer Object pattern. The idea behind this design pattern is to represent each type of data exchange with a simple class that just holds data and does not have any behavior (besides methods for accessing the contained data). Data Transfer Objects (a.k.a. DTOs) are reusable classes that contain related data and no business logic. DTOs can be composed of other DTOs. DTOs are separate entities from the domain model whose sole purpose is to define how data is received and/or returned form a service. Data Transfer Objects were first described as a means to reduce the number of method calls in distributed object systems (e.g. CORBA, DCOM).

DTOs can be created in the service side, client side, or both sides of the communication channel. Domain model entities may be mapped into and out of DTOs through custom code or with data-binding technologies. With the former approach, the logic to move data to and from DTOs may be centralized in the DTO itself, or within a dedicated class (defined in the Service layer) that knows how to convert DTOs into domain entities and vice-versa. Using this approach, each service that has to send information to the *Presentation* layer creates a data transfer object, populates it with the required information of the application state and then returns it, as we can see in the next code example.

The extra work of having to define DTOs and the conversion to/from domain entities can be very high when your domain model has hundreds of entities. To make this worse, the number of DTOs may be higher than the number of domain entities since different services may represent the same domain entities with a different detail. In the Phonebook application, we have two DTOs, PersonDTO and PersonSimpleDTO, to represent the Person entity. The first DTO provides all information concerning a Person entity while the second one just holds the name of a person.

For more information concerning this design pattern you can check the following page http://msdn.microsoft.com/en-us/magazine/ee236638.aspx that describes this design pattern in detail.

Hence, a simple class (PhoneBookDto), which simply stores the persons of the PhoneBook application, is defined to contain only the necessary data instead of the overall domain model:

```
1. package pt.ist.phonebook.service.dto;
2.
3. import java.util.List;
4.
5. public class PhoneBookDto {
6.
7.   private List<PersonSimpleDto> personList;
8.
9.   public PhoneBookDto(List<PersonSimpleDto> personList) {
10.    this.personList = personList;
11.  }
12.
13.  public List<PersonSimpleDto> getPersons() {
14.    return this.personList;
15.   }
16. }
```

Although these DTO objects are merely data containers, they can be composed by other DTOs as exemplified above since *PhoneBookDto* contains a list of *PersonSimpleDto*:

Also, a more detailed DTO (containing more detailed information about the object) can extend other simple DTOs referring to the same object. For instance, the PersonDto class, that holds the name and list of contacts of a person, should be a subclass of the PersonSimpleDto that just holds the name of a person.

Finally, as the ListPersonsService contained in the *Service* layer returns the corresponding PhoneBookDto to the Presentation Layer, the *Presentation* layer must know how to display that information (DTO) to the user. For this purpose, a Presenter class may be defined in this layer:

```
1. package pt.ist.phonebook.presentation;
2.
3. import pt.ist.phonebook.service.dto.ContactDetailedDto;
4. import pt.ist.phonebook.service.dto.PhoneBookDto;
5.
6. public class PhoneBookPresenter {
7.     public static void show(PhoneBookDto dto) {
8.         System.out.println("List of Persons:");
9.         for(PersonSimpleDto person : dto.getPersons()) {
10.            PersonSimplePresenter.show(person);
11.        }
12.    }
13. }
```

Lesson 9: Testing Software

No one develops software without bugs. If someone wants to produce software with quality she or he must test the developed code.

Tests can be manual or automatic. In the former case, there is someone that is responsible for testing the application, giving inputs to the application and checking that the application behaves as expected. In the latter case, there is someone, usually called *tester*, which is responsible for implementing a program that checks that the application has the expected behavior. The latter case involves a higher development cost but the cost of checking that the application has the desired behavior is much lower, since it does not require any human interaction.

The JUnit framework is a framework that helps building automatic tests. It is the most used framework for developing unit tests. You can get detailed information about JUnit in its site, http://www.junit.org.

Lesson 10: Implementing the User Interface

The user interface of our application should be developed using the Google Web Toolkit. (GWT). This toolkit allows programmers to develop Ajax web application with Java. Programmers only have to write Java code to specify the user interaction and this code is automatically translated into HTML and Javascript via the GWT compiler.

For a more detailed description of the GWT you should check the tutorial present in https://developers.google.com/web-toolkit/doc/latest/tutorial/.

GWT provides two modes:

- Development Mode: allows programmers to debug the Java code of your application directly via the standard Java debugger.
- Web mode: the application is translated into HTML and Javascript code and can be deployed to a webserver.

All GWT applications run as JavaScript code in the end user's web browser. However, usually your application needs to communicate with a web server, sending requests and receiving updates. The code that runs in the user's web browser is called client-side code and the code that runs in the web server is called server-side code.

Modules and Entry Point

GWT applications are described as modules. Modules are defined in XML and should be placed into your project's package hierarchy. A module *ModuleName* is described by a file *ModuleName.get.xml*. Each module can define one or more **Entry point** classes. Modules may appear in any package in your classpath, although it is strongly recommended that they appear in the root package of a standard project layout. For example, the file that contains the definition of the Phonebook module, named *Phonebook.gwt.xml*, is present in the *pt.ist.phonebook* package.

Any class that implements *com.google.gwt.core.client.EntryPoint* and that has the default constructor can be an entry-point of a GWT module. When a module is loaded, every entry point class of this module is instantiated and its *onModuleLoad()* method is called. Despite the fact that you can define several entry-point classes for a GWT module, it is easier to implement the application if you have a single entry-point. An entry point can be seen as the starting point for a GWT application, similar to the main method in a standard Java program. The entry-points of a module are defined in the entry-point tag specified in the xml module description file. In our example, the *Phonebook.gwt.xml* contains the following:

```
<!-- Specify the app entry point class.                         -->
<entry-point class='pt.ist.phonebook.presentation.client.PhoneBook'/>
```

The module is connected to a HTML page, which is called "host page". The code for a GWT web application executes within this HTML document.

The Host Page

The host HTML page is the first page your clients should visit when they browse to your application and is also where the rest of your application files are loaded from. Any HTML page can include a GWT application via a *<script>* tag. This tag specifies the path of JavaScript source code responsible for the dynamic elements on the page and corresponds to the bootstrap file of your GWT application. This JavaScript is automatically generated by the GWT and it is stored in the *war* directory of the project.

A typical host page for a GWT application might not include any visible HTML body content at all. In this case, it is the responsibility of the GWT application to specify all the visual content. But, GWT applications can also insert widgets into specific places in an HTML page in order to make it easy to add GWT functionality to existing web application with only minor changes. To accomplish this, you should use the *id* attribute in your HTML tags to specify a unique identifier that the GWT application will use to attach widgets to that HTML For example, consider the following host page

```
<body>
    <!-- other sample HTML omitted  -->
    <table align=center>
        <tr>
            <td id=”slot1”/>
            <td id=”slot1”/>
        </td>
    </table>
</body>
```

Notice that the *<td>* tags include an *id* attribute that holds a unique identifier in the context of this HTML page. You can attach widgets to these *<td>* tags using the method *get(String id*) of the *RooPanel* class. For instance, to attach a button and a label to the *<td>* tags, we could have the following code on the *onModuleLoad* method of the Entry point class of the GWT application:

```
public void onModuleLoad() {
    Button button = new Button(“Click here”);
    Label label = new Label(“This is a Label”);
    …
    RootPanel.get(“slot1”).add(button);
    RootPanel.get(“slot1”).add(button);
    …
}
```

The *<td>* elements are just used as placeholders for the dynamically generated portions of the page. Thus, GWT functionality can be added as just part of an existing page, and changing the application layout can be done in plain HTML.

The *Phonebook* application provides two host pages, that are almost equal: *index.html* and *remote.html*. These two host files are used to configure the application to run in *local* mode or *remote* mode. The local mode should be used to run the *ES-only* version of the project while the remote mode should be used to run the *ES+SD* version of the project. The difference between the two host pages is that *index.html* includes a *<td>* element with an *id* attribute equal to *ES-only* while *remote.html* includes a *<td>* element with an *id* attribute assigned to *ES+SD*. The Entry point class of the *Phonebook* application checks whether the host page contains an element with the identifier *ES-only*, and then invokes the *initServer* method of the servlet defined in this GWT application with argument "ES-only" (if host page is index.html) or "ES+SD" (if host page is remote.html).

Organizing your code

To better organize the code of your application, the code that concerns the GWT user interface and the communication with the web server should be placed into the *client* and *server* sub-packages of the *presentation* package. The use of these two packages allows us to distinguish between the client-side code (which is translated into JavaScript) from the server-side code (which is not). If there is code that should be shared by both the client and server-side code, then this code should be placed into another sub-package called *shared.*

The location of the client and shared (if needed) packages is specified in the xml module configuration file through the *source* tag. In our Phonebook example, we have the following:

```
1.  <!-- Specify the paths for translatable code                    -->
2.  <source path='presentation/client'/>
3.  <source path='presentation/shared'/>
```

The specification of the server-side code is defined in another file. Java web applications use a deployment descriptor file to determine how URLs map to servlets, which URLs require authentication, and other information. This file is named *web.xml*, and resides in the app's WAR under the WEB-INF/ directory. The *web.xml* file is part of the servlet standard for web applications. When the web server receives a request for the application, it uses the deployment descriptor to map the URL of the request to the code that ought to handle the request.

In this file, you can specify the existing servlets in your application and the mapping between those servlets and URLs. In the Phonebook application, the following excerpt of the *web.xml* file register the *pt.ist.phonebook.presentation.server.PhoneBookServletImpl* java class as a servlet and associates it with the /phonebook/service URL.

```
2. <servlet>
3.     <servlet-name>phoneBookServlet</servlet-name>
4.     <servlet-class> pt.ist.phonebook.presentation.server.PhoneBookServletImpl
5.     </servlet-class>
6. </servlet>
7. <servlet-mapping>
8.     <servlet-name>phoneBookServlet</servlet-name>
9.     <url-pattern>/phonebook/service</url-pattern>
10. </servlet-mapping>
```

Client and Server Communication (RPC)

Usually, GWT applications running in the user's browser need to interact with the web server (for instance, to load or save data). GWT provides its own remote procedure call (RPC) mechanism to allow the GWT client to call server-side methods. The implementation of GWT RPC is based on the servlet technology.

This mechanism includes generation of efficient client and server side code to serialize and deserialize objects across the network. There are some restrictions concerning the classes that can be used in the remote communication. The restrictions are the following ones:

- The classes must implement the java.io.Serializable interface and have the static field *serialVersionUID* defined.
- The classes must provide the default constructor.

The GWT RPC makes the remote communication almost transparent for the GWT client and always asynchronous so that the client does not block during the communication.

To create a GWT servlet you need to define the following:

- An interface which extends *RemoteService.* This interface defines the methods of this Servlet available for remote invocation. This interface is called the synchronous interface.
- An asynchronous interface to the servlet. It must have the same name as the synchronous interface of the Servlet, appended with *Async*. This interface has the same methods as the synchronous interface. The difference is that all methods have no return type and have an extra parameter (the last one) of type *AsyncCallback*. The client code invokes the server-side code using the asynchronous interface. Both interfaces should be placed in the client *presentation.client* package.
- A class that implements the synchronous interface and extends the *RemoteServiceServlet* class. This class corresponds to the server-side code that is executed whenever the client code executes a remote call. This class should be placed in the *presentation.server* package.

The process of making an RPC from the client always involves the same steps:

- Instantiate the Servlet interface using *GWT.create()*. This method receives as argument the object that represents the synchronous interface of the desired Servlet and returns an object that implements the asynchronous interface. The remote invocations to the Setvlet are made through this instance. You should execute the *GWT.create* method once and store the result somewhere in your code. For instance, in the Phonebook application, we have the following:

```
public class PhoneBook implements EntryPoint {

   private static final String         remoteServerType    = "ES+SD";
```

```
        private static final String         localServerType    = "ES-only";

        private final PhoneBookServletAsync rpcService =
                                            GWT.create(PhoneBookServlet.class);
```

- Before making the remote call, you need to create an asynchronous callback object. All remote invocations are asynchronous. This callback object is used by the GWT to notify the client code when the RPC has completed. The *AsyncCallback* interface defines two methods: *OnSuccess* and *OnFailure*. The former one is invoked when the asynchronous call has completed without any error, while the latter one is invoked when the asynchronous call has finished and some error happened during its execution. The error is given in the argument of this method.
- Invoke the desired *remote* method through the instance created in the first step, specifying the required arguments, which include the callback object created in the second step.

Running the application in development mode

The GWT development mode includes an embedded version of Jetty. Jetty is a pure Java-based HTTP server and Java Servlet container. In the GWT development mode, Jetty acts as a development-time servlet container for testing the application. This makes the testing cycle faster since you do not need to compile and load the application into a Web server. Moreover, this allows programmers to debug both server-side code and client-side code of the GWT application using a Java debugger.

To run a GWT application in development mode inside Eclipse, you first need to select the GWT project that contains the desired application in the *Package Explorer* view. Then, click the Run button in the toolbar and select the *Run as Web Application* option. Another way of running the application is to right-click the GWT project in the *Package Explorer* view and select the option *Run As -> "Web application".* Another way of doing this is to select the GWT project in the *Package Explorer* view and click the *Run* button in the toolbar (and select the option *Run as Web Application*).

This opens a new view, called *Development Mode*. Copy the URL from this view and paste it in your preferred browser. The first time you run a GWT application in development mode, you will need to install the GWT Developer plugin. Follow the instructions on the page to install the plugin, then restart the browser and return to the same URL

If you have updated your *build.xml* file as described in the laboratorial guide, you should also have a new target for running the application in development mode outside Eclipse. Executing this target starts the Jetty HTTP server. Then, you can press the "Launch Default Browser" button to launch your GWT application in development mode using your default browser. Or, you can click "Copy to Clipboard" to copy the launch URL and paste it into the browser of your choice.

For debugging reasons, it is better to run the GWT application via Eclipse instead of via ant target, since usually when the GWT applications have errors, you have more information concerning the errors that happened if you are running the application via Eclipse. Moreover, if you want to used a debugger to help you to correct those errors, you need to run the GWT application via Eclipse, as described next.

Debugging a GWT application

It is easier to debug GWT applications in development mode than in production mode since you can use the debugger of the Eclipse in the former case and debug the just the Java code. This option is not available in production mode since in this case the code running is not the Java code but the Javascript code translated by the GWT compiler. To debug a GWT application in development mode you can just put a breakpoint into the Java code and start the debugger in the Eclipse via selecting your project, right-click -> debug as -> Web Application. For more information about this topic you should check https://developers.google.com/web-toolkit/doc/latest/tutorial/debug.

The End